EXTENDING RISC-V ISA WITH A CUSTOM INSTRUCTION SET EXTENSION

Introduction

RISC-V ISA (Instruction Set Architecture) is designed in a modular way. It means 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 application needs, without having to pay for area or power that will not be used. One of the groups is special; it has no 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 application-specific instructions (custom ISA extension) and how to build all the needed tools in SDK, as well as implementing the custom ISA extension in HDL (e.g. Verilog).

RISC-V Instruction Set Architecture

RISC-V ISA is organized into groups of instructions (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 extension 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 16bit encoding. This extension is very important for applications requiring low memory footprint.
F	*Yes*	Single-precision floating-point instructions
D	*Yes*	Double-precision floating-point instructions

Codasip GmbH – Headquarters
Paul-Gerhardt-Allee 50
81245 Munich, Germany

Codasip s.r.o. – R&D Center
Božetěchova 1/2
612 00 Brno, Czech Republic

Phone: +420 541 141 475
E-mail: contact@codasip.com
Website: www.codasip.com

ISA Extension	Ratified	Notes
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 above 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, 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 below 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 bare-metal 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; 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**.

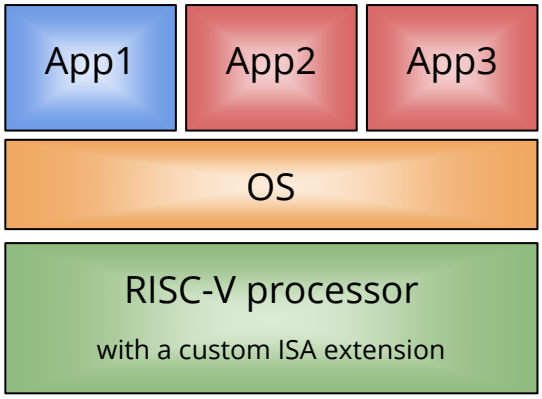

Figure 1

Figure 2 below 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 the 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.

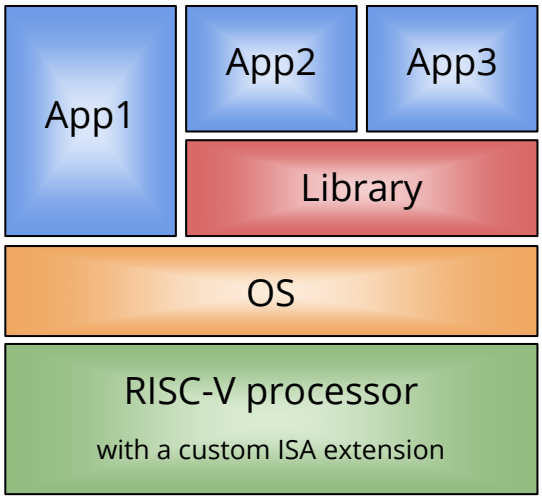

Figure 2

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

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. 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, 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, we need to automate as much as possible, from identification of suitable instructions to RTL verification. This is exactly what Codasip can do 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 full-blown

RISC-V-compliant processor and 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 some special load/store instructions with post- or pre-increments. As you can see, custom instructions differ by complexity. This, among other things, 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. 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 (see below), X10 hold the value of 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.
```

So, not only performance of the application is increased; code size is also significantly smaller. You can have many more similar instructions including pop-count, various bit manipulation instructions, control instructions, etc.

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

Codasip Studio

Studio is an EDA tool for processor design. It can generate all needed 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. Thus, 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 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 needs to be done 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 instructions. There are many ways to do this. Studio uses a profiler. The designer runs the key application using the off-the-shelf processor, and the profiler then provides specific sequences of instructions that may be of

interest, and 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 assembly and binary form of the instructions. Then, more importantly, semantics of the instruction. The semantics is 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 implementation. Let's assume straightforward implementation and do the whole instruction on 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 LLVM compiler, so the new instruction is generated for LLVM. 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.

Let's look at the implementation in HDL. Studio supports three major HDL languages. They are SystemVerilog, Verilog, 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 you can see, Studio produces all necessary output based on the CodAL code. We automated both SDK generation and RTL.

The last step is verification. Studio includes functional view and implementation view. The functional view is used for a reference model, and the implementation view is used as a DUT. In other words, 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.

Conclusion

RISC-V ISA offers a wide range of ISA groups to choose from. Only the basic group (I/E) is mandatory, the rest is optional. The concept allows designers to select precisely what they want or need. If this is still insufficient for 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 an error-prone task which also requires substantial amount of resources. Codasip targets this with its EDA tool Studio. A RISC-V processor and any added custom instructions are described in a high-level architecture description language, CodAL. Studio uses the description to generate an SDK and an implementation of the processor. The high level of automation allows adding new instructions within hours, days, or weeks at most.

Codasip GmbH – Headquarters
Paul-Gerhardt-Allee 50
81245 Munich, Germany

Codasip s.r.o. – R&D Center
Božetěchova 1/2
612 00 Brno, Czech Republic

Phone: +420 541 141 475
E-mail: contact@codasip.com
Website: www.codasip.com