Enhanced LLVM Support for RISC-V

Zdeněk Přikryl

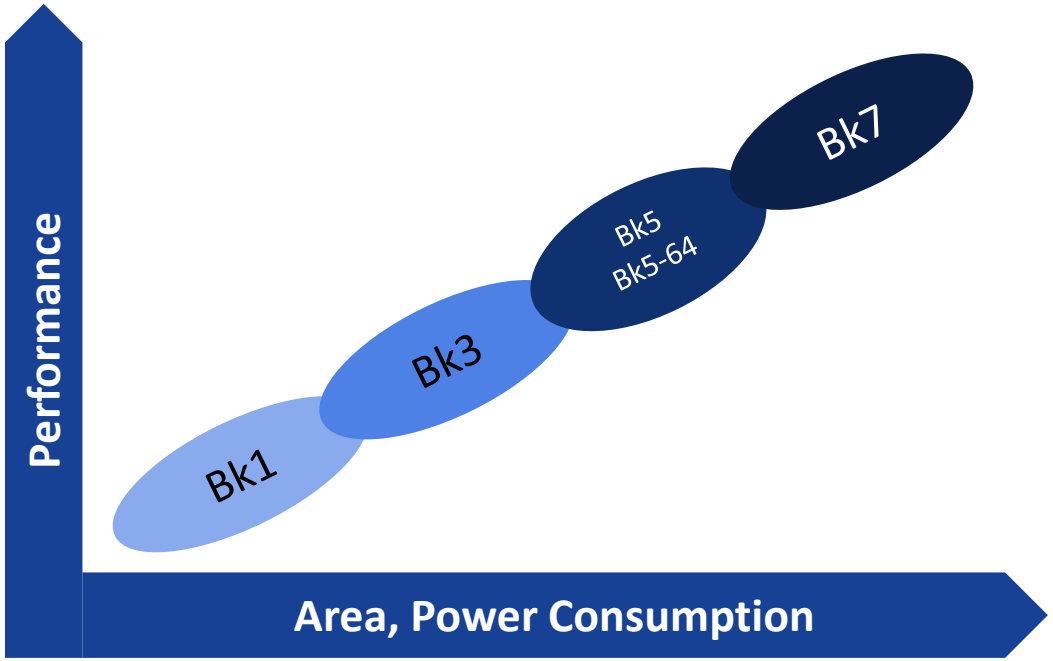
Agenda

- Who is Codasip
- Why we need customization-aware SDK?
- LLVM C/C++ compiler
- Profiler
- LLDB
- Conclusion

Who is Codasip

- Leading provider of RISC-V processor IP and design tools
  - Based on 10 years of university research in processor design automation
  - Introduced its first RISC-V processor in November 2015
- Company founded in Brno, Czech Republic
  - Offices in Brno and Silicon Valley
  - Sales representation worldwide
- Founding member of RISC-V Foundation, www.riscv.org
  - Member of several working groups within the Foundation
- Active contributor to LLVM and other open-source projects

Our Products: The Bk Series



Bk series = Codasip's own portfolio of RISC-V processors

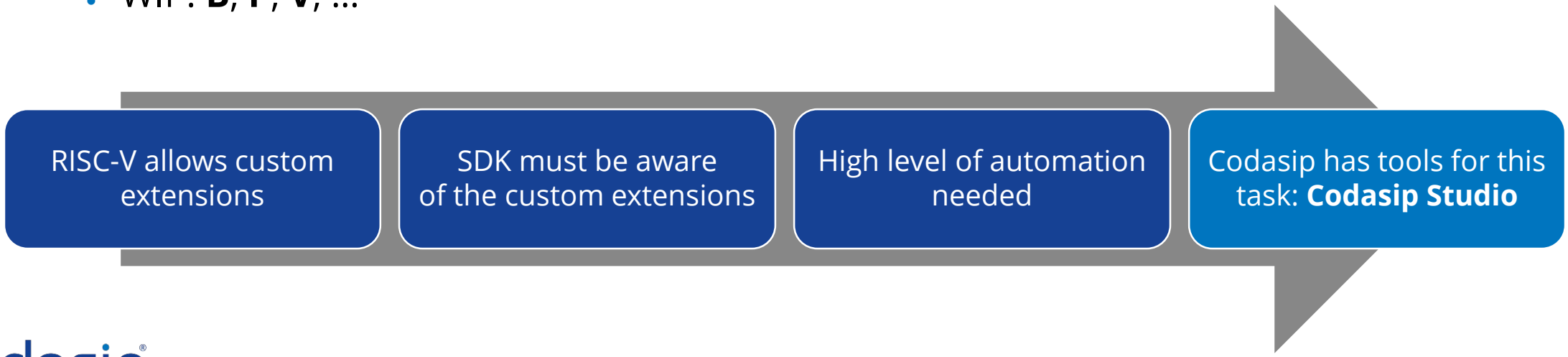
- ✓ Wide selection for any application needs
- ✓ Fully compliant with the RISC-V ISA
- ✓ Fully configurable
- ✓ Easily extensible with custom instruction sets

Configuration and Custom Extensions

RISC-V offers a wide range of ISA modules:

- **I/E** for integer instructions
- **M** for multiplication and division
- **C** for compact instruction
- **F/D** for floating point operations
- WIP: **B**, **P**, **V**, ...

However, it may **not be enough** for your application domain or if you are looking for a key **differentiator...**



Why Customization-Aware SDK?

One of the biggest advantages of the RISC-V open ISA is customization.
However, a customized processor also needs a customized SDK…

Standard customization (manually adding custom ISA extensions):

1. Model and simulate a new instruction
2. Modify the compiler
3. Modify assembler
4. Add support in the debugger
5. Verify, verify, verify…

→ Challenging, time-consuming, expensive

Benefits of automatic generation of a customized LLVM compiler:

- ✓ Reduced time needed for compiler modification
- ✓ Reduced cost of custom processor development
- ✓ The resultant processor will be easily programmable using standard C/C++
- ✓ Proven open-source LLVM framework allows for easy integration

Our Tools: Codasip Studio

Studio = unique collection of tools for fast & easy modification of RISC-V processors. All-in-one, highly automated. Introduced in 2014, silicon-proven by major vendors.

Customization of base instruction set:

- Single cycle MAC
- Custom crypto functions
- And many more…

Complete IP package on output:

- C/C++ LLVM-based compiler
- C/C++ Libraries
- Assembler, disassembler, linker
- ISS (incl. cycle accurate), debugger, profiler
- UVM SystemVerilog testbench

Codasip Studio

RTL Automation

Verilog **VHDL**

SDK automation

LLVM COMPILER INFRASTRUCTURE

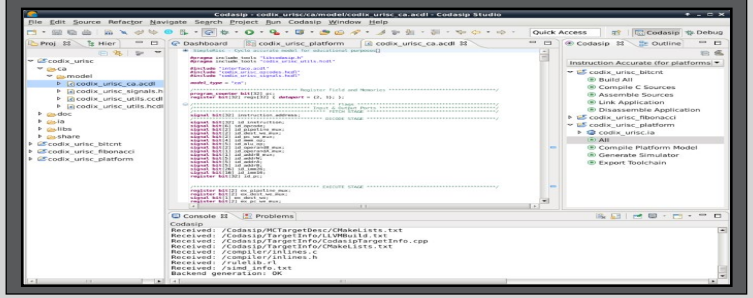
Verification Automation

UVM SYSTEMC

CodAL – processor description language

```
element i_mac {
   use reg as dst, src1, src2;
   assembly { "mac" dst "," src1 "," src2 };
   binary { OP_MAC dst src1 src2 0:bit[9] };
      semantics {
         rf[dst] += rf[src1] * rf[src2];
      };
};
```

Integrated processor development environment



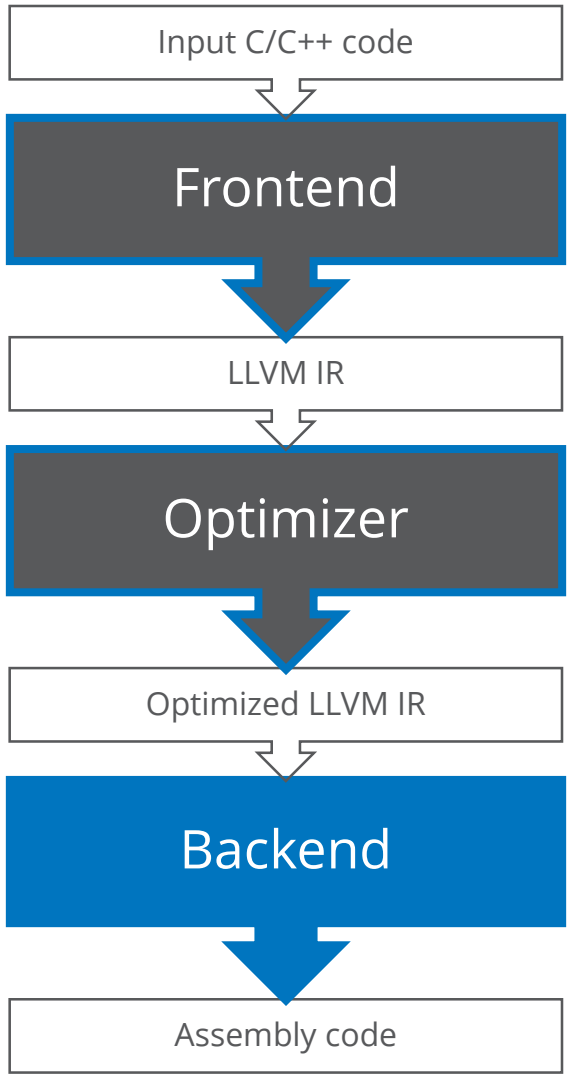
What is LLVM

- Open-source collection of modular and reusable compiler and toolchain technologies, llvm.org
- Supports a wide range of architectures
- Easily modified and extendable
- Three-stage design
  - `clang` = LLVM C/C++ compiler
    - fast compiles
    - useful error and warning messages
  - `opt` = LLVM optimizer
  - `llc` = LLVM backend

Codasip integrates **LLVM 7.0.1** in its Studio

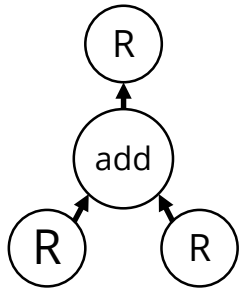
LLVM Three-Stage Design

- **Frontend (clang)**
  - Parses input code and produces LLVM Intermediate Representation (IR)
  - Added improvements by Codasip
- **Optimizer (opt)**
  - Performs mostly target-independent optimizations of LLVM IR
  - Added improvements by Codasip
- **Backend (llc)**
  - The actual code generator that transforms optimized LLVM IR into target-dependent assembly
  - Added improvements by Codasip
  - Generated by Codasip Studio



Compiler Generation: Needed Input

1. Information about registers (for register allocator)
   - General purpose registers, dedicated registers, flags, …
2. Instruction selection patterns (for instruction selector and the following passes)
   - DAG representation of what each instruction does, assembly form, …
3. Scheduling information
   - Memory latencies, delay slots, …
4. ABI definition
   - Callee/caller saved registers, stack pointer, …



Instruction selection pattern

Compiler Generation: Gathering Input

- Everything captured in a CodAL model
- **CodAL** = easy-to-understand C-like language that models a rich set of processor capabilities
- All Codasip processors are modelled and verified using CodAL models
- CodAL models are provided to Codasip IP customers as a starting point for their own processor optimizations and modifications

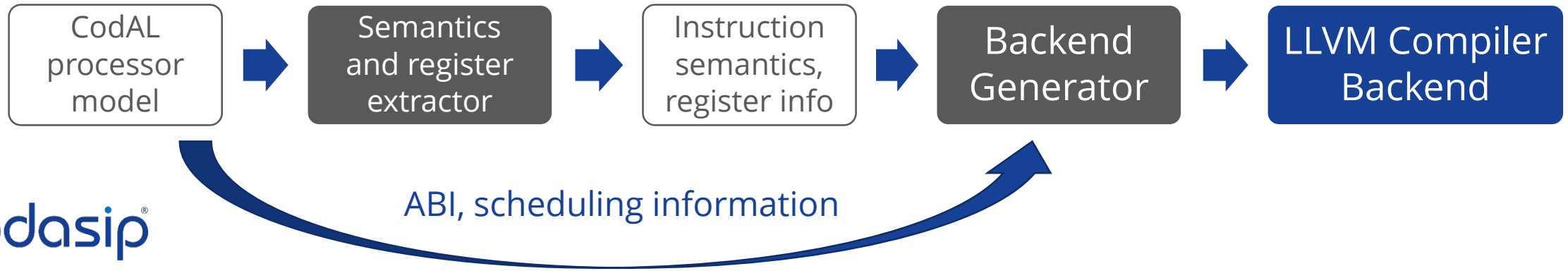
```
// register file
arch register_file bit[32] rf_regs {
      size = 32;
};

// ABI definition
compiler {
    // stack pointer information
    stack_pinter = rf_gpr[STACK_POINTER];
   ...
};

// Instruction: semantics is dst += src1 * src2
element i_mac {
      use reg as dst, src1, src2;
      assembly { “mac” dst “,” src1 “,” src2 };
      binary { OP_MAC dst src1 src2 0:bit[9] };
      semantics {
            rf[dst] += rf[src1] * rf[src2];
      };
};
```

Compiler Generation: 2-step Flow

1. Semantics and register extraction
   - Generates a semantics file that contains:
     - List of instructions (add, sub, …) with a precise specification of what each instruction does
     - Information on registers and register operands
2. Compiler generation
   - Uses the semantics produced by the extractor, ABI, and scheduling information to generate the Codasip LLVM compiler backend



Compiler Generation: Example (1/3)

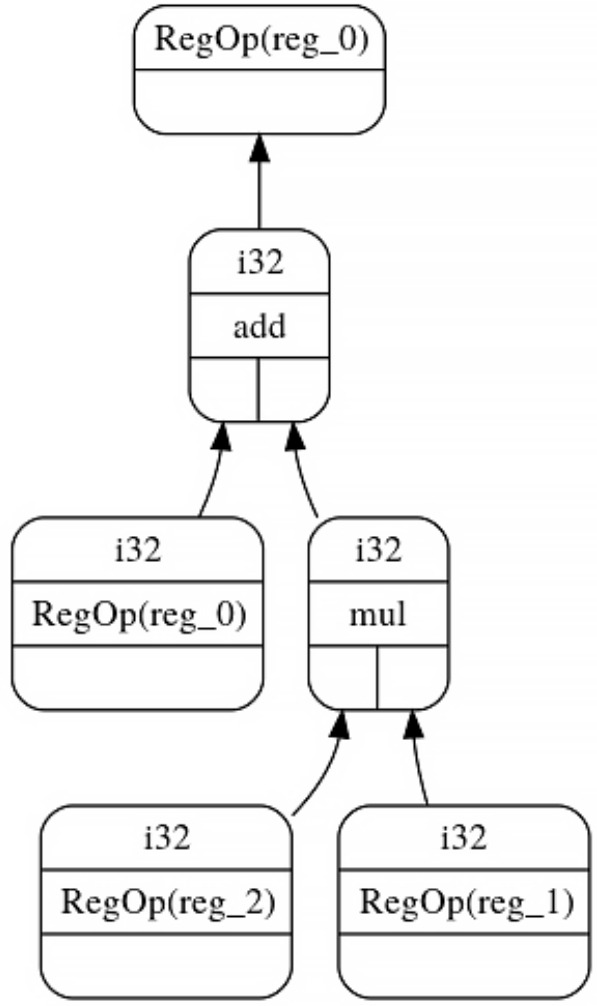
Adding a new custom instruction for a simple multiply and accumulate (e. g. mac t0, x10, x11):

```
// Instruction: semantics is dst += src1 * src2
element i_mac {
    use reg_all as dst, src1, src2;
    assembly { “mac” dst “,” src1 “,” src2 };
    binary { OP_MAC:bit[FRAG2] src2 src1 OPC_MAC:bit[FRAG1] dst OPC_MAC:bit[FRAG0] };
    semantics {
        rf[dst] += rf[src1] * rf[src2];
    };
};
```

Compiler Generation: Example (2/3)

Output of the semantic extraction

```
instruction i_mac {
    operands { reg_0 = regop(gpr_all), reg_1 = regop(gpr_all), … };
    assembly { "mac“ reg_0~”,” reg_1~"," reg_2 };
    binary { 0b… reg_2[4..0] reg_1[4..0] 0b… reg_0[4..0] 0b…};
    semantics {
        %5 = i32 regop(reg_0);
        %7 = i32 regop(reg_2);
        %8 = i32 regop(reg_1);
        %6 = i32 mul(i32 %7, i32 %8);
        %4 = i32 add(i32 %5, i32 %6);
        regop(reg_0) = i32 %4;
    };
};
```



i_mac

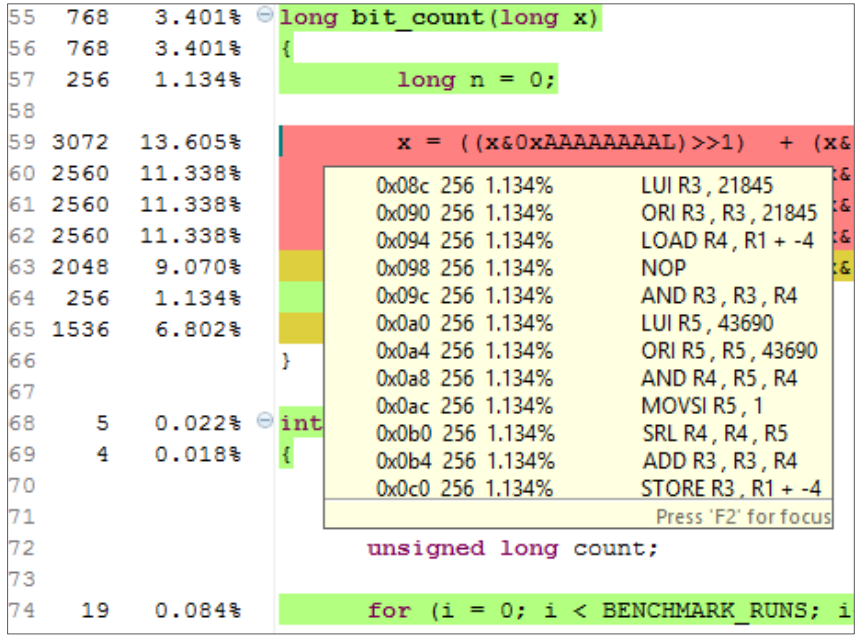
Compiler Generation: Example (3/3)

Output of the backend generator

```
def i_mac: CodasipMicroClass_<(outs gpr_all:$op0), (ins gpr_all:$op1, gpr_all:$op2,
gpr_all:$op3)>
{
  let AsmString = “mac $op0, $op3, $op2";
  let Pattern = [(set gpr_all:$op0, (i32 (add (i32 gpr_all:$op1),
                                          (i32 (mul (i32 gpr_all:$op2),
                                          (i32 gpr_all:$op3))))))];
  let Constraints = "$op1 = $op0";
  let Size = 4;
  let isReMaterializable = 1;
  let mayLoad = 0;
  let mayStore = 0;
}
```

Profiler

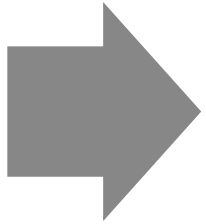
- Part of SDKs
- **Profiling** = dynamic form of analysis, identifies places in source code that should be optimized. Applicable to:
  - Applications running on an processor
  - The processor itself
- Two main modes:
  - Tracking/watching
  - Instrumentation/annotations
- Codasip IP packages support both modes
- Useful for instruction set extensions



Debugger

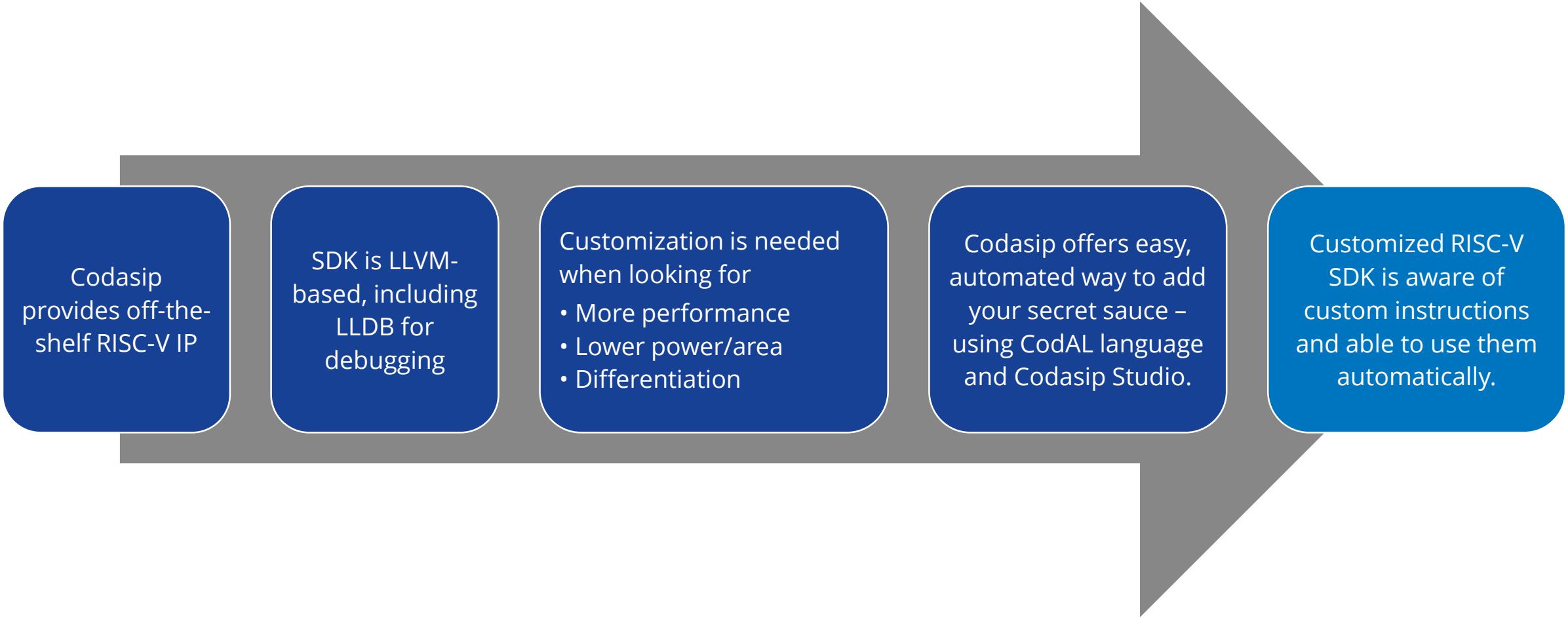
- Part of SDKs
- Should work with both ISS and on-chip debugger
- Must be aware of custom instructions as well
  - Disassembly view and instruction stepping
- **LLDB** = open-source debugger built on libraries provided by LLVM and Clang
  - Clean architecture with easy-to-extend plugin system
  - Its own commands – mapping to gdb commands exists

Codasip integrates **LLVM 7.0.1** in its Studio



We intend to present how we integrate LLDB at the next RISC-V workshop in Zurich

Conclusion


Codasip provides off-the-shelf RISC-V IP
SDK is LLVM-based, including LLDB for debugging
Customization is needed when looking for
• More performance
• Lower power/area
• Differentiation
Codasip offers easy, automated way to add your secret sauce – using CodAL language and Codasip Studio.
Customized RISC-V SDK is aware of custom instructions and able to use them automatically.

Thank you

Questions?

prikryl@codasip.com
www.codasip.com
Codasip GmbH