Code Density Improvements Beyond The C Standard Extension

Zdeněk Přikryl
CTO
Who is Codasip?

- The **leading provider** of RISC-V processor IP
  - Bk series of RISC-V-compliant processors
- Company founded in 2014 in the Czech Republic
- Founding member of the RISC-V Foundation, [www.riscv.org](http://www.riscv.org)
  - Member of several working groups in the Foundation
  - Actively contributing to LLVM and other open-source projects
- Now **Codasip GmbH**
  - Headquarters in Munich, Germany
  - R&D in Brno, Czech Republic
  - Offices in Silicon Valley, US, and Shanghai, Pudong PRC
What is Codasip Bk?

**Bk** = the Berkelium series, Codasip’s RISC-V processors

- Available immediately
- Pre-verified, tape-out quality IP
- Industry-standard interfaces
- Fully customizable

**Bk3** – entry-level 32bit RISC-V Core
- 3-stage single issue in-order pipeline
- Very small, efficient design

**Bk5** – 32 and 64bit RISC-V Cores with Balanced Pipeline
- 5-stage single issue in-order pipeline
- Optional dynamic branch prediction
- Optional cache memories

**Bk7** – Linux-ready 64bit RISC-V Core
- 7-stage single issue in-order pipeline with branch prediction
- MMU for Linux
- Cache memories (I$, D$)
What is Codasip Studio?

A 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

- Verification Automation

**CodAL** – processor description language

```plaintext
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**

**Codasip GmbH**
Bk Core Customization with Codasip Studio

Start from Bk3/5/7 cores
1. Add instructions
2. Add resources
3. Modify pipeline
4. ...

Your RISC-V CodAL Models

Codasip Studio Toolset

Your RISC-V HDK
Hardware Design Kit
- RTL models
- Synthesis scripts
- Verification models and simulators
- Virtual prototypes

Your RISC-V SDK
Software Design Kit
- Compiler
- Assembler
- Linker
- Debugger
- IDE etc

Profiling of embedded application SW enables processor optimizations

ISA extensions are quickly implemented and analyzed during design space exploration

Codasip Studio automatically generates all processor IP design kits and verifies for RISC-V compliance (you still need to verify your own resources and instructions).
Code Density

Small code footprint is essential in the embedded domain

Power/area-sensitive apps optimize to the last byte

RISC-V C extension:

- 16-bit compressed instructions for smaller code
- Some instructions work with only 8 registers (x8-x15)
- Mostly integer ops with a few floating point instructions
The C Extension

Extensive use of compressed instructions can result in 20–30% code density improvements.

Table 16.1: Compressed 16-bit RVC instruction formats.
# Improvements Beyond C Extension

## Goal
Find other areas to optimize in ISA for further code size improvement

## Test conditions

<table>
<thead>
<tr>
<th>Hardware:</th>
<th>Bk3 from Codasip</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RV32IMFC</td>
</tr>
<tr>
<td></td>
<td>3-stage single issue in-order pipeline, very small, efficient design</td>
</tr>
</tbody>
</table>

| Toolchain: | Codasip’s LLVM, (Flags -Oz -msave-restore) |

<table>
<thead>
<tr>
<th>Benchmark Code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreMark</td>
</tr>
<tr>
<td>Dhrystone</td>
</tr>
<tr>
<td>Whetstone</td>
</tr>
<tr>
<td>Embench</td>
</tr>
<tr>
<td>MiBench</td>
</tr>
<tr>
<td>FreeRTOS</td>
</tr>
<tr>
<td>Customer code</td>
</tr>
</tbody>
</table>
Baseline Results

~53% of 32-bit instructions are replaced by 16-bit
  • Leads to ~27% code reduction

The table shows the most used 16-bit instructions
  • Usage means how often the instruction is overall used in benchmarks/applications

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Usage [%]</th>
<th>Instructions</th>
<th>Usage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.mv</td>
<td>11.88</td>
<td>c.slli</td>
<td>0.85</td>
</tr>
<tr>
<td>c.lwsp</td>
<td>6.42</td>
<td>c.or</td>
<td>0.80</td>
</tr>
<tr>
<td>c.swsp</td>
<td>5.09</td>
<td>c.and</td>
<td>0.63</td>
</tr>
<tr>
<td>c.li</td>
<td>5.02</td>
<td>c.addi4spn</td>
<td>0.57</td>
</tr>
<tr>
<td>c.add</td>
<td>3.63</td>
<td>c.slli</td>
<td>0.36</td>
</tr>
<tr>
<td>c.addi</td>
<td>2.95</td>
<td>c.sub</td>
<td>0.31</td>
</tr>
<tr>
<td>c.j</td>
<td>2.62</td>
<td>c.jalr</td>
<td>0.23</td>
</tr>
<tr>
<td>c.lw</td>
<td>2.27</td>
<td>c.andi</td>
<td>1.19</td>
</tr>
<tr>
<td>c.beqz</td>
<td>1.91</td>
<td>c.xor</td>
<td>0.13</td>
</tr>
<tr>
<td>c.sw</td>
<td>1.77</td>
<td>c.srai</td>
<td>0.11</td>
</tr>
<tr>
<td>c.jr</td>
<td>1.70</td>
<td>c.flwsp</td>
<td>0.03</td>
</tr>
<tr>
<td>c.lui</td>
<td>1.23</td>
<td>c.fsw</td>
<td>0.02</td>
</tr>
<tr>
<td>c.bnez</td>
<td>1.23</td>
<td>c.fswsp</td>
<td>0.02</td>
</tr>
<tr>
<td>c.addi16sp</td>
<td>1.02</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Observations

## Accesses to memory
- Byte, half, word load/store
  - 8 register limit has more impact than the size of immediate offset
  - More registers lead into more compact instructions even if the offset is 0

## Branches
- $\equiv$, $\neq$, $>$ or $<$ operations
- Usage of $x0$
- $x13, x14, x15$ are used more often than the lower ones

## Data movement
- Have more bits for immediate in c.li or c.addi

## Floating point operations (SP/DP)
- Rarely used compared to other C instructions in our benchmark code
- Opcodes reused for new C instructions
Exploring Possible Improvements: Access to Memory

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Semantics (RVC → RVI)</th>
<th>Additional RVC [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.sb rd, rs</td>
<td>rd, rs – any of 32 registers (x0-x31)</td>
<td>lw rd, rs, 0</td>
<td>+ ~3 %</td>
</tr>
<tr>
<td>c.lbu rd, rs</td>
<td></td>
<td>lb rd, rs, 0</td>
<td></td>
</tr>
<tr>
<td>c.swx rd, rs</td>
<td></td>
<td>sw r0, rs, 0</td>
<td></td>
</tr>
<tr>
<td>c.lwx rd, rs</td>
<td></td>
<td>lw r0, rs, 0</td>
<td></td>
</tr>
</tbody>
</table>

**Additional improvements for half-words**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Semantics (RVC → RVI)</th>
<th>Additional RVC [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.sh rd, rs</td>
<td>rd, rs – any of 32 registers (x0-x31)</td>
<td>sh rd, rs, 0</td>
<td>+ ~1 %</td>
</tr>
<tr>
<td>c.lhu rd, rs</td>
<td></td>
<td>lh rd, rs, 0</td>
<td></td>
</tr>
</tbody>
</table>
### Exploring Possible Improvements: Branches

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Semantics (RVC → RVI)</th>
<th>Additional RVC [%]</th>
</tr>
</thead>
</table>
| c.bne rs1, rs2, imm | • rs1, rs2 – x0, x9,…,x15  
• imm – relative address shifted by one on 5 bits | beq rs1, rs2, imm     | + ~1 %            |
| c.bne rs1, rs2, imm |                                           | bne rs1, rs2, imm     |                   |
| c.bge rs1, rs2, imm  |                                           | bge rs1, rs2, imm     |                   |
| c.blt rs1, rs2, imm   |                                           | blt rs1, rs2, imm     |                   |
Exploring Possible Improvements: Data Movement

We changed the current `c.li` and `c.addi`:
- Bigger immediate
- Reduced number of usable registers

Compression ratio decrease:
- LLVM is more sensitive to registers than immediate (aligned with the access to the memory and branches)
Exploring Possible Improvements: ABI

- In some cases, an ABI change can improve by \(~5\%\)
  - Callee-saved registers
  - Function arguments and return values
- Highly code-dependent
  - In some cases, the gain is very nice
  - In other cases, it leads to bigger code
Possible Improvements All Together

- **Memory access**
  - Extra ~4% improvement over baseline

- **Branches**
  - 3 instructions → extra ~1%

- **ABI enhancements**
  - Extra ~5%

~~10% improvement is significant~ for many applications
  - HW cost in low, mainly decoder is influenced

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Usage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access to Memory</td>
<td></td>
</tr>
<tr>
<td>c.lbu</td>
<td>0.87</td>
</tr>
<tr>
<td>c.sb</td>
<td>0.39</td>
</tr>
<tr>
<td>c.lwx</td>
<td>1.06</td>
</tr>
<tr>
<td>c.swx</td>
<td>0.65</td>
</tr>
<tr>
<td>c.lhu</td>
<td>0.59</td>
</tr>
<tr>
<td>c.sh</td>
<td>0.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Branches</th>
<th>Usage [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.beq</td>
<td>0.31</td>
</tr>
<tr>
<td>c.bne</td>
<td>0.29</td>
</tr>
<tr>
<td>c.bge</td>
<td>0.19</td>
</tr>
<tr>
<td>c.blt</td>
<td>0.20</td>
</tr>
</tbody>
</table>
Conclusion

- The **current C extension** is beneficial, but it can be still **significantly improved**
  1. By adding new instructions
     - On average, ratio increased from **53% up to 63%**
  2. By adding custom fused instructions for further improvement
     - Load with post increment + add + mull, etc.

- **Codasip Studio** was used for the design space exploration
  - Started with baseline Codasip RISC-V implementation and toolchain
  - Several iterations per day → fast design space exploration
Thank you!

Questions?

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