PULP PLATFORM
Open Source Hardware, the way it should be!

Working with RISC-V

from open ISA to open Architecture to open Hardware

Part 1 of 5 : Introduction to RISC-V ISA

Luca Benini <luca.benini@unibo.it>
Davide Rossi <davide.rossi@unibo.it>

ETH zürich

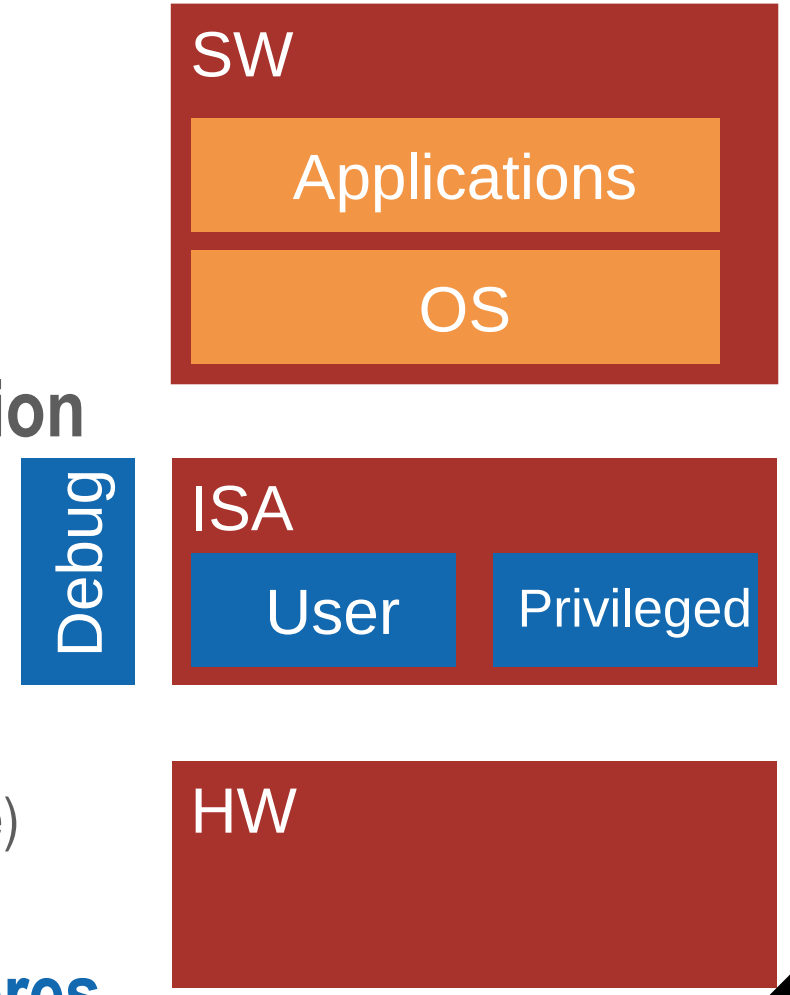
http://pulp-platform.org @pulp_platform https://www.youtube.com/pulp_platform

Summary

- **Part 1 – Introduction to RISC-V ISA**
  - What RISC-V is 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 Extensions and Accelerators**
- **Part 5 – PULP based chips**

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 ETHZ+UNIBO we specialize in **efficient implementations of RISC-V cores**



RISC-V maintains basically a PDF document

riscv.org/specifications/

Join the Mailing Lists | info@riscv.org | Member Login

RISC-V®

ABOUT | MEMBERSHIP | SPECS & SUPPORT | HARDWARE & SOFTWARE | NEWS | EVENTS

Specifications

/ Specifications

RISC-V SPECIFICATIONS

- Unprivileged Specification
- Privileged ISA Specification
- Debug Specification

RISC-V SOFTWARE

- Software Status

RISC-V CORES

- RISC-V Cores

RISC-V EDUCATION

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++ source #1

```
1  void gemm_bin(int M, int N, int K, float ALPHA,
2          char  *A, int lda,
3          float *B, int ldb,
4          float *C, int ldc)
5  {
6      int i,j,k;
7      for(i = 0; i < M; ++i){
8          for(k = 0; k < K; ++k){
9              char A_PART = A[i*lda+k];
10             if(A_PART){
11                 for(j = 0; j < N; ++j){
12                     C[i*ldc+j] += B[k*ldb+j];
13                 }
14             } else {
15                 for(j = 0; j < N; ++j){
16                     C[i*ldc+j] -= B[k*ldb+j];
17                 }
18             }
19         }
20     }
21 }
22
```

RISC-V rv64gc clang (trunk) (Editor #1, Compiler #1) C++

```
63  .LBB0_7:                                    #   in Loop: Header=BB0_6 D
64          ld      a0, -56(s0)
65          lw      a1, -76(s0)
66          lw      a2, -60(s0)
67          mul     a1, a1, a2
68          lw      a2, -72(s0)
69          addw    a1, a1, a2
70          slli    a1, a1, 2
71          add     a0, a0, a1
72          flw     ft0, 0(a0)
73          ld      a0, 0(s0)
74          lw      a1, -68(s0)
75          lw      a3, -64(s0)
76          mul     a1, a1, a3
77          addw    a1, a1, a2
78          slli    a1, a1, 2
79          add     a0, a0, a1
80          flw     ft1, 0(a0)
81          fadd.s  ft0, ft1, ft0
82          fsw     ft0, 0(a0)
```

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

This will run on ANY RISC-V implementation

COMPILER EXPLORER

Screen shot from the excellent Compiler Explorer by Matt Godbolt https://godbolt.org/

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

I	Integer instructions (frozen)
E	Reduced number of registers
M	Multiplication and Division (frozen)
A	Atomic instructions (frozen)
F	Single-Precision Floating-Point (frozen)
D	Double-Precision Floating-Point (frozen)
C	Compressed Instructions (frozen)
X	Non Standard 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

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, DSP

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.

We originally chose little-endian byte ordering for the RISC-V memory system because little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). A minor point is that we have also found little-endian memory systems to be more natural for hardware designers. However, certain application areas, such as IP networking,

- 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**

RISC-V

Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP

ARM

C2.9 ADD

Add without Carry.

Syntax

```
ADD{S}{cond} {Rd}, Rn, Operand2
ADD{cond} {Rd}, Rn, #imm12 ; T32, 32-bit encoding only
```

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 well as other modern ISA implementations
  - In our experiments, we see **no major 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 microarchitecture, memory hierarchy, target technology, 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)
  - Getting there (Alibaba XT910, SiFive P550, Esperanto ET-Maxion, Semidynamics Avispado, Rivos ??? and more coming every day!)

Reduced Instruction Set: all in one page

Free & Open RISC-V Reference Card

Basic Instructions (I)

Privilege Mode

Multiply/Divide (M)

Atomic Extensions (A)

Compressed Instructions (C)

Floating Point Extensions

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

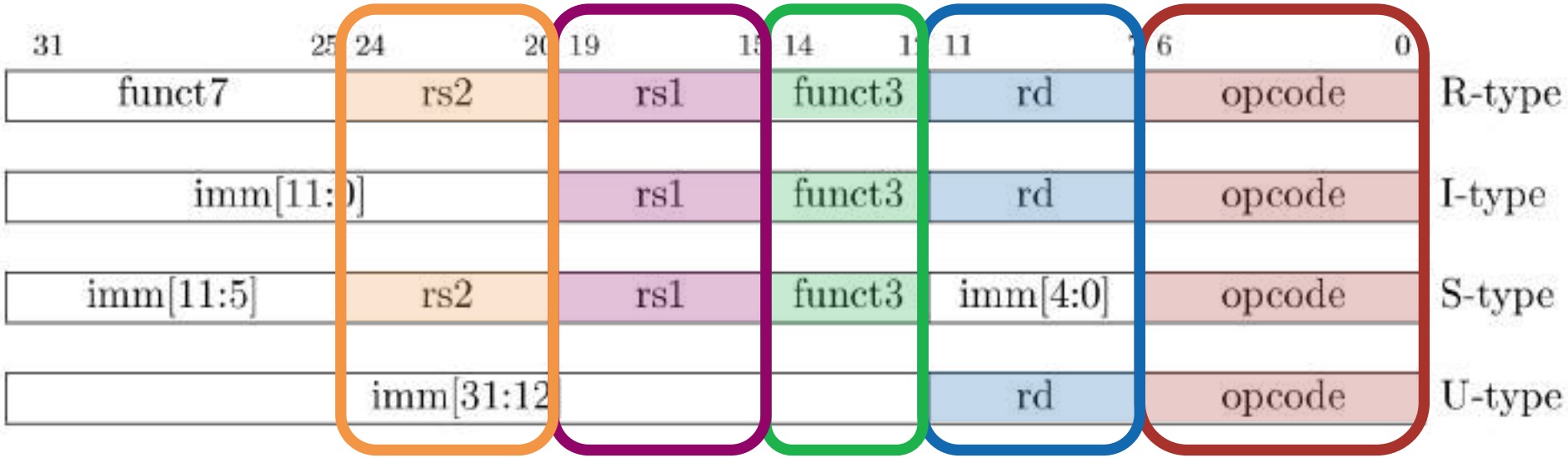
RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, {} means set, so L{D|Q} is both LD and LQ. See risc.org. (8/21/15 revision)

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



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

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

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

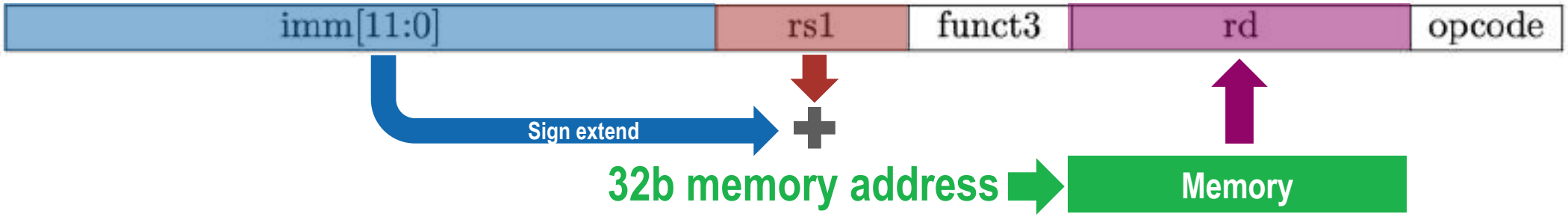
Constants (Immediates) in Instructions

- **In 32bit instructions, not possible to have 32b constants**
  - Constants are distributed in instructions, and then sign extended
  - The **L**oad **U**pper **I**mmediate (`lui`) instruction to assemble/push constants
- **Instruction types according to immediate encoding**

31 30 25	24 21 20	19 15	14 12	11 8	7	6 0	
funct7	rs2	rs1	funct3	rd		opcode	R-type
imm[11:0]		rs1	funct3	rd		opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]		opcode	S-type
imm[12] imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]				rd		opcode	U-type
imm[20] imm[10:1] imm[11]	imm[19:12]			rd		opcode	J-type

Load from memory (`ld`), how immediates work

```
ld 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 `ld` instruction seems overly complicated, but you can use this for everything

Branching, how addresses come together

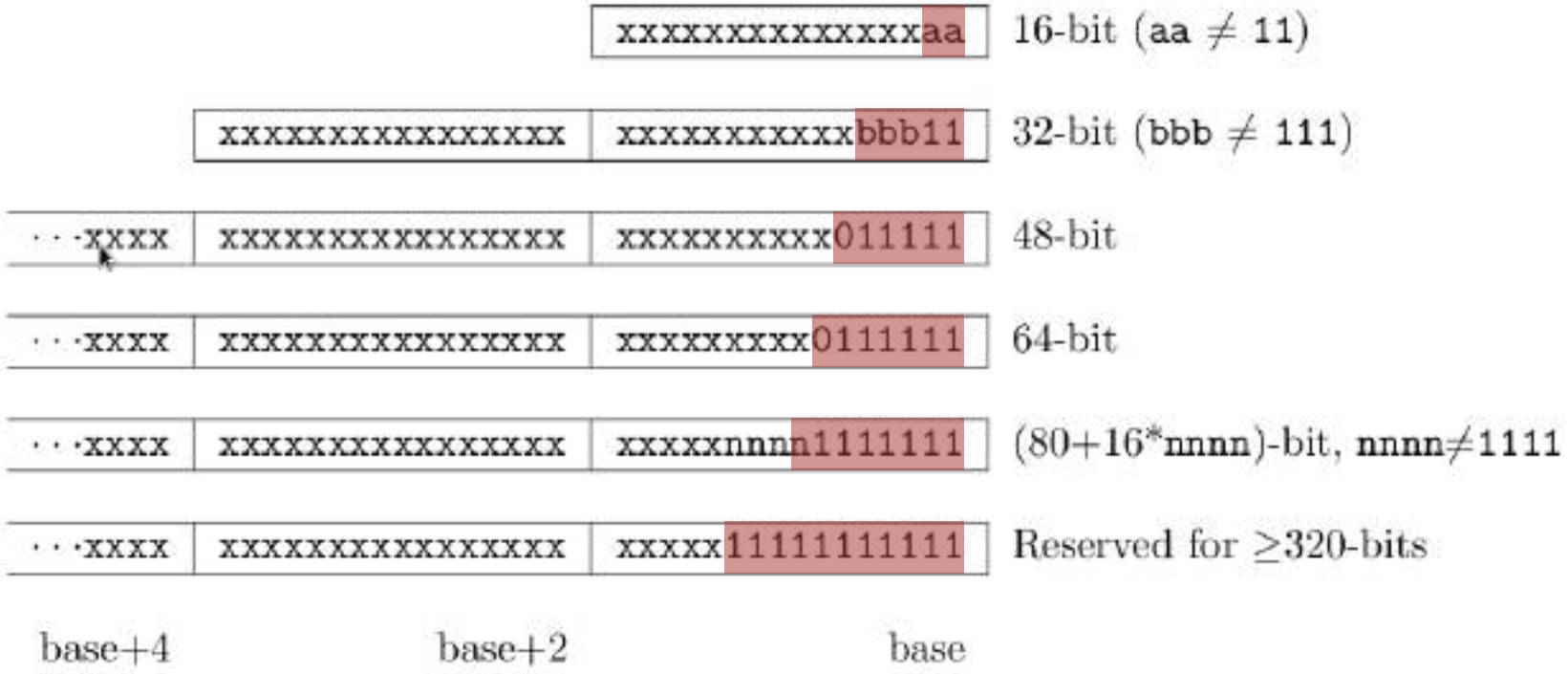
```
bne x10, x11, 2000 // if x10 != x11, jump 2000 ahead
```

imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

- **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 ± 4096. To branch further, we need several instructions.

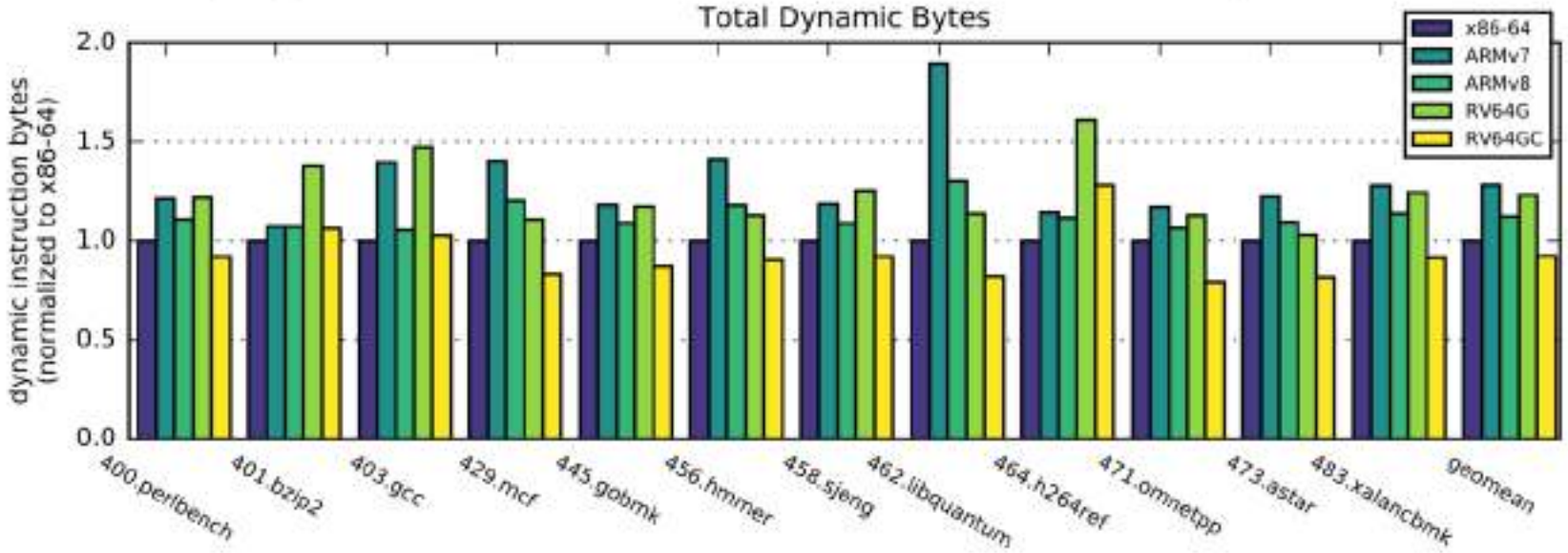
RISC-V Instruction Length is Encoded

- 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



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 function**
  - 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 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

32 bit			64 bit
Low Cost Core	**DSP Enhanced Core**	**Streaming Compute Core**	**Linux capable Core**
▪ **Zero-riscy** ▪ RV32-ICM ▪ **Micro-riscy** ▪ RV32-C	▪ **RI5CY** ▪ RV32-ICMDFX ▪ SIMD ▪ HW loops ▪ Bit manipulation ▪ Fixed point	▪ **Snitch** ▪ RV32-ICMDFX	▪ **Ariane** ▪ RV64-IC(MA) ▪ Full privileged specification

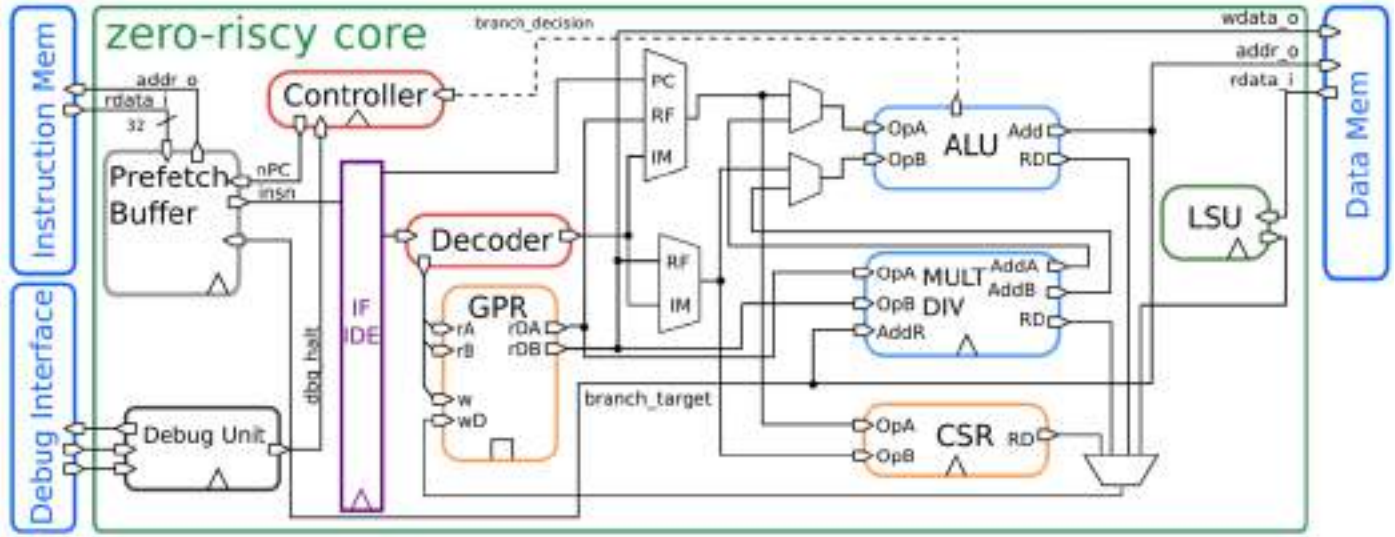
Ibex by LowRISC

CV32E40P by OpenHW

CV6A by OpenHW

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



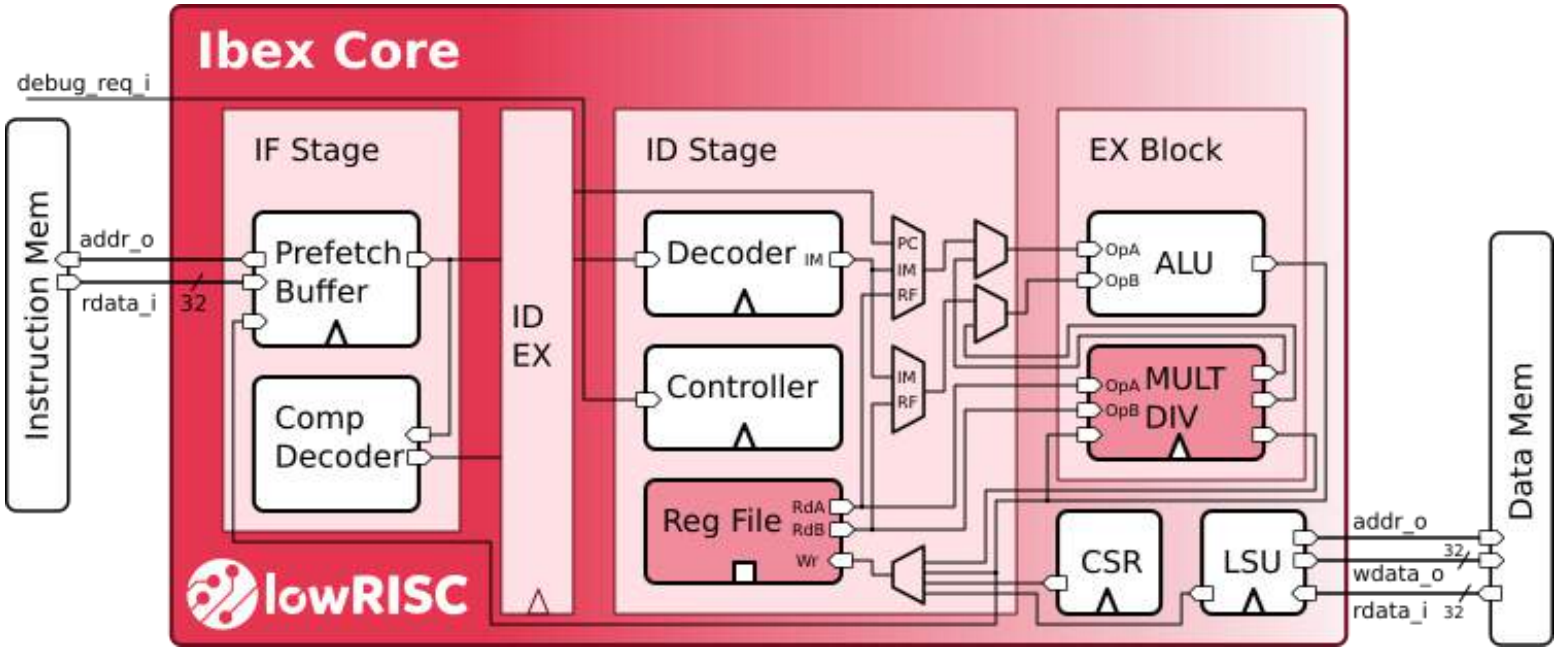
- **Two Configurations:**
  - **Zero-riscy**: RV32IMC (2,44 Coremark/MHz)
    - 32 registers, hardware multiplier
  - **Micro-riscy** : RV32EC (0,91 Coremark/MHz)
    - 16 registers (E), software emulated multiplier

P. Davide Schiavone *et al.*, "Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications," *2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017, pp. 1-8.

Ibex continues to grow with LowRISC

40+	Contributors
680	Pull Requests
314	GitHub Issues

lowRISC



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

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

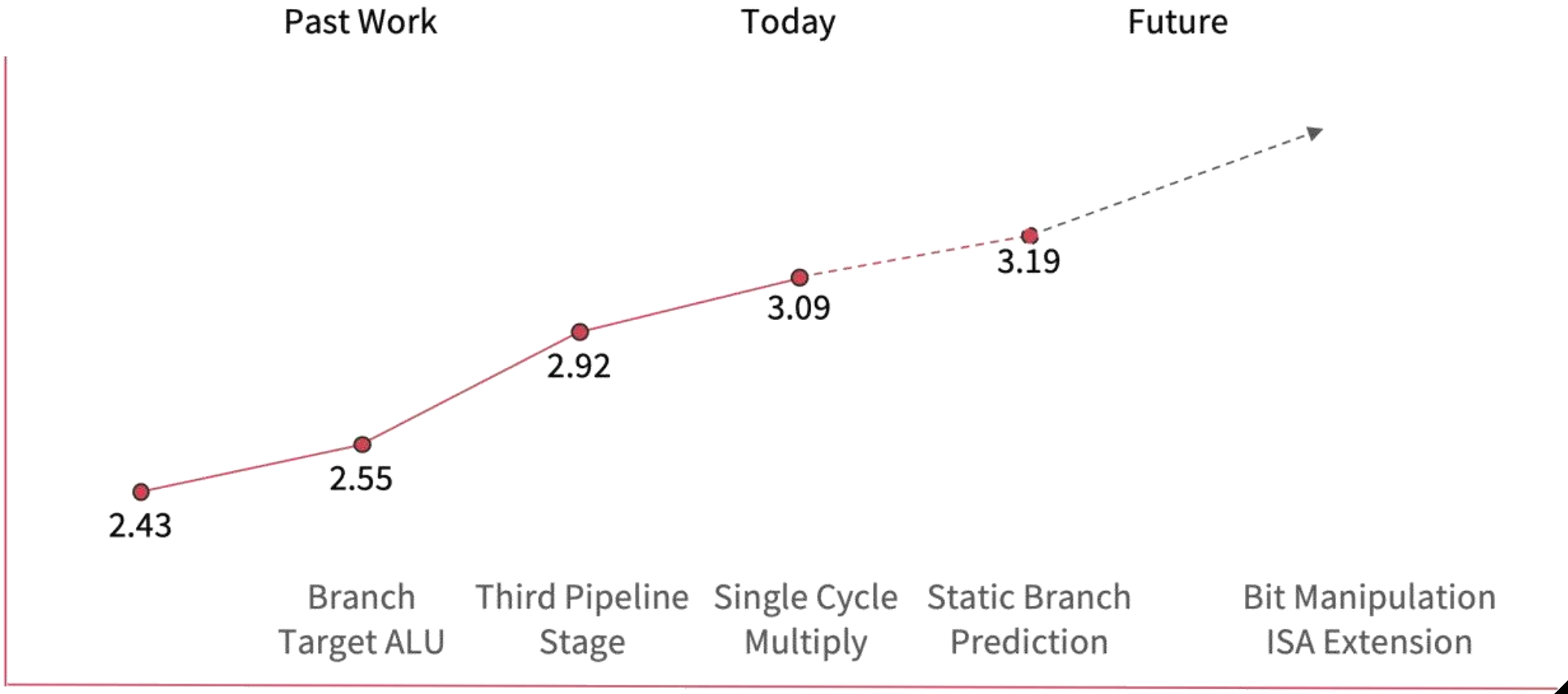
Security hardening phase 1 20Q2

- Randomised execution time
- Non-data-dependent fixed execution time
- Parity checks

Security hardening phase 2 20Q3

- Bus scrambling
- CFI (TBD)
- Shadow PMP regs
- OT secure coding guidelines conform

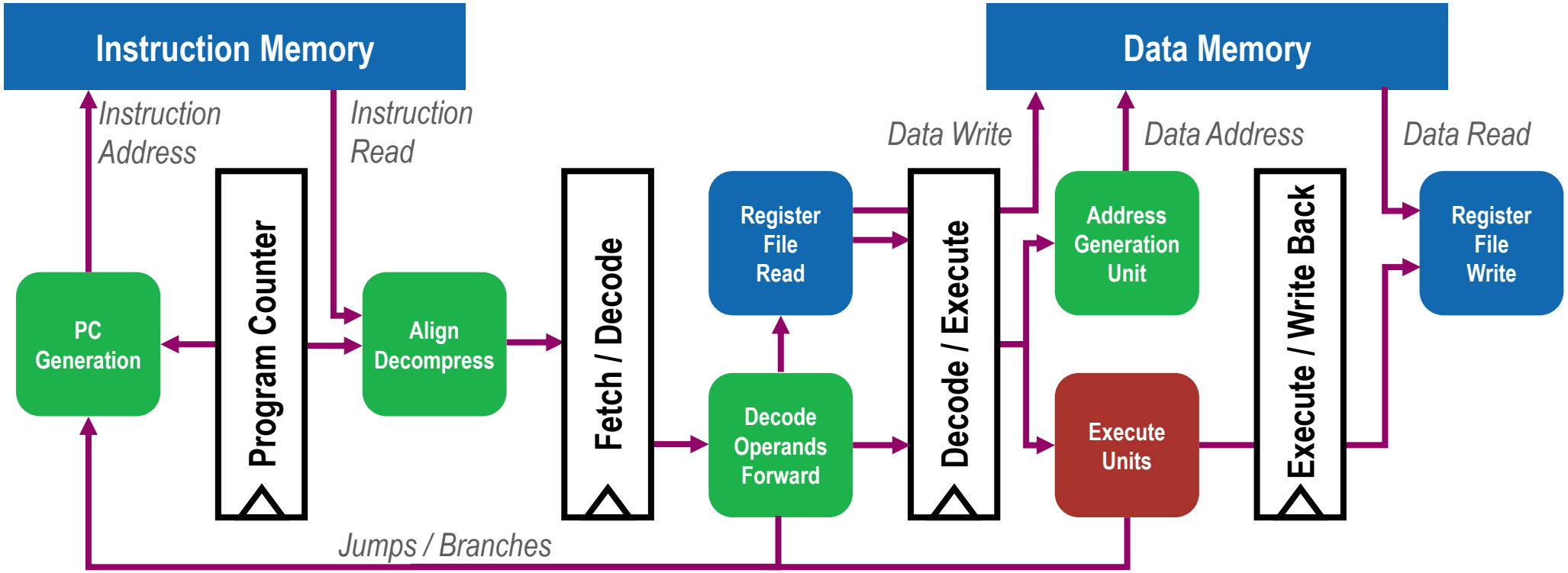
Growth of Ibex measured with Coremark/MHz



RI5CY / CV32E40P our main 32bit RISC-V core

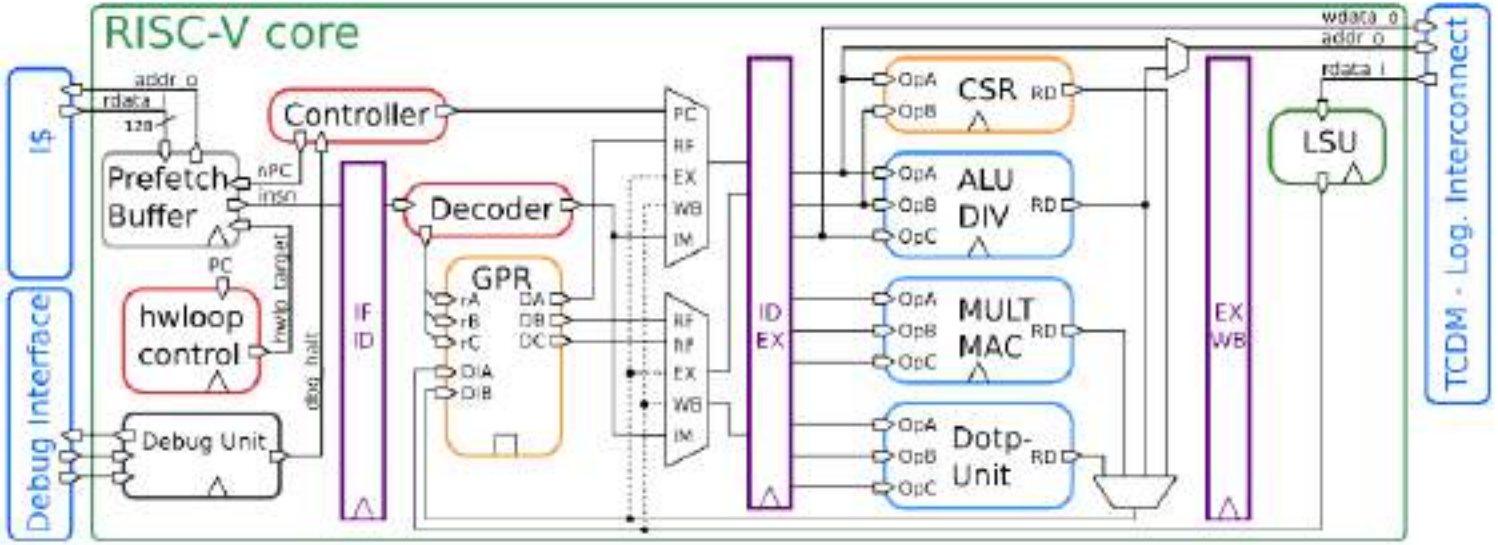
- **Zero-riscy / Ibex is suitable for simple applications**
  - Control applications, book-keeping
- **For number crunching, we need more capable cores**
  - Mainly used in clusters for signal processing / machine learning applications
- **Tuned for energy efficiency**
  - Not necessarily lowest power
- **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**

M. Gautschi *et al.*, "Near-Threshold RISC-V Core With DSP Extensions for Scalable IoT Endpoint Devices," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700-2713, Oct. 2017.

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
- **We use a lot this degree of freedom**
  - Great tool for exploring
  - The goal is to help ratify these extensions as standards through working groups

Our extensions to RI5CY & support in GCC, LLVM

- **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 (`pv.dotup.b`)

$$Z = D_1 \times K_1 + D_2 \times K_2 + D_3 \times K_3 + D_4 \times K_4$$

RI5CY ISA extensions improve performance

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

Baseline

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb     x2, 0(x10)
  lb     x3, 0(x11)
  addi   x10,x10, 1
  addi   x11,x11, 1
  add    x2, x3, x2
  sb     x2, 0(x12)
  addi   x4, x4, -1
  addi   x12,x12, 1
bne      x4, x5, Lstart
```

11 cycles/output

Auto-incr load/store

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb    x2, 0(x10!)
  lb    x3, 0(x11!)
  addi  x4, x4, -1
  add   x2, x3, x2
  sb    x2, 0(x12!)
bne     x4, x5, Lstart
```

8 cycles/output

HW Loop

```
lp.setupi 100, Lend
  lb    x2, 0(x10!)
  lb    x3, 0(x11!)
  add   x2, x3, x2
Lend:  sb x2, 0(x12!)
```

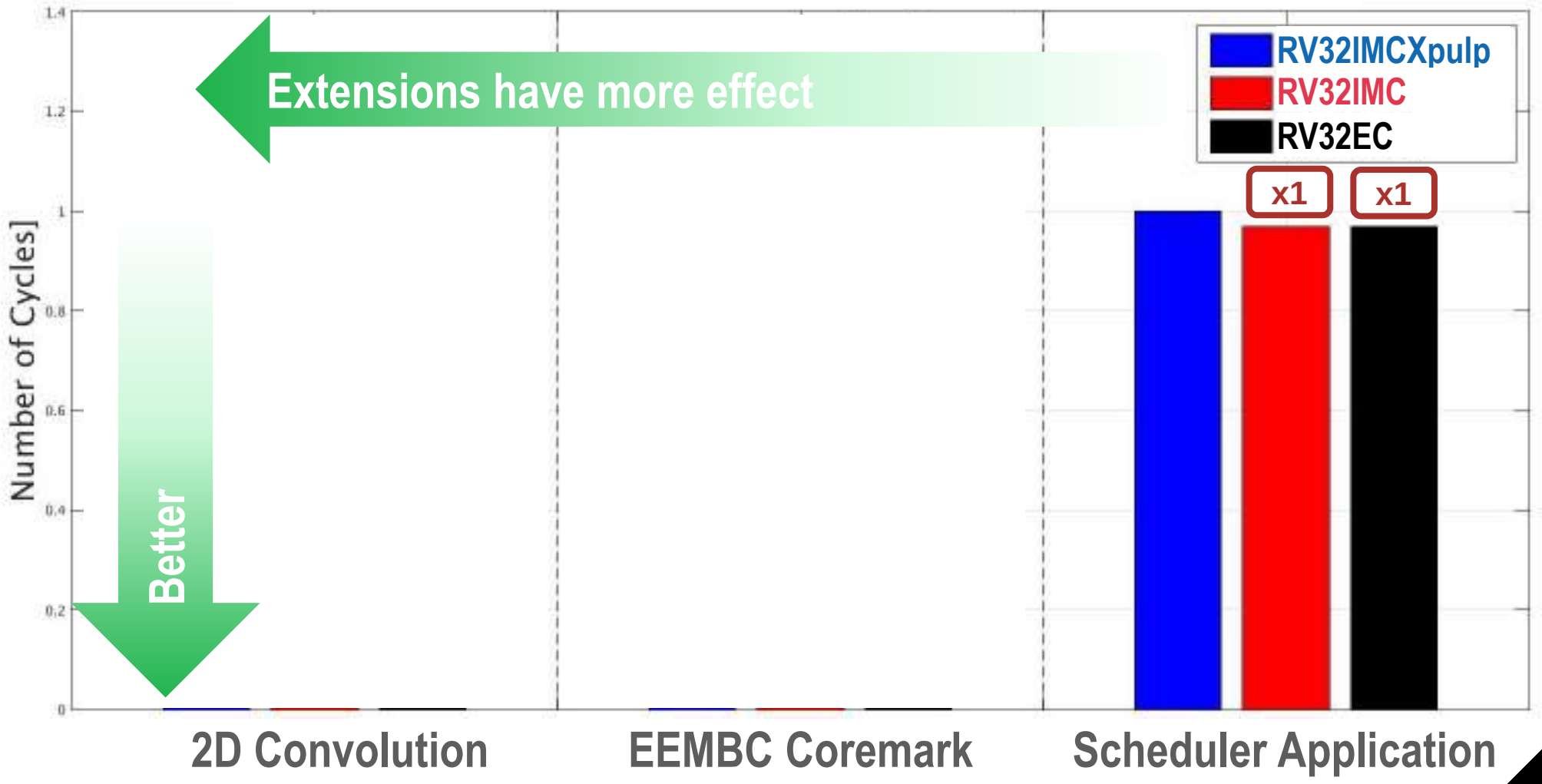
5 cycles/output

Packed-SIMD

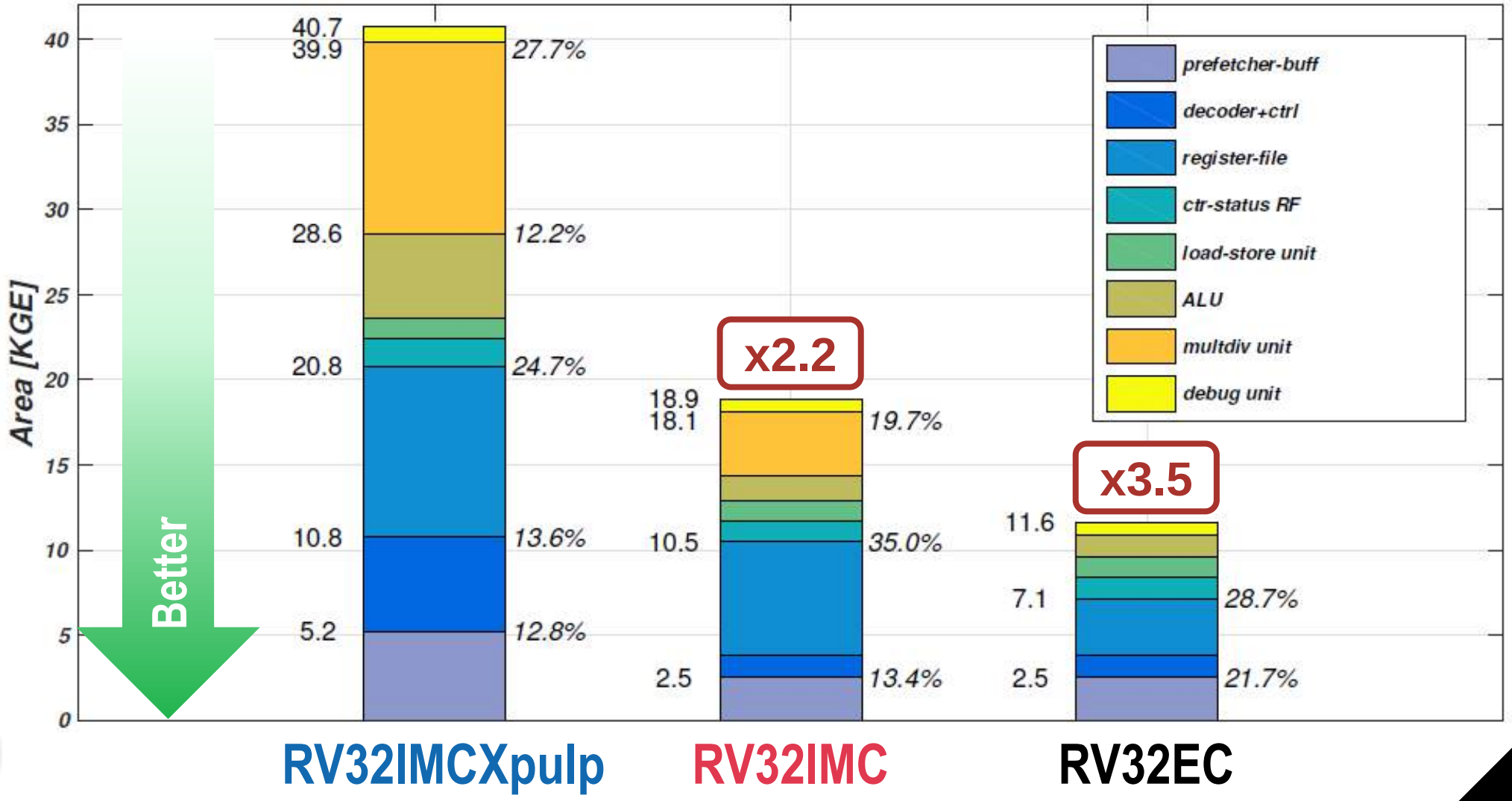
```
lp.setupi 25, Lend
  lw   x2, 0(x10!)
  lw   x3, 0(x11!)
  pv.add.b x2, x3, x2
Lend: sw x2, 0(x12!)
```

1,25 cycles/output

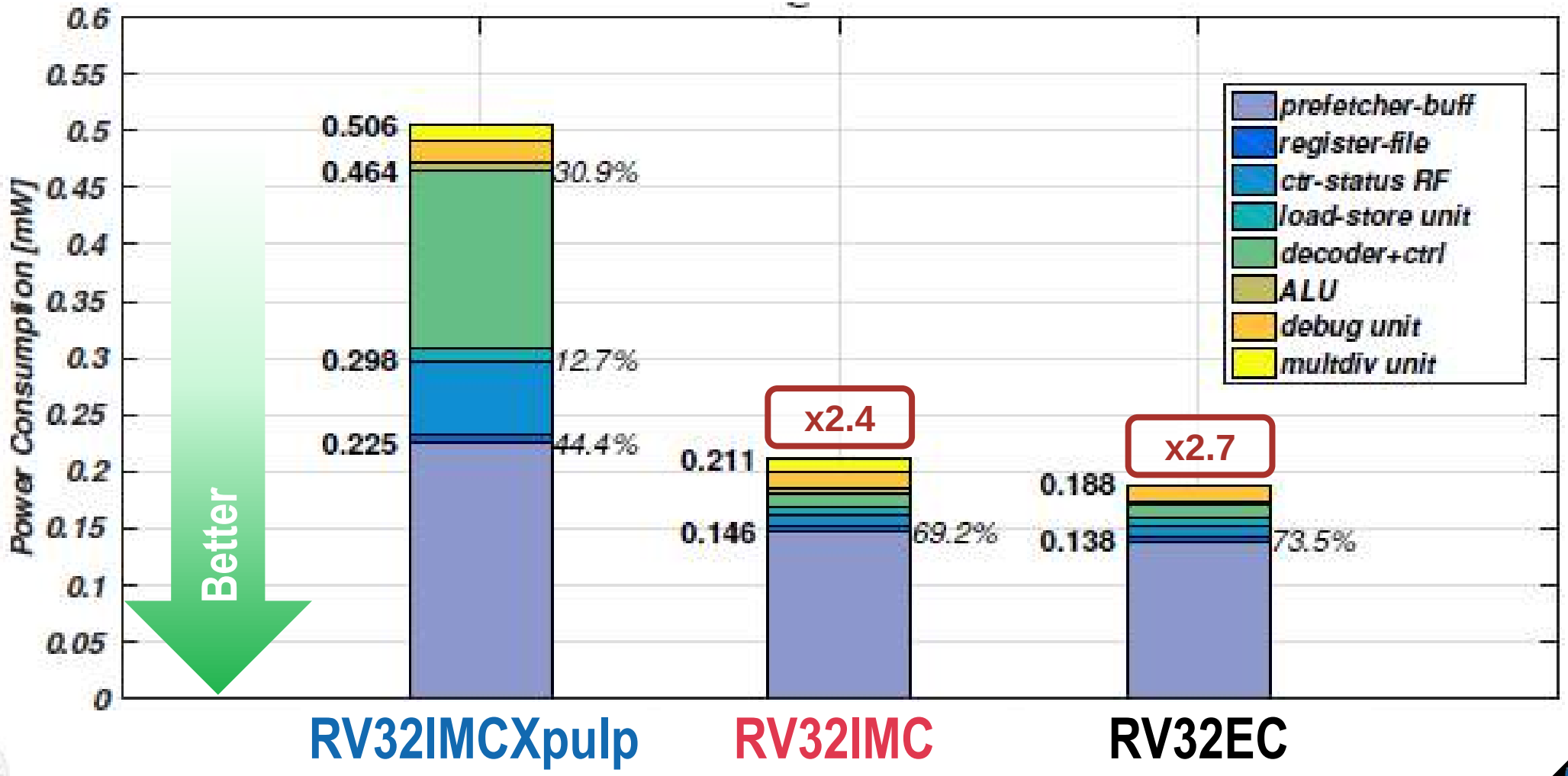
Runtime for three different applications



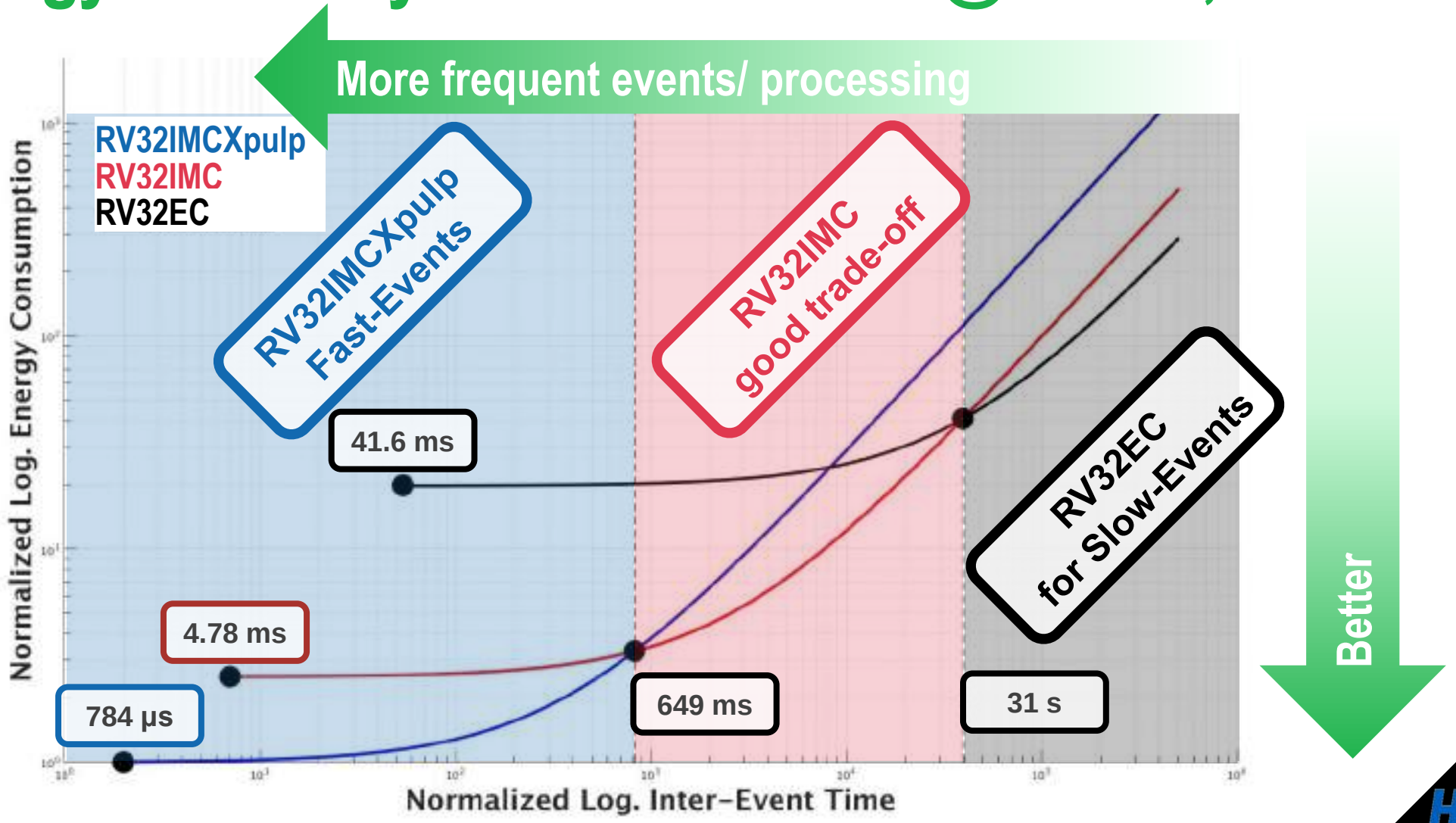
Different cores for different area budgets



Different cores for different power budgets



Energy Efficiency: 2D-Convolution @55MHz, 0.8V



This was a short overview of basics of RISC-V

- **Tomorrow, more advanced cores**
  - 64bit RISC-V core
  - Discussion on performance
  - Vector processing
- **On Wednesday-Friday, 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

PULP

Parallel Ultra Low Power

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**