Working with RISC-V

Part 1 of 4: Introduction to RISC-V ISA

Frank K. Gürkaynak  kgf@ee.ethz.ch
Luca Benini  lbenini@iis.ee.ethz.ch
Summary

- Part 1 – Introduction to RISC-V ISA
  - What is RISC-V about
  - Description of ISA, and basic principles
  - Simple 32b implementation (Ibex by LowRISC)
  - How to extend the ISA (CV32E40P by OpenHW group)

- Part 2 – Advanced RISC-V Architectures

- Part 3 – PULP concepts

- Part 4 – PULP based chips
Few words about myself

Working with RISC-V
RISC-V Instruction Set Architecture

- Started by UC-Berkeley in 2010
- Contract between SW and HW
  - Partitioned into user and privileged spec
  - External Debug
- Standard governed by RISC-V foundation
  - ETHZ is a founding member of the foundation
  - Necessary for the continuity
- Defines 32, 64 and 128 bit ISA
  - No implementation, just the ISA
  - Different implementations (both open and close source)
- At ETH Zurich we specialize in efficient implementations of RISC-V cores
RISC-V maintains basically a PDF document

Please note, RISC-V ISA and related specifications are developed, ratified and maintained by RISC-V International contributing members within the RISC-V International Technical Committee. Operating details of the Technical Committee can be found in the RISC-V International Tech Group. Work on the specification is performed on GitHub and the GitHub issue mechanism can be used to provide input into the specification.

ISA Specification

The specifications shown below represent the current, ratified releases:

- Volume 1, Unprivileged Spec v. 20191213 [PDF] [GitHub (latest)]
- Volume 2, Privileged Spec v. 20190608 [PDF] [GitHub (latest)]

Debug Specification

- External Debug Support v. 0.13.2 [PDF]
ISA defines the instructions that processor uses

C++ program translated to RISC-V instructions defined by ISA.

This will run on ANY RISC-V implementation.
RISC-V Ecosystem

- Binutils – upstream
- GCC – upstream
- LLVM – upstream
- Simulator:
  - "Spike" - reference
  - QEMU, Gem5
- OpenOCD
- OS
  - Linux, sel4, freeRTOS, zephyr
- Runtimes
  - Jikes, Ocaml, Go
- SW maintained by different parties
  - Binutils and GCC by Sifive a Berkeley start-up

See https://github.com/riscv/riscv-wiki/wiki/RISC-V-Software-Status for an updated list
RISC-V ISA is divided into extensions

- Kept very simple and extendable
  - Wide range of applications from IoT to HPC

- RV + word-width + extensions
  - RV32IMC: 32bit, integer, multiplication, compressed

- User specification:
  - Separated into extensions, only I is mandatory

- Privileged Specification (WIP):
  - Governs OS functionality: Exceptions, Interrupts
  - Virtual Addressing
  - Privilege Levels

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Integer instructions (frozen)</td>
</tr>
<tr>
<td>E</td>
<td>Reduced number of registers</td>
</tr>
<tr>
<td>M</td>
<td>Multiplication and Division (frozen)</td>
</tr>
<tr>
<td>A</td>
<td>Atomic instructions (frozen)</td>
</tr>
<tr>
<td>F</td>
<td>Single-Precision Floating-Point (frozen)</td>
</tr>
<tr>
<td>D</td>
<td>Double-Precision Floating-Point (frozen)</td>
</tr>
<tr>
<td>C</td>
<td>Compressed Instructions (frozen)</td>
</tr>
<tr>
<td>X</td>
<td>Non Standard Extensions</td>
</tr>
</tbody>
</table>
Work continues on new RISC-V extensions

- Foundation members work in **task-groups**
- Dedicated task-groups
  - Formal specification
  - Memory Model
  - Marketing
  - External Debug Specification
- ETH Zurich also contributes
  - Bit manipulation
  - Packed SIMD

| Q | Quad-precision Floating-Point |
| L | Decimal Floating Point |
| B | Bit Manipulation |
| T | Transactional Memory |
| P | Packed SIMD |
| J | Dynamically Translated Languages |
| V | Vector Operations |
| N | User-Level Interrupts |
What is so special about RISC-V

RISC-V base ISAs have either little-endian or big-endian memory systems, with the privileged architecture further defining bi-endian operation. Instructions are stored in memory as a sequence of 16-bit little-endian parcels, regardless of memory system endianness. Parcels forming one instruction are stored at increasing halfword addresses, with the lowest-addressed parcel holding the lowest-numbered bits in the instruction specification.

- Major design decisions have been properly motivated and explained
- Reserved space for extensions, modular
- Open standard, you can help decide how it is developed
The FREEDOM in RISC-V is implementation

- You can access all ISAs without (many) restrictions
  - SW tools need to be developed so that they can generate code for that ISA
- Most ISAs are closed. Only specific vendors can implement it
  - To use a core that implements an ISA, you have to license/buy it from vendor
  - Open source SW (for the ISA) is possible but building HW is not allowed
Are RISC-V processors better than XYZ?

- **Actual performance depends on the implementation**
  - RISC-V does not specify implementation details (on purpose)

- **Modern design, should deliver comparable performance**
  - If implemented well, it should perform as good as other modern ISA implementations
  - In our experiments, we see no weaknesses when compared to other ISAs
  - It also is not magically 2x better

- **High-end processor performance is not so much about ISA**
  - Implementation details like technology capabilities, memory hierarchy, pipelining, and power management are more important.
What is not so good about RISC-V?

- Still in development
  - Some standards (privilege, vector, debug etc.) still being refined, adjusted.
  - Tools and development environment needs to catch up.

- No canonical implementation (the RISC-V core)
  - It is free to implement, so many people did so, resulting in many cores

- Higher end (out of order, superscalar) cores not yet mature
  - In theory there is nothing to prevent a RISC-V based Linux laptop.
  - It will take some more time until RISC-V implementations can compete with other commercial processors (which needed hundreds of man months of work).
### Reduced Instruction Set: all in one page

#### Privilege Mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Regular user access</td>
</tr>
<tr>
<td>Supervisor</td>
<td>Enhanced access</td>
</tr>
<tr>
<td>Controller</td>
<td>Special access</td>
</tr>
</tbody>
</table>

#### Compressed Instructions (C)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Word</td>
<td>Load memory word</td>
</tr>
<tr>
<td>Load Byte</td>
<td>Load memory byte</td>
</tr>
<tr>
<td>Store Word</td>
<td>Store memory word</td>
</tr>
<tr>
<td>Store Byte</td>
<td>Store memory byte</td>
</tr>
<tr>
<td>Add</td>
<td>Arithmetic addition</td>
</tr>
<tr>
<td>Sub</td>
<td>Arithmetic subtraction</td>
</tr>
<tr>
<td>And</td>
<td>Logical AND</td>
</tr>
<tr>
<td>Or</td>
<td>Logical OR</td>
</tr>
<tr>
<td>Not</td>
<td>Logical NOT</td>
</tr>
</tbody>
</table>

#### Floating Point Extensions

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Float addition</td>
</tr>
<tr>
<td>Advanced</td>
<td>Float division</td>
</tr>
<tr>
<td>Special</td>
<td>Float comparison</td>
</tr>
</tbody>
</table>

#### Atomic Extensions (A)

<table>
<thead>
<tr>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load Upper Word</td>
<td>Load upper memory word</td>
</tr>
<tr>
<td>Store Upper Word</td>
<td>Store upper memory word</td>
</tr>
<tr>
<td>Load Upper Byte</td>
<td>Load upper memory byte</td>
</tr>
<tr>
<td>Store Upper Byte</td>
<td>Store upper memory byte</td>
</tr>
</tbody>
</table>

#### Multiply/Divide (M)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>Multiply two numbers</td>
</tr>
<tr>
<td>Divide</td>
<td>Divide two numbers</td>
</tr>
<tr>
<td>Square Root</td>
<td>Calculate square root</td>
</tr>
</tbody>
</table>

---

**ACACES 2020 - July 2020**

**ETH Zürich**

**Working with RISC-V**
RISC-V Architectural State

- There are 32 registers, each 32 / 64 / 128 bits long
  - Named x0 to x31
  - x0 is hard wired to zero
  - There is a standard ‘E’ extension that uses only 16 registers (RV32E)

- In addition one program counter (PC)
  - Byte based addressing, program counter increments by 4/8/16

- For floating point operation 32 additional FP registers

- Additional Control Status Registers (CSRs)
  - Encoding for up to 4’096 registers are reserved. Not all are used.
RISC-V Instructions four basic types

- **R** register to register operations
- **I** operations with immediate/constant values
- **S / SB** operations with two source registers
- **U / UJ** operations with large immediate/constant value

<table>
<thead>
<tr>
<th>31</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>18</th>
<th>17</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rd</td>
<td>opcode</td>
<td>U-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ETH Zürich

ACACES 2020 - July 2020
Encoding of the instructions, main groups

- **Reserved** opcodes for standard extensions
- Rest of opcodes free for *custom* implementations
- Standard extensions will be frozen/not change in the future

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00 LOAD</td>
<td>LOAD-FP</td>
<td>custom-0</td>
<td>MISC-MEM</td>
<td>OP-IMM</td>
<td>AUIPC</td>
<td>OP-IMM-32</td>
<td>48b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>01 STORE</td>
<td>STORE-FP</td>
<td>custom-1</td>
<td>AMO</td>
<td>OP</td>
<td>LUI</td>
<td>OP-32</td>
<td>64b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 MADD</td>
<td>MSUB</td>
<td>NMSUB</td>
<td>NMADD</td>
<td>OP-FP</td>
<td>reserved</td>
<td>custom-2/ru128</td>
<td>48b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 BRANCH</td>
<td>JALR</td>
<td>reserved</td>
<td>JAL</td>
<td>SYSTEM</td>
<td>reserved</td>
<td>custom-3/ru128</td>
<td>≥ 80b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
RISC-V is a load/store architecture

- All operations are on internal registers
  - Can not manipulate data in memory directly
- Load instructions to copy from memory to registers
- R-type or I-type instructions to operate on them
- Store instructions to copy from registers back to memory
- Branch and Jump instructions
Constants (Immediates) in Instructions

- In 32bit instructions, not possible to have 32b constants
  - Constants are distributed in instructions, and then sign extended
  - The Load Upper Immediate (lui) instruction to assemble/push constants

- Instruction types according to immediate encoding

<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>30</th>
<th>25</th>
<th>24</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
<td>opcode</td>
<td>I-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>imm[31:12]</td>
<td></td>
<td>rd</td>
<td>opcode</td>
<td>U-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Load from memory (ld), how immediates work

لد x9, 64(x22)

- Not possible to fit a 32b address in 32b encoding directly
  - Take the content in source (rs1), add the immediate (imm) to it. This is the address
  - Read from this address in the memory and load into the destination (rd) register

- RISC-V tries to minimize number of instructions
  - The 1d instruction seems overly complicated, but you can use this for everything
Branching, how addresses come together

\[ \text{bne } x10, \ x11, \ 2000 \ // \text{if } x10 \neq x11, \text{ jump 2000 ahead} \]

- Similar problem, how to encode jump address in branches
  - Branch on Equal (beq) and Branch on Not Equal (bne)
  - They use B type operations, need two source registers

- Jumps are relative to Program Counter (PC)
  - The immediate (constant) shows how far we have to jump (PC-relative addressing)
  - Works addresses within \( \pm 4096 \). To branch further, we need several instructions.
### RISC-V Instruction Length is Encoded

- LSB of the instruction tells how long the instruction is.
- Supports instructions of 16, 32, 48, 64, 80, 96, … , 320 bit
  - Allows RISC-V to have Compressed instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bit Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxaa</code></td>
<td>16-bit (aa ≠ 11)</td>
<td></td>
</tr>
<tr>
<td><code>xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxbb11</code></td>
<td>32-bit (bbb ≠ 111)</td>
<td></td>
</tr>
<tr>
<td><code>··xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxx011111</code></td>
<td>48-bit</td>
<td></td>
</tr>
<tr>
<td><code>··xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxx01111111</code></td>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td><code>··xxxx xxxxxxxxxxxxxxxxxxxxxx xxxxxxnnn11111111</code></td>
<td>(80+16*nnn)-bit, nnnn≠1111</td>
<td></td>
</tr>
<tr>
<td><code>··xxxx xxxxxxxxxxxxxxxxxxxxxx xxxx11111111111</code></td>
<td>Reserved for ≥320-bits</td>
<td></td>
</tr>
</tbody>
</table>

Byte Address:  
- base+4  
- base+2  
- base
Compressed Instruction extension ‘C’

- Use 16-bit instructions for common operations
  - Code size reduction by 34 %
  - Compressed instructions increase fetch-bandwidth
  - Allow for macro-op fusion of common patterns

x86-64: 3.71 bytes / instruction  RV64IC: 3.00 bytes / instruction
So how to build RISC-V cores

- **RISC-V ISA tells you the architecture**
  - You know which instructions are supported
  - How they are encoded
  - What they are supposed to do

- **It does not tell you any implementation details**
  - Pipeline stages, memory hierarchy, computation units, in-order or out-of order
  - Everyone is free to figure out how to best implement these

- **Need to come up with a micro-architecture to implement it**
  - Determine which standard extensions are supported, how
  - Choose a micro-architecture that fits performance requirements
What are the Performance Metrics

- **Area**
  - in kGE equivalent (\# of simple logic gates) or \( \text{mm}^2 \) (technology dependent)

- **Frequency**
  - Depends on \# of gates on longest path

- **Power**
  - Strongly depends on the above metrics
  - **Leakage**: dissipated even when not working (Area)
  - **Dynamic Power**: dissipated on logic transitions (frequency and area)

- **CPU Design**
  - **IPC** (Instructions per cycle)
    - IPC implicitly measured in commonly used benchmarks (Coremark, Dhrystone, SpecInt)
  - **Energy Efficiency**: OPs/Joule

- **Hardware Designer**
  - Tries to find a good balance
  - Application dependent
    - IoT and HPC have different requirements
  - One size does not fit all
### RISC-V Cores Developed at ETH Zurich

<table>
<thead>
<tr>
<th>32 bit</th>
<th>64 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Low Cost Core</strong></td>
<td><strong>Linux capable Core</strong></td>
</tr>
<tr>
<td>▪ Zero-riscy</td>
<td>▪ Ariane</td>
</tr>
<tr>
<td>▪ RV32-ICM</td>
<td>▪ RV64-IC(MA)</td>
</tr>
<tr>
<td>▪ Micro-riscy</td>
<td>▪ Full privileged specification</td>
</tr>
<tr>
<td>▪ RV32-CE</td>
<td></td>
</tr>
<tr>
<td><strong>DSP Enhanced Core</strong></td>
<td></td>
</tr>
<tr>
<td>▪ RI5CY</td>
<td></td>
</tr>
<tr>
<td>▪ RV32-ICMFX</td>
<td></td>
</tr>
<tr>
<td>▪ SIMD</td>
<td></td>
</tr>
<tr>
<td>▪ HW loops</td>
<td></td>
</tr>
<tr>
<td>▪ Bit manipulation</td>
<td></td>
</tr>
<tr>
<td>▪ Fixed point</td>
<td></td>
</tr>
<tr>
<td><strong>Streaming Compute</strong></td>
<td></td>
</tr>
<tr>
<td>▪ Core Snitch</td>
<td></td>
</tr>
<tr>
<td>▪ RV32-ICMDFX</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- Ibex by LowRISC
- CV32E40P by OpenHW
- CV6A by OpenHW

These RISC-V cores are used in various applications, including Linux-capable cores, enhanced cores for DSP, and streaming compute cores. The low-cost cores are ideal for low-power applications, while the full-privileged 64-bit cores are suitable for high-performance computing tasks.
**Zero-riscy / Ibex, small core for control applications**

- **2-stage pipeline**
- **Optimized for area**
  - Area: 19 kGE (Zero-riscy)
  - 12 kGE (Micro-riscy)
  - Critical path: ~30 logic levels
- **New name: Ibex**
  - LowRISC has taken over Zero/Micro-Riscy in 2019

**Two Configurations:**

- **Zero-riscy**: RV32IMC (2,44 Coremark/MHz)
  - 32 registers, hardware multiplier
- **Micro-riscy**: RV32EC (0,91 Coremark/MHz)
  - 16 registers (E), software emulated multiplier
Ibex is a small and efficient, 32-bit, in-order RISC-V core with a 2-stage (or optionally 3-stage) pipeline that implements the RV32IMCB instruction set architecture.

Since being contributed to lowRISC by ETH Zürich, it has seen substantial investment of development effort.
Roadmap of Ibex

**lowRISC**

- Randomised execution time
- Non-data-dependent fixed execution time
- Parity checks
- Bus scrambling
- CFI (TBD)
- Shadow PMP regs
- OT secure coding guidelines conform

**Stabilisation 19Q3-19Q4**
- RISC-V specification conformance
- Code clean up and refactoring (~50% LoC changed)
- CI & DV (riscv-dv, Google)

**Perf phase 1 20Q1**
- Branch target ALU
- Third pipeline stage
- Single-cycle MUL
- I$ prototype

**Perf phase 2 20Q2**
- Finalise I$
- Static branch predictor
- Bitmanip ISA extension

**Security hardening phase 1 20Q2**

**Security hardening phase 2 20Q3**
Growth of Ibex measured with Coremark/MHz

Past Work
- Branch Target ALU: 2.43
- Third Pipeline Stage: 2.55

Today
- Single Cycle Multiply: 2.92
- Static Branch Prediction: 3.09

Future
- Bit Manipulation ISA Extension: 3.19
RI5CY / CV32E40P our main 32bit RISC-V core

- Zero-riscy / Ibex is suitable for simple applications
  - Control applications, book-keeping

- For our research we need more capable cores
  - Mainly used in clusters for signal processing / machine learning applications

- Tuned for energy efficiency
  - Not necessarily low power

- Make use of custom extensions
  - The Xpulp extensions enhance the capabilities
  - Several Xpulp extensions in discussions for ratification
Simplified pipeline for RI5CY / CV32E40P
RI5CY: Our 32-bit workhorse

- 4-stage pipeline
  - 41 kGE
  - Coremark/MHz 3.19
- Includes Xpulp extensions
  - SIMD
  - Fixed point
  - Bit manipulations
  - HW loops

Different Options:
- **FPU**: IEEE 754 single precision
  - Including hardware support for FDIV, FSQRT, FMAC, FMUL
- **Privilege support**:
  - Supports privilege mode M and U
RISC-V has space for custom instructions (X)

- There is a reserved decoding space for custom instructions
  - Allows everyone to add new instructions to the core
  - The address decoding space is reserved, it will not be used by future extensions
  - Implementations supporting custom instructions will be compatible with standard ISA
    - Code compiled for standard RISC-V will run without issues
  - The user has to provide support to take advantage of the additional instructions
    - Compiler that generates code for the custom instructions

- ETH Zurich regularly uses these instructions
  - Great tool for exploring
  - The goal is to help ratify these extensions as standards through working groups
Our extensions to RI5CY (with additions to GCC)

- Post-incrementing load/store instructions
- Hardware Loops (lp.start, lp.end, lp.count)
- ALU instructions
  - Bit manipulation (count, set, clear, leading bit detection)
  - Fused operations: (add/sub-shift)
  - Immediate branch instructions
- Multiply Accumulate (32x32 bit and 16x16 bit)
- SIMD instructions (2x16 bit or 4x8 bit) with scalar replication option
  - add, min/max, dotproduct, shuffle, pack (copy), vector comparison

For 8-bit values the following can be executed in a single cycle

\[ Z = D_1 \times K_1 + D_2 \times K_2 + D_3 \times K_3 + D_4 \times K_4 \]
RI5CY ISA extensions improve performance

```
for (i = 0; i < 100; i++)
    d[i] = a[i] + b[i];
```

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Auto-increment load/store</th>
<th>HW Loop</th>
<th>Packed-SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv x5, 0</td>
<td>mv x5, 0</td>
<td>lp.setupi 100, Lend</td>
<td>lp.setupi 25, Lend</td>
</tr>
<tr>
<td>mv x4, 100</td>
<td>mv x4, 100</td>
<td>lb x2, 0(x10!)</td>
<td>lw x2, 0(x10!)</td>
</tr>
<tr>
<td>Lstart:</td>
<td>Lstart:</td>
<td>lb x3, 0(x11!)</td>
<td>lb x3, 0(x11!)</td>
</tr>
<tr>
<td>lb x2, 0(x10)</td>
<td>lb x3, 0(x11)</td>
<td>add x2, x3, x2</td>
<td>add x2, x3, x2</td>
</tr>
<tr>
<td>lb x3, 0(x11)</td>
<td>addi x4, x4, -1</td>
<td>sb x2, 0(x12!)</td>
<td>pv.add.b x2, x3, x2</td>
</tr>
<tr>
<td>addi x10, x10, 1</td>
<td>add x2, x3, x2</td>
<td>bne x4, x5, Lstart</td>
<td>Lend: sb x2, 0(x12!)</td>
</tr>
<tr>
<td>addi x11, x11, 1</td>
<td></td>
<td></td>
<td>Lend: sw x2, 0(x12!)</td>
</tr>
<tr>
<td>add x2, x3, x2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb x2, 0(x12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sb x2, 0(x12)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne x4, x5, Lstart</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 cycles/output 8 cycles/output 5 cycles/output 1,25 cycles/output
Runtime for three different applications

2D Convolution

EEMBC Coremark

Scheduler Application

Extensions have more effect

Better

RV32IMCXpulp
RV32IMC
RV32EC
Different cores for different area budgets

---

Better RV32IMCXpulp

---

Area [KGE]

- RV32IMC
  - prefetcher-bufi
  - decoder+ctrl
  - register-file
  - load-status RF
  - load-store unit
  - ALU
  - multi/div unit
  - debug unit

- RV32EC
  - prefetcher-bufi
  - decoder+ctrl
  - register-file
  - load-status RF
  - load-store unit
  - ALU
  - multi/div unit
  - debug unit

- RV32IMCXpulp
  - prefetcher-bufi
  - decoder+ctrl
  - register-file
  - load-status RF
  - load-store unit
  - ALU
  - multi/div unit
  - debug unit

---

x2.2

---

x3.5
Different cores for different power budgets

- RV32IMCxPulp: Better
- RV32IMC: x2.4
- RV32EC: x2.7

Power Consumption [mW]
Energy Efficiency: 2D-Convolution @55MHz, 0.8V

More frequent events/processing


- RV32IMCXpulp: Fast-Events
- RV32IMC: good trade-off
- RV32EC: for Slow-Events

- 784 μs
- 4.78 ms
- 41.6 ms
- 649 ms
- 31 s
This was a short overview of basics of RISC-V

- After the break, more advanced cores
  - 64bit RISC-V core
  - Discussion on performance
  - Vector processing

- Tomorrow, we learn about PULP systems
  - Cores alone can not do much, they need a system around
  - Many core systems
  - Managing Data
  - Acceleration
  - Actual Integrated Circuits from the PULP group
Luca Benini, Davide Rossi, Andrea Borghesi, Michele Magno, Simone Benatti, Francesco Conti, Francesco Beneventi, Daniele Palossi, Giuseppe Tagliavini, Antonio Pullini, Germain Haugou, Manuele Rusci, Florian Glaser, Fabio Montagna, Bjoern Forsberg, Pasquale Davide Schiavone, Alfio Di Mauro, Victor Javier Kartsch Morinigo, Tommaso Polonelli, Fabian Schuiki, Stefan Mach, Andreas Kurth, Florian Zaruba, Manuel Eggimann, Philipp Mayer, Marco Guermandi, Xiaying Wang, Michael Hersche, Robert Balas, Antonio Mastrandrea, Matheus Cavalcante, Angelo Garofalo, Alessio Burrello, Gianna Paulin, Georg Rutishauser, Andrea Cossettini, Luca Bertaccini, Maxim Mattheeuws, Samuel Riedel, Sergei Vostrikov, Vlad Niculescu, Hanna Mueller, Matteo Perotti, Nils Wistoff, Luca Bertaccini, Thorir Ingulfsson, Thomas Benz, Paul Scheffler, Alessio Burello, Moritz Scherer, Matteo Spallanzani, Andrea Bartolini, Frank K. Gurkaynak, and many more that we forgot to mention

http://pulp-platform.org  @pulp_platform
The extensions translate to real speed-ups

- **8-bit convolution**
  - Open source DNN library
- **10x through xPULP**
  - Extensions bring real speedup
- **Near-linear speedup**
  - Scales well for regular workloads.
- **75x overall gain**

![Graph showing speedup](chart.png)

- Overall Speedup of **75x**
- Near-Linear Speedup
- 10x Speedup w.r.t. RV32IMC (ISA does matter 😊)