



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://twitter.com/pulp_platform)



[https://www.youtube.com/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**

SW

Applications

OS

ISA

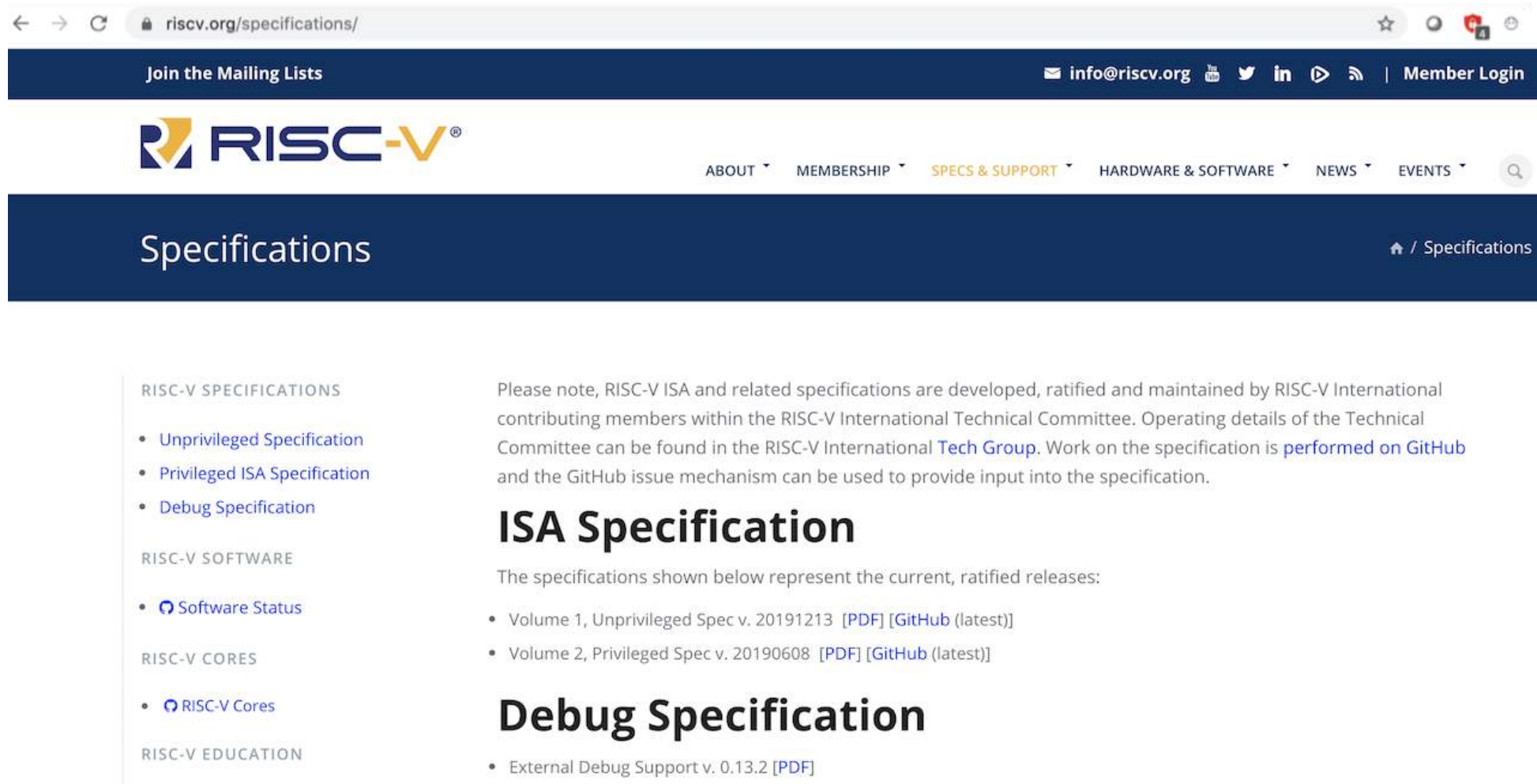
User

Privileged

HW

Debug

# RISC-V maintains basically a PDF document



The screenshot shows the RISC-V International website's specifications page. The browser address bar shows `riscv.org/specifications/`. The page has a dark blue header with navigation links: "Join the Mailing Lists", "info@riscv.org", and social media icons. Below the header is the RISC-V logo and a navigation menu with "ABOUT", "MEMBERSHIP", "SPECS & SUPPORT" (highlighted), "HARDWARE & SOFTWARE", "NEWS", and "EVENTS". The main content area has a dark blue banner with the word "Specifications" and a breadcrumb "Home / Specifications". On the left, there are four categories: "RISC-V SPECIFICATIONS" (with links to Unprivileged, Privileged ISA, and Debug specifications), "RISC-V SOFTWARE" (with a link to Software Status), "RISC-V CORES" (with a link to RISC-V Cores), and "RISC-V EDUCATION". The main text area contains a paragraph about the development process, followed by a section for "ISA Specification" listing two ratified releases (Volume 1 and Volume 2) with links to their PDFs and GitHub pages. Below that is a section for "Debug Specification" with a link to the External Debug Support PDF.

← → ↻ 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

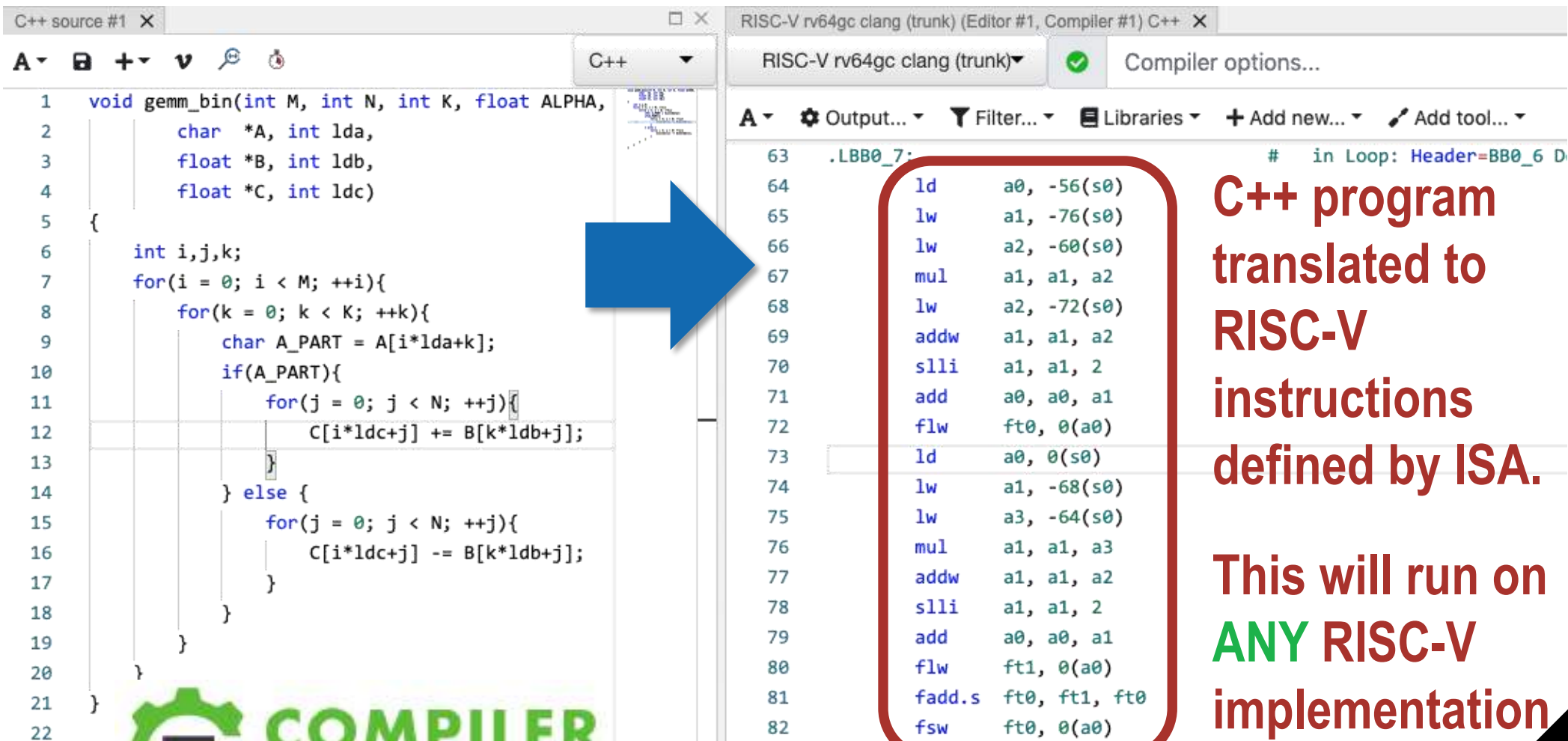
The specifications shown below represent the current, ratified releases:

- 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



The screenshot shows the Compiler Explorer interface. On the left, the C++ source code for a matrix multiplication function is displayed. A large blue arrow points from this code to the RISC-V assembly code on the right. A red box highlights a section of the assembly code, which includes instructions like `ld a0, -56(s0)`, `lw a1, -76(s0)`, `lw a2, -60(s0)`, `mul a1, a1, a2`, `lw a2, -72(s0)`, `addw a1, a1, a2`, `slli a1, a1, 2`, `add a0, a0, a1`, `flw ft0, 0(a0)`, `ld a0, 0(s0)`, `lw a1, -68(s0)`, `lw a3, -64(s0)`, `mul a1, a1, a3`, `addw a1, a1, a2`, `slli a1, a1, 2`, `add a0, a0, a1`, `flw ft1, 0(a0)`, `fadd.s ft0, ft1, ft0`, and `fsw ft0, 0(a0)`.

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

**This will run on ANY RISC-V implementation**

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



# RISC-V ISA is divided into extensions

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

- **Kept very simple and extendable**
  - Wide range of applications from IoT to HPC
- **RV + word-width + extensions**
  - RV32**IMC**: 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

<b>Q</b>	Quad-precision Floating-Point
<b>L</b>	Decimal Floating Point
<b>B</b>	Bit Manipulation
<b>T</b>	Transactional Memory
<b>P</b>	Packed SIMD
<b>J</b>	Dynamically Translated Languages
<b>V</b>	Vector Operations
<b>N</b>	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

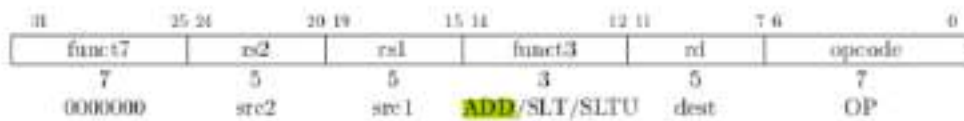
- You can access all ISAs without (many) restrictions
  - SW tools need to be developed so that they can generate code for that ISA
- Most ISAs are **closed**. Only specific vendors can implement it
  - To use a core that implements an ISA, you have to license/buy it from vendor
  - Open source SW (for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

## RISC-V

## ARM

### Integer Register-Register Operations

PV32I 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.



### 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

Base Integer Instructions: RV32I, RV64I, and RV128I					RV1 Privileged Instructions				
Category	Name	Fmt	RV32I Base	+RV(64,128)	Category	Name	RV mnemonic		
Loads	Load Byte	I	LB rd,rs1,imm		CSR Access	Atomic R/W	CSRWR rd,csr,rs1		
	Load Halfword	I	LH rd,rs1,imm			Atomic Read & Set Bit	CSRRS rd,csr,rs1		
	Load Word	I	LW rd,rs1,imm	L{D Q} rd,rs1,imm		Atomic Read & Clear Bit	CSRRC rd,csr,rs1		
	Load Byte Unsigned	I	LBU rd,rs1,imm			Atomic R/W Imm	CSRWRI rd,csr,imm		
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W D}U rd,rs1,imm		Atomic Read & Clear Imm	CSRRCI rd,csr,imm		
Stores	Store Byte	S	SB rs1,rs2,imm		<b>Privilege Mode</b>				
	Store Halfword	S	SH rs1,rs2,imm		Change Level	Env. Call	SCALL		
	Store Word	S	SW rs1,rs2,imm	S{D Q} rs1,rs2,imm	Environment Base	Env. Base	EBREAK		
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W D} rd,rs1,rs2	Trap Redirect to Supervisor	MRTS			
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W D} rd,rs1,shamt	Redirect Trap to Hypervisor	MNRTH			
	Shift Right	R	SRL rd,rs1,rs2	SRL{W D} rd,rs1,rs2	Hypervisor Trap to Supervisor	MRTS			
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W D} rd,rs1,shamt	Interrupt Wait for Interrupt	WFI			
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W D} rd,rs1,rs2	MMU Supervisor FENCE	SBFENCE.VM rs1			
Arithmetic	ADD	I	ADD rd,rs1,rs2	ADD{W D} rd,rs1,rs2					
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W D} rd,rs1,imm					
	SUBtract	I	SUB rd,rs1,rs2						
	Load Upper Imm	I	LUI rd,imm						
	Add Upper Imm to PC	I	AUIPC rd,imm						
Logical	XOR	R	XOR rd,rs1,rs2						
	XOR Immediate	I	XORI rd,rs1,imm						
	OR	R	OR rd,rs1,rs2						
	OR Immediate	I	ORI rd,rs1,imm						
	AND	R	AND rd,rs1,rs2						
Compare	AND Immediate	I	ANDI rd,rs1,imm						
	Set <	R	SLT rd,rs1,rs2						
	Set < Immediate	I	SLTI rd,rs1,imm						
	Set < Unsigned	R	SLTU rd,rs1,rs2						
	Set < Imm Unsigned	R	SLTIU rd,rs1,imm						
Branches	Branch =	R	BEQ rs1,rs2,imm						
	Branch ≠	R	BNE rs1,rs2,imm						
	Branch <	R	BLT rs1,rs2,imm						
	Branch ≥	R	BGE rs1,rs2,imm						
	Branch < Unsigned	R	BLTU rs1,rs2,imm						
Jump & Link	Branch ≥ Unsigned	R	BGEU rs1,rs2,imm						
	JAL	UJ	JAL rd,imm						
	Jump & Link Register	UJ	JALR rd,rs1,imm						
	Synch thread	I	FENCE						
	Synch Instr & Data	I	FENCE.I						
System	System Call	I	SCALL						
	System Break	I	SBREAK						
	Read CYCLE	I	RDYCYCLE rd						
	Read CYCLE upper Half	I	RDYCYCLEH rd						
	Read TIME	I	RDTIME rd						
Counters	Read TIME upper Half	I	RDTIMEH rd						
	Read INSTR RETired	I	RDINSTRET rd						
	Read INSTR upper Half	I	RDINSTRETH rd						

32-bit Instruction Formats														
R	31	30	25-24	21	20	19	15-14	12-11	8	7	6	0	0	0
I	funct7			rs2			funct3		rd		opcode			
S	imm[11:0]			rs2			funct3		rd		opcode			
B	imm[11:0]			rs2			funct3		imm[4:0]		opcode			
UJ	imm[20:1]			rs2			funct3		imm[4:1]		imm[11]		rd	opcode
	imm[20:1]			imm[11]			imm[19:12]						rd	opcode

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 RV1 base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RV1 instruction. See risc.org.

Optional Multiply-Divide Instructions Extension (M)									
Category	Name	Fmt	RV32M (Multiply-Divide)	+RV(64,128)					
Multiply	Multiply	R	MUL rd,rs1,rs2	MUL{W D} rd,rs1,rs2					
	Multiply upper Half	R	MULH rd,rs1,rs2						
	Multiply Half Sign	R	MULHS rd,rs1,rs2						
	Multiply upper Half Sign	R	MULHUS rd,rs1,rs2						
Divide	Divide	R	DIV rd,rs1,rs2	DIV{W D} rd,rs1,rs2					
	Divide Unsigned	R	DIVU rd,rs1,rs2						
Remainder	REMAinder	R	REM rd,rs1,rs2	REM{W D} rd,rs1,rs2					
	REMAinder Unsigned	R	REMU rd,rs1,rs2	REMU{W D} rd,rs1,rs2					

Optional Atomic Instructions Extension (A)									
Category	Name	Fmt	RV32A (Atomic)	+RV(64,128)					
Load	Load Reserved	R	LR.W rd,rs1	LR.{D Q} rd,rs1					
Store	Store Conditional	R	SC.W rd,rs1,rs2	SC.{D Q} rd,rs1,rs2					
Swap	SWAP	R	AMOSWAP.W rd,rs1,rs2	AMOSWAP.{D Q} rd,rs1,rs2					
Add	ADD	R	AMOADD.W rd,rs1,rs2	AMOADD.{D Q} rd,rs1,rs2					
Logical	AND	R	AND.W rd,rs1,rs2	AND.{D Q} rd,rs1,rs2					
	OR	R	OR.W rd,rs1,rs2	OR.{D Q} rd,rs1,rs2					
	ANDNOT	R	ANDNOT.W rd,rs1,rs2	ANDNOT.{D Q} rd,rs1,rs2					
	ORNOT	R	ORNOT.W rd,rs1,rs2	ORNOT.{D Q} rd,rs1,rs2					
Min/Max	MINimum	R	AMIN.W rd,rs1,rs2	AMININ.{D Q} rd,rs1,rs2					
	MAXimum	R	AMAX.W rd,rs1,rs2	AMAXIN.{D Q} rd,rs1,rs2					
	MINimum Unsigned	R	AMINU.W rd,rs1,rs2	AMINUIN.{D Q} rd,rs1,rs2					
	MAXimum Unsigned	R	AMAXU.W rd,rs1,rs2	AMAXUIN.{D Q} rd,rs1,rs2					

Three Optional Floating-Point Instructions Extensions: RVF, RVF6, & RVF16									
Category	Name	Fmt	RV32F (F D Q) (HP/SP,DP,OP,FP,FC)	+RV(64,128)					
Move	Move from Integer	R	FMV.W{H S}.X rd,rs1	FMV.{D Q}.X rd,rs1					
	Move to Integer	R	FMV.X{H S}.W rd,rs1	FMV.X.{D Q} rd,rs1					
Convert	Convert from Int	R	FCVTF.{H S D Q}.W rd,rs1	FCVTF.{H S D Q}.L{T} rd,rs1					
	Convert from Int Unsigned	R	FCVTFU.{H S D Q}.WU rd,rs1	FCVTFU.{H S D Q}.L{T}U rd,rs1					
	Convert to Int	R	FCVTT.W{H S D Q} rd,rs1	FCVTT.L{T}.H{S D Q} rd,rs1					
	Convert to Int Unsigned	R	FCVTTU.WU{H S D Q} rd,rs1	FCVTTU.L{T}U.H{S D Q}U rd,rs1					
Load	Load	R	FLW.W{D Q} rd,rs1,imm						
Store	Store	R	FSW.W{D Q} rs1,rs2,imm						
	ADD	R	FADD.{S D Q} rd,rs1,rs2						
	SUBTRACT	R	FSUB.{S D Q} rd,rs1,rs2						
	MULTIPLY	R	FMUL.{S D Q} rd,rs1,rs2						
	DIVIDE	R	FDIV.{S D Q} rd,rs1,rs2						
Mul-Add	Square Root	R	FSQRT.{S D Q} rd,rs1						
	Multiply-Add	R	FMAADD.{S D Q} rd,rs1,rs2,rs3						
Sign Inject	Multiply-Subtract	R	FMSUB.{S D Q} rd,rs1,rs2,rs3						
	Negative Multiply-Subtract	R	FNMSUB.{S D Q} rd,rs1,rs2,rs3						
	Negative Multiply-Add	R	FNMADD.{S D Q} rd,rs1,rs2,rs3						
	Sign source	R	FNMJZ.{S D Q} rd,rs1,rs2						
Min/Max	Negative SIGN source	R	FNMJNUS.{S D Q} rd,rs1,rs2						
	Xor SIGN source	R	FNMJXU.{S D Q} rd,rs1,rs2						
	MINimum	R	FMIN.{S D Q} rd,rs1,rs2						
	MAXimum	R	FMAX.{S D Q} rd,rs1,rs2						
Compare	Compare Float =	R	FEQ.{S D Q} rd,rs1,rs2						
	Compare Float <	R	FLT.{S D Q} rd,rs1,rs2						
	Compare Float <=	R	FLE.{S D Q} rd,rs1,rs2						
	Classify Type	R	FCCLASS.{S D Q} rd,rs1						
Configuration	Read Status	R	FRCSR rd						
	Read Rounding Mode	R	FRFRM rd						
	Read Flags	R	FRFLGAS rd						
	Swap Status Reg	R	FRSCSR rd,rs1						
	Swap Rounding Mode	R	FRSRM rd,rs1						
	Swap Flags	R	FRSFLGAS rd,rs1						
	Swap Rounding Mode Imm	R	FRSRMI rd,imm						
	Swap Flags Imm	R	FRSFLGAS rd,imm						

RISC-V Cal		
Register	ABI Name	Saver
x0	zero	---
x1	ra	Caller
x2	sp	Callee
x3	gp	---
x4	tp	---
x5-7	t0-t2	Caller
x8	s0/fp	Callee
x9	s1	Callee
x10-11	a0-1	Caller
x12-17	a2-7	Callee
x18-27	s2-11	Callee
x28-31	t3-t6	Caller
ft0-7	ft0-7	Caller
fs-9	fs0-1	Callee
f10-11	fa0-1	Caller
f12-17	fa2-7	Caller
f18-27	fs2-11	Callee
f28-31	ft8-11	Caller

Floating-Point Extensions									
Category	Name	Fmt	RV32F (F D Q)	RV64F (F D Q)	RV128F (F D Q)				
Load	Load Single	R	FLW.W rd,rs1,imm	FLW.D rd,rs1,imm	FLW.Q rd,rs1,imm				
	Load Double	R	FLD.D rd,rs1,imm	FLD.D rd,rs1,imm	FLD.Q rd,rs1,imm				
	Load Quad	R	FLQ.Q rd,rs1,imm	FLQ.Q rd,rs1,imm	FLQ.Q rd,rs1,imm				
	Load Quad SP	R	FLQSP.Q rd,imm	FLQSP.Q rd,imm	FLQSP.Q rd,imm				
Store	Store Single	R	FSW.W rs1,rs2,imm	FSW.D rs1,rs2,imm	FSW.Q rs1,rs2,imm				
	Store Double	R	FSW.D rs1,rs2,imm	FSW.D rs1,rs2,imm	FSW.Q rs1,rs2,imm				
	Store Quad	R	FSW.Q rs1,rs2,imm	FSW.Q rs1,rs2,imm	FSW.Q rs1,rs2,imm				
	Store Quad SP	R	FSWQSP.Q rs2,imm	FSWQSP.Q rs2,imm	FSWQSP.Q rs2,imm				
Arithmetic	ADD	R	FADD.{S D Q} rd,rs1,rs2	FADD.{S D Q} rd,rs1,rs2	FADD.{S D Q} rd,rs1,rs2				
	SUBTRACT	R	FSUB.{S D Q} rd,rs1,rs2	FSUB.{S D Q} rd,rs1,rs2	FSUB.{S D Q} rd,rs1,rs2				
	MULTIPLY	R	FMUL.{S D Q} rd,rs1,rs2	FMUL.{S D Q} rd,rs1,rs2	FMUL.{S D Q} rd,rs1,rs2				
	DIVIDE	R	FDIV.{S D Q} rd,rs1,rs2	FDIV.{S D Q} rd,rs1,rs2	FDIV.{S D Q} rd,rs1,rs2				
Mul-Add	Square Root	R	FSQRT.{S D Q} rd,rs1	FSQRT.{S D Q} rd,rs1	FSQRT.{S D Q} rd,rs1				
	Multiply-Add	R	FMAADD.{S D Q} rd,rs1,rs2,rs3	FMAADD.{S D Q} rd,rs1,rs2,rs3	FMAADD.{S D Q} rd,rs1,rs2,rs3				
	Multiply-Subtract	R	FMSUB.{S D Q} rd,rs1,rs2,rs3	FMSUB.{S D Q} rd,rs1,rs2,rs3	FMSUB.{S D Q} rd,rs1,rs2,rs3				
	Negative Multiply-Subtract	R	FNMSUB.{S D Q} rd,rs1,rs2,rs3	FNMSUB.{S D Q} rd,rs1,rs2,rs3	FNMSUB.{S D Q} rd,rs1,rs2,rs3				
Sign Inject	Negative Multiply-Add	R	FNMADD.{S D Q} rd,rs1,rs2,rs3	FNMADD.{S D Q} rd,rs1,rs2,rs3	FNMADD.{S D Q} rd,rs1,rs2,rs3				
	Sign source	R	FNMJZ.{S D Q} rd,rs1,rs2	FNMJZ.{S D Q} rd,rs1,rs2	FNMJZ.{S D Q} rd,rs1,rs2				
	Negative SIGN source	R	FNMJNUS.{S D Q} rd,rs1,rs2	FNMJNUS.{S D Q} rd,rs1,rs2	FNMJNUS.{S D Q} rd,rs1,rs2				
	Xor SIGN source	R	FNMJXU.{S D Q} rd,rs1,rs2	FNMJXU.{S D Q} rd,rs1,rs2	FNMJXU.{S D Q} rd,rs1,rs2				
Min/Max	MINimum	R	FMIN.{S D Q} rd,rs1,rs2	FMIN.{S D Q} rd,rs1,rs2	FMIN.{S D Q} rd,rs1,rs2				
	MAXimum	R	FMAX.{S D Q} rd,rs1,rs2	FMAX.{S D Q} rd,rs1,rs2	FMAX.{S D Q} rd,rs1,rs2				
	Compare Float =	R	FEQ.{S D Q} rd,rs1,rs2	FEQ.{S D Q} rd,rs1,rs2	FEQ.{S D Q} rd,rs1,rs2				
	Compare Float <	R	FLT.{S D Q} rd,rs1,rs2	FLT.{S D Q} rd,rs1,rs2	FLT.{S D Q} rd,rs1,rs2				
Compare	Compare Float <=	R	FLE.{S D Q} rd,rs1,rs2	FLE.{S D Q} rd,rs1,rs2	FLE.{S D Q} rd,rs1,rs2				
	Classify Type	R	FCCLASS.{S D Q} rd,rs1	FCCLASS.{S D Q} rd,rs1	FCCLASS.{S D Q} rd,rs1				
	Configuration	R	FRCSR rd	FRCSR rd	FRCSR rd				
	Read Status	R	FRFRM rd	FRFRM rd	FRFRM rd				
Read Rounding Mode	Read Flags	R	FRFLGAS rd	FRFLGAS rd	FRFLGAS rd				
	Swap Status Reg	R	FRSCSR rd,rs1	FRSCSR rd,rs1	FRSCSR rd,rs1				
	Swap Rounding Mode	R	FRSRM rd,rs1	FRSRM rd,rs1	FRSRM rd,rs1				
	Swap Flags	R	FRSFLGAS rd,rs1	FRSFLGAS rd,rs1	FRSFLGAS rd,rs1				
Swap Rounding Mode Imm	Swap Rounding Mode Imm	R	FRSRMI rd,imm	FRSRMI rd,imm	FRSRMI rd,imm				
	Swap Flags Imm	R	FRSFLGAS rd,imm	FRSFLGAS rd,imm	FRSFLGAS rd,imm				
	Swap Rounding Mode Imm	R	FRSRMI rd,imm	FRSRMI rd,imm	FRSRMI rd,imm				
	Swap Flags Imm	R	FRSFLGAS rd,imm	FRSFLGAS rd,imm	FRSFLGAS rd,imm				

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 Calling Convention				
Register	ABI Name	Saver	Description	
x0	zero	---	Hard-wired zero	
x1	ra	Caller	Return address	
x2	sp	Callee	Stack pointer	
x3	gp	---	Global pointer	
x4	tp	---	Thread pointer	
x5-7	t0-t2	Caller	Temporaries	
x8	s0/fp	Callee	Saved register/frame pointer	
x9	s1	Callee	Saved register	
x10-11	a0-1	Caller	Function arguments/return values	
x12-17	a2-7	Caller	Function arguments	
x18-27	s2-11	Callee	Saved registers	
x28-31	t3-t6	Caller	Temporaries	
f0-7	ft0-7	Caller	FP temporaries	
f8-9	fs0-1	Callee	FP saved registers	
f10-11	fs0-1	Caller	FP arguments/return values	
f12-17	fs2-7	Caller	FP arguments	
f18-27	fs2-11	Callee	FP saved registers	
f28-31	ft8-11	Caller	FP temporaries	



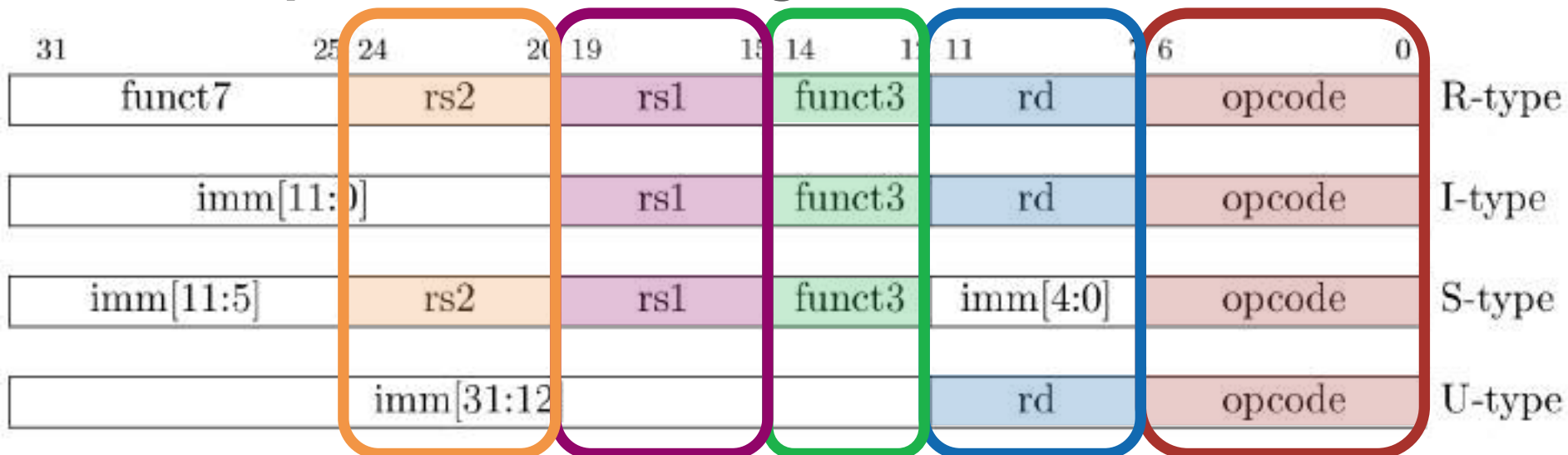


# RISC-V Architectural State

- **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 **i**mmediate/constant values
- **S / SB** operations with two **s**ource 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	<i>custom-0</i>	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	<i>custom-1</i>	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	<i>reserved</i>	<i>custom-2/rv128</i>	48b
11	BRANCH	JALR	<i>reserved</i>	JAL	SYSTEM	<i>reserved</i>	<i>custom-3/rv128</i>	$\geq 80b$



# 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**

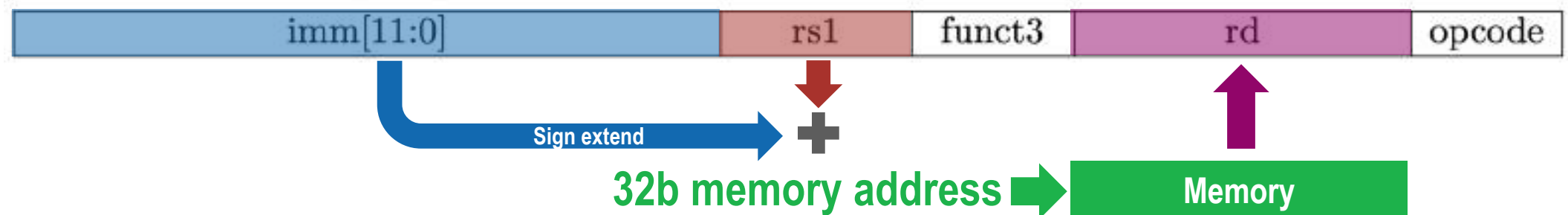
# Constants (Immediates) in Instructions

- 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

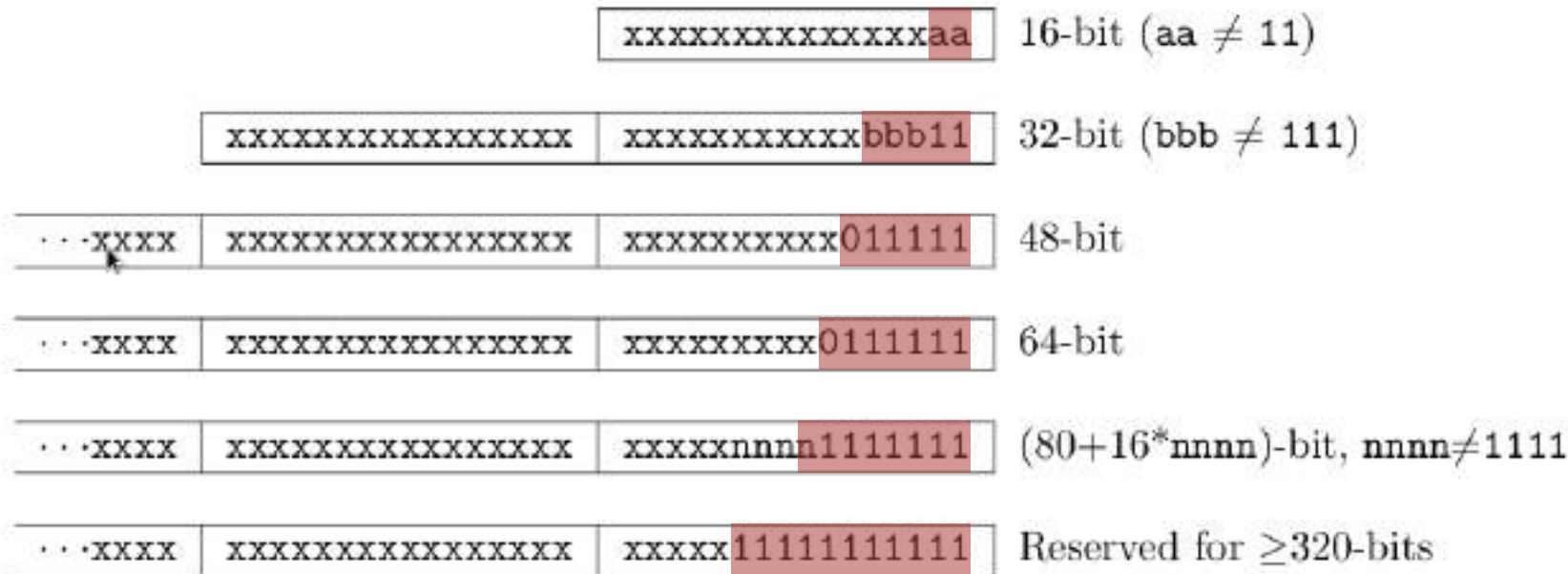
`bne x10, x11, 2000` // if `x10 != x11`, jump 2000 ahead



- 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



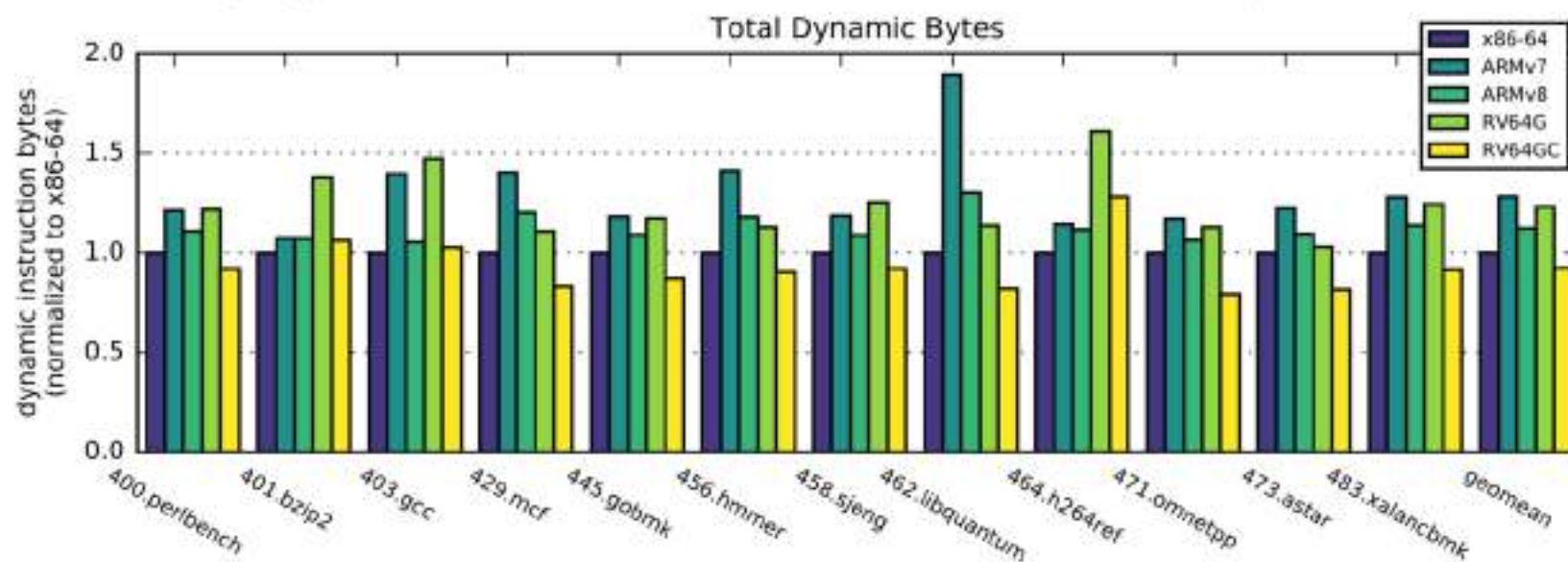
Byte Address: base+4

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 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<sup>2</sup> (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

### Low Cost Core

- Zero-riscy

- RV32-ICM

- Micro-riscy

- RV32-ICM

**Ibex**  
by LowRISC

### DSP Enhanced Core

- RI5CY

- RV32-ICMDFX

- SIMD

- Hypothesis

- Branch manipulation

- Fixed point

**CV32E40P**  
by OpenHW

### Streaming Compute Core

- Snitch

- RV32-ICMDFX

## 64 bit

### Linux capable Core

- Ariane

- RV64-IC(MA)

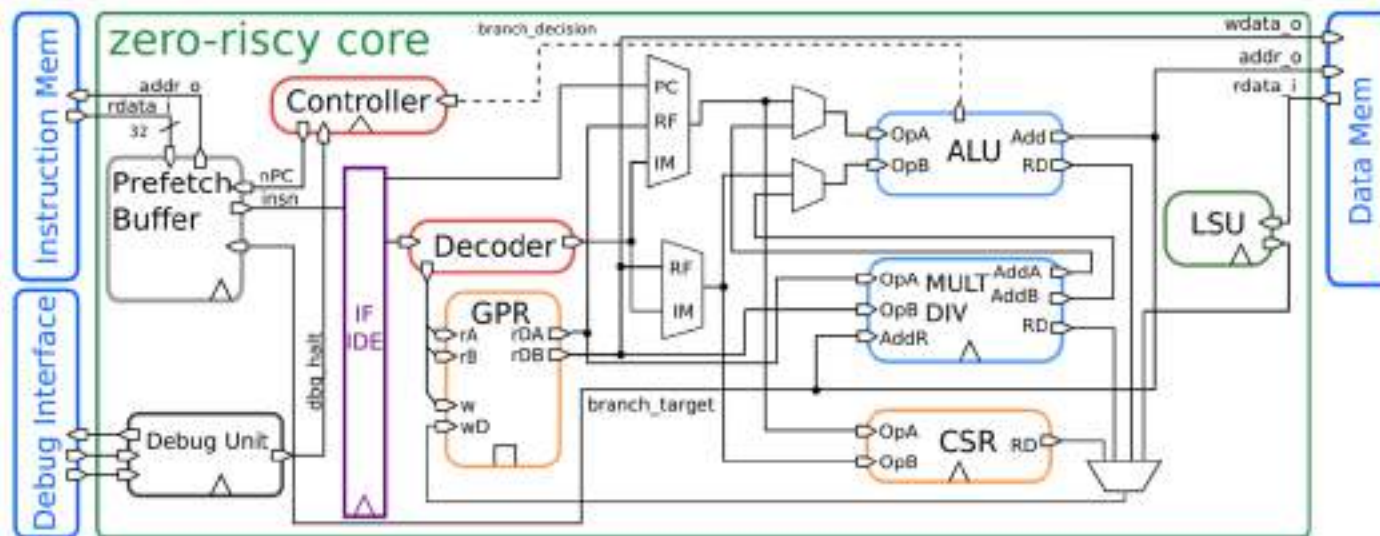
- Full

**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+

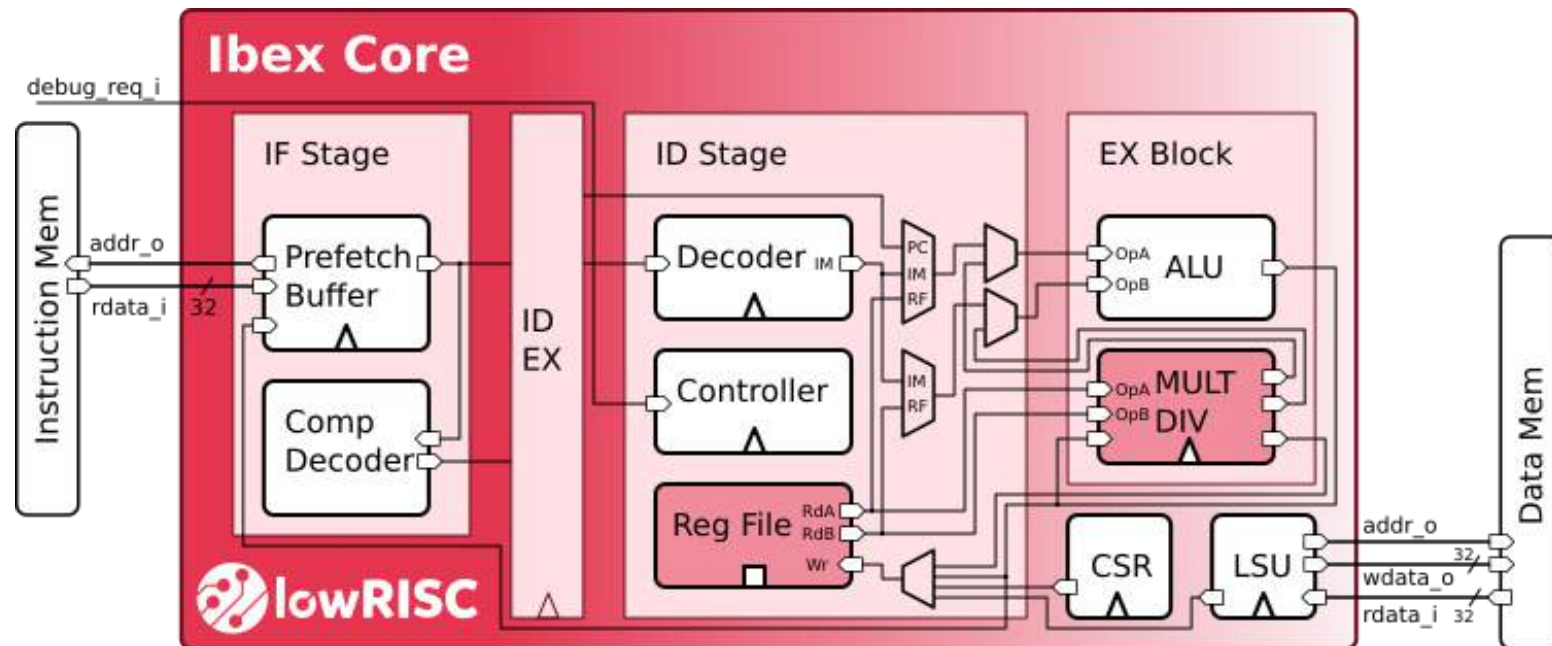
Contributors

680

Pull Requests

314

GitHub Issues



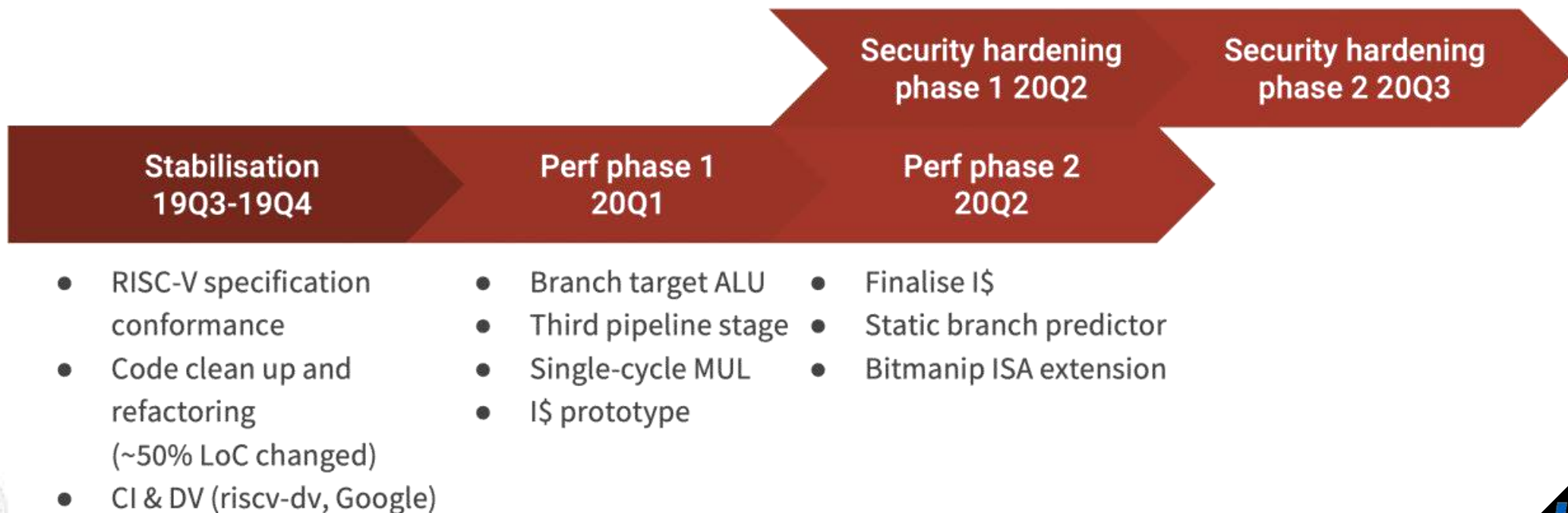
*Ibex is a small and efficient, 32-bit, in-order RISC-V core with a 2-stage (or optionally 3-stage) pipeline that implements the RV32IMCB instruction set architecture.*

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

# Roadmap of Ibex



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





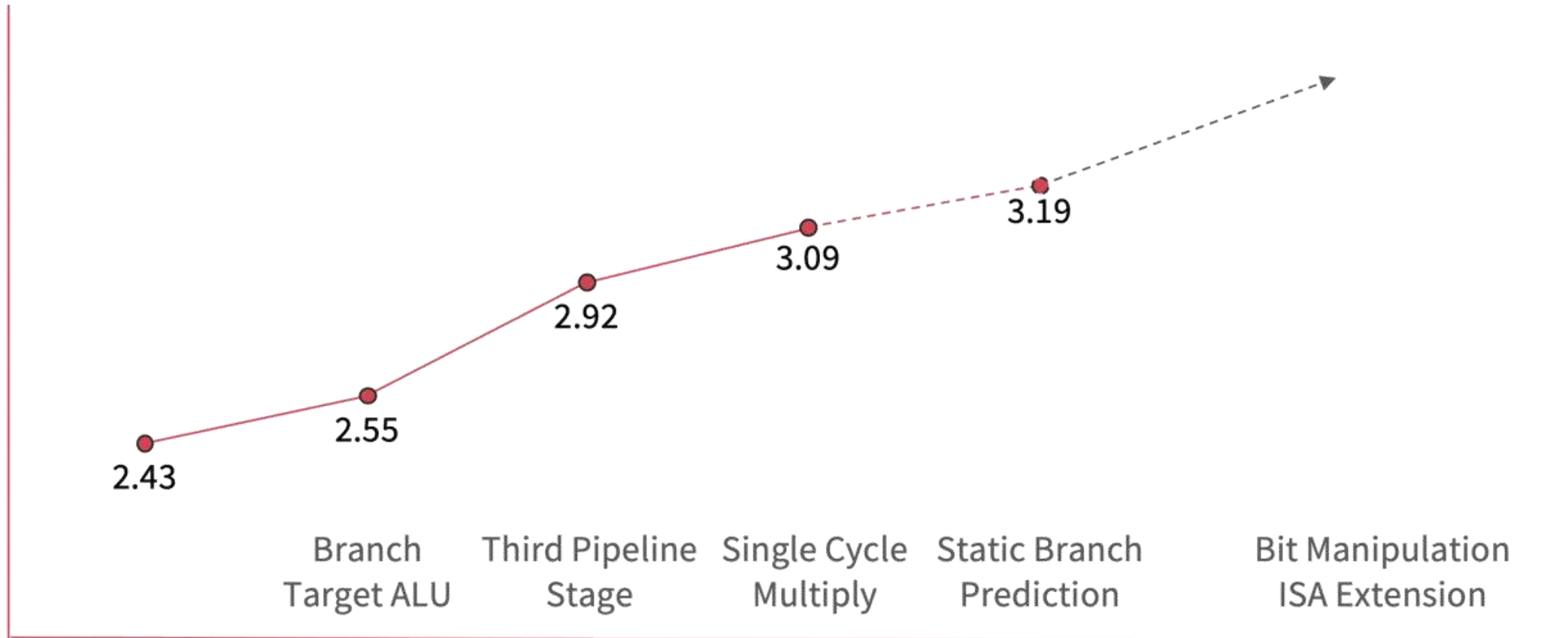


# Growth of Ibex measured with Coremark/MHz

Past Work

Today

Future



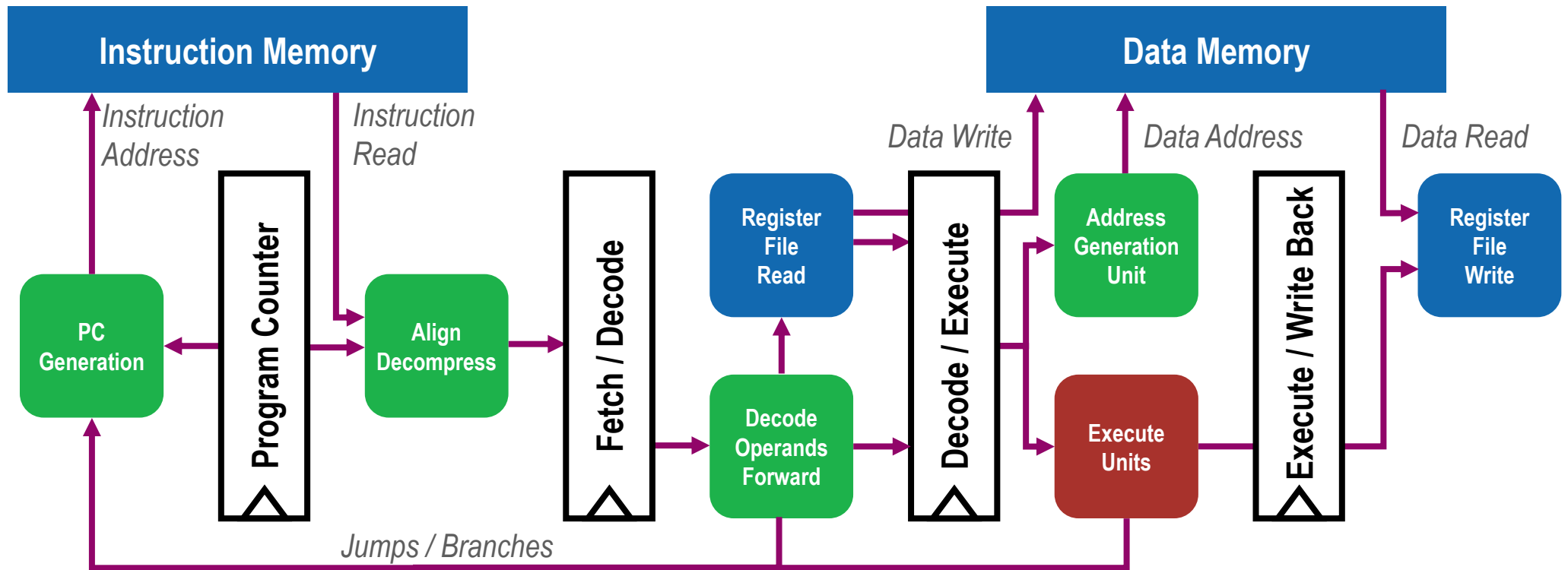




# RI5CY / CV32E40P our main 32bit RISC-V core

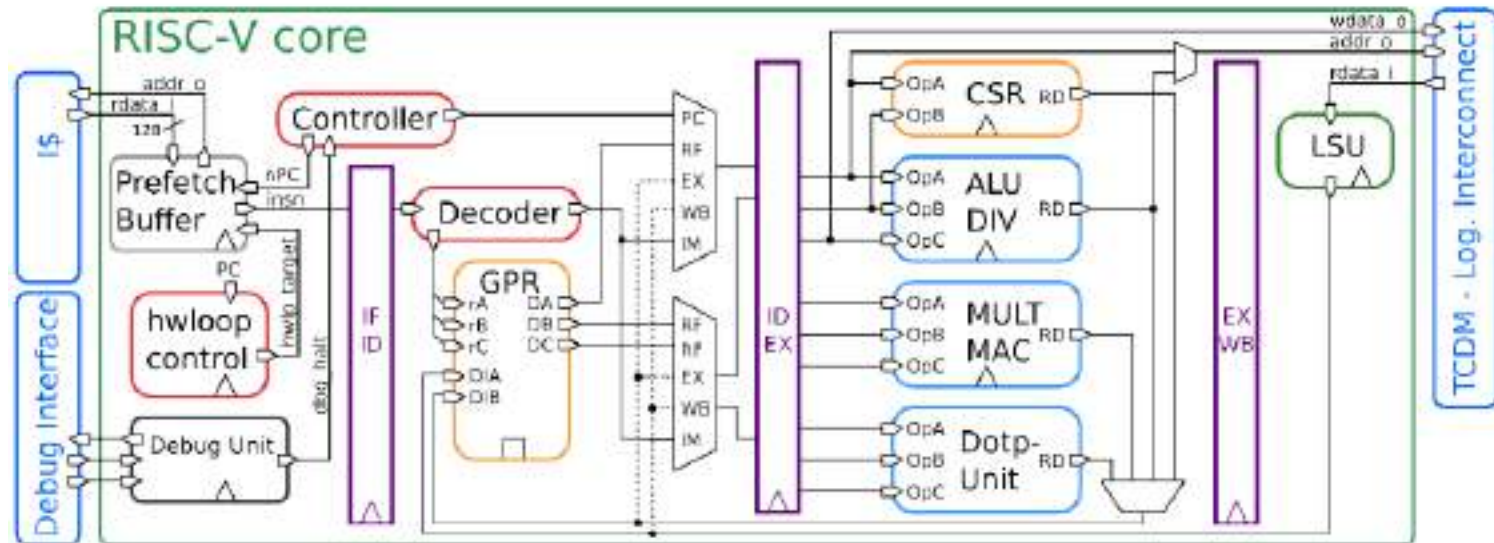
- **Zero-riscy / Ibex is suitable for simple applications**
  - Control applications, book-keeping
- **For number crunching, we need more capable cores**
  - Mainly used in clusters for signal processing / machine learning applications
- **Tuned for energy efficiency**
  - Not necessarily lowest power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

```
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

## lp.setupi 100, Lend

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

5 cycles/output

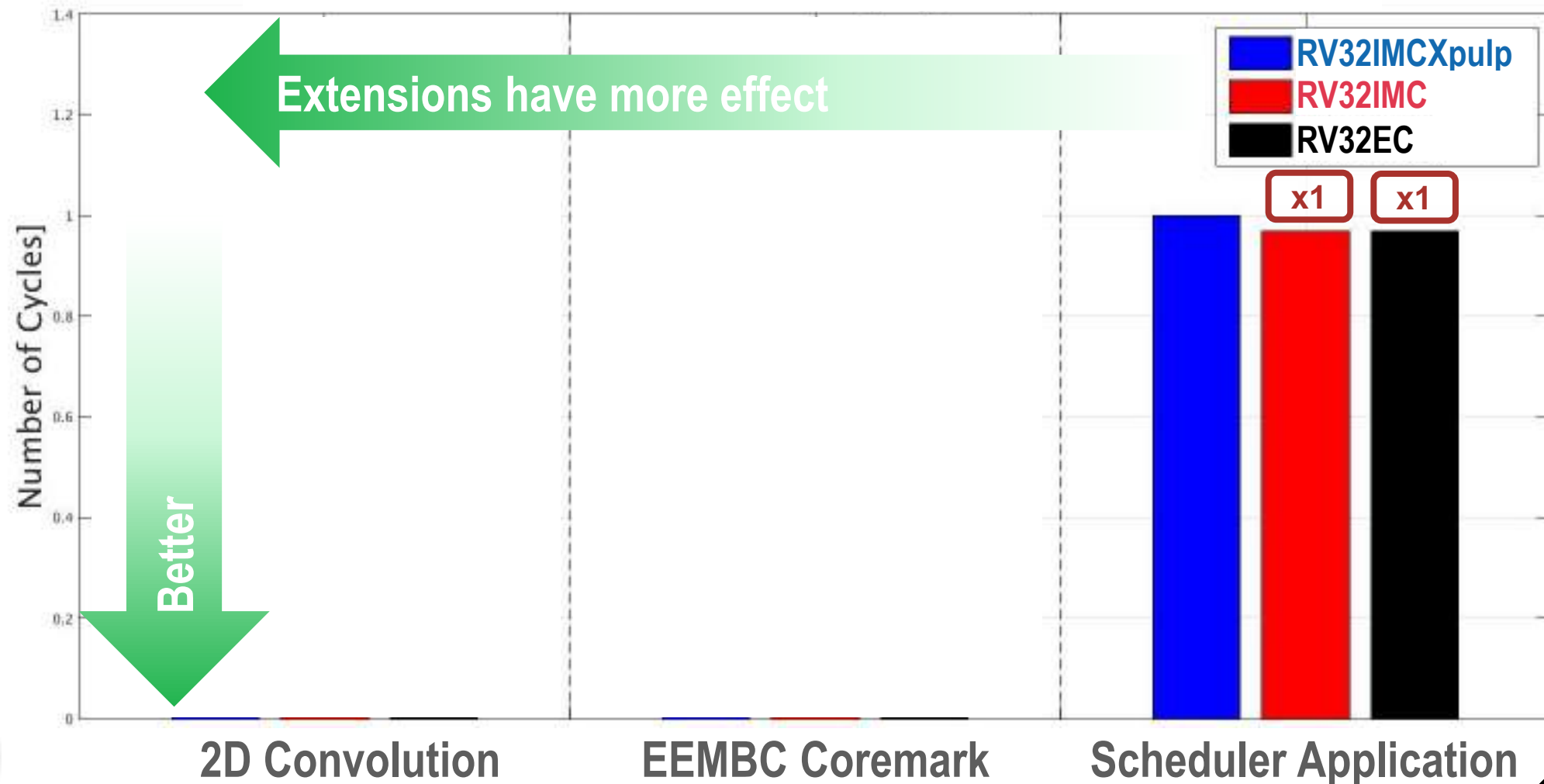
## Packed-SIMD

```
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

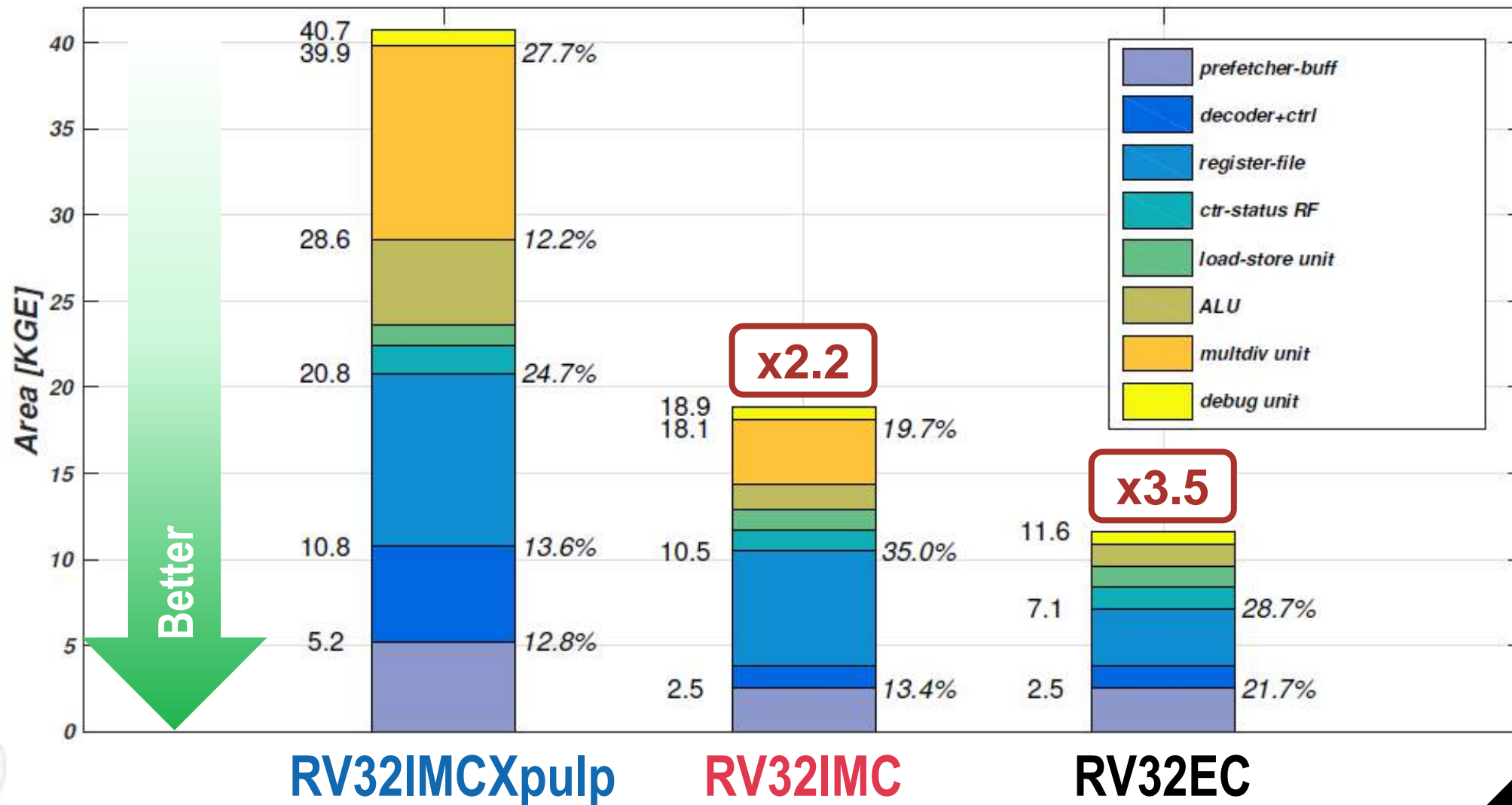


# Runtime for three different applications

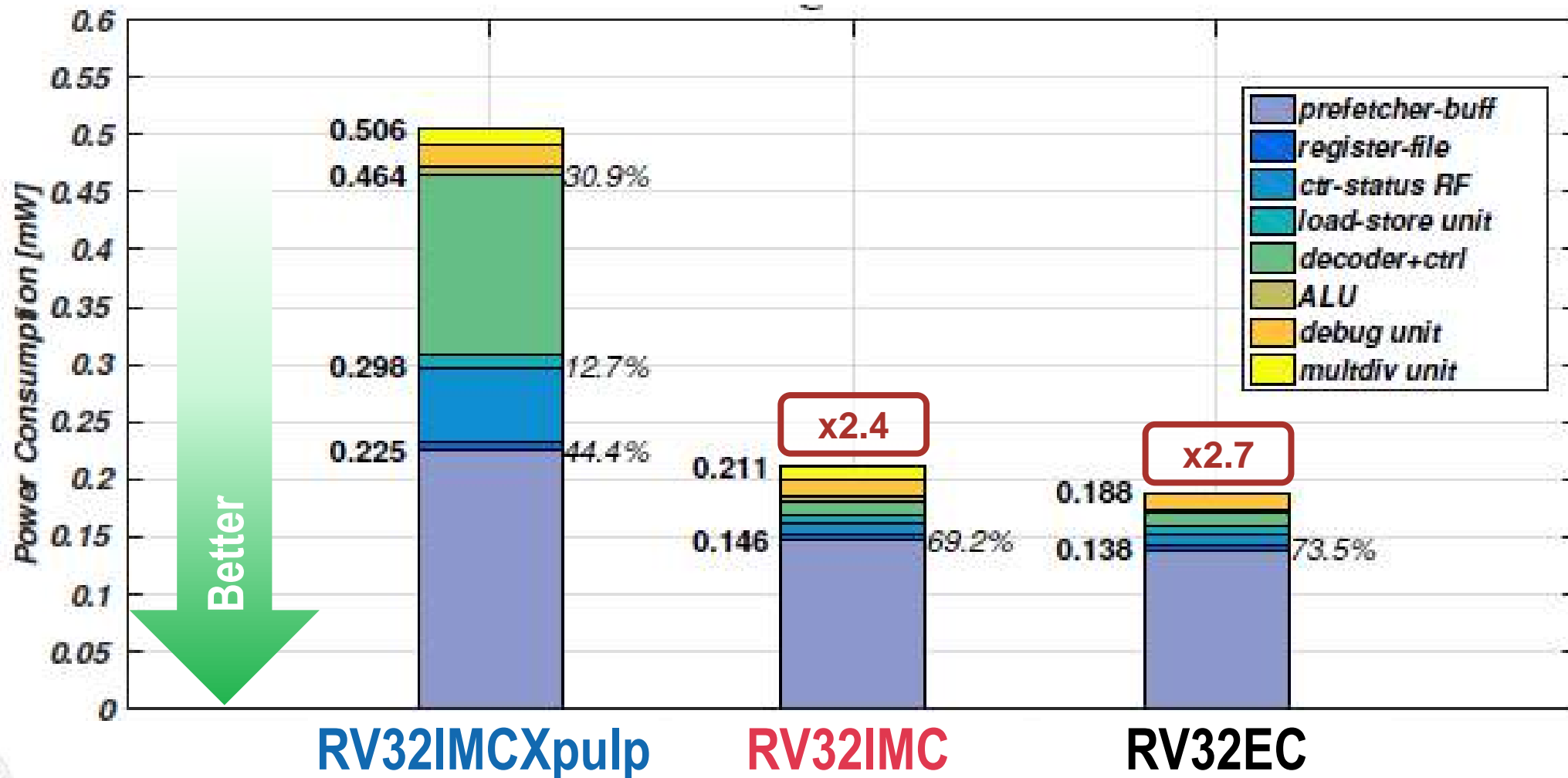




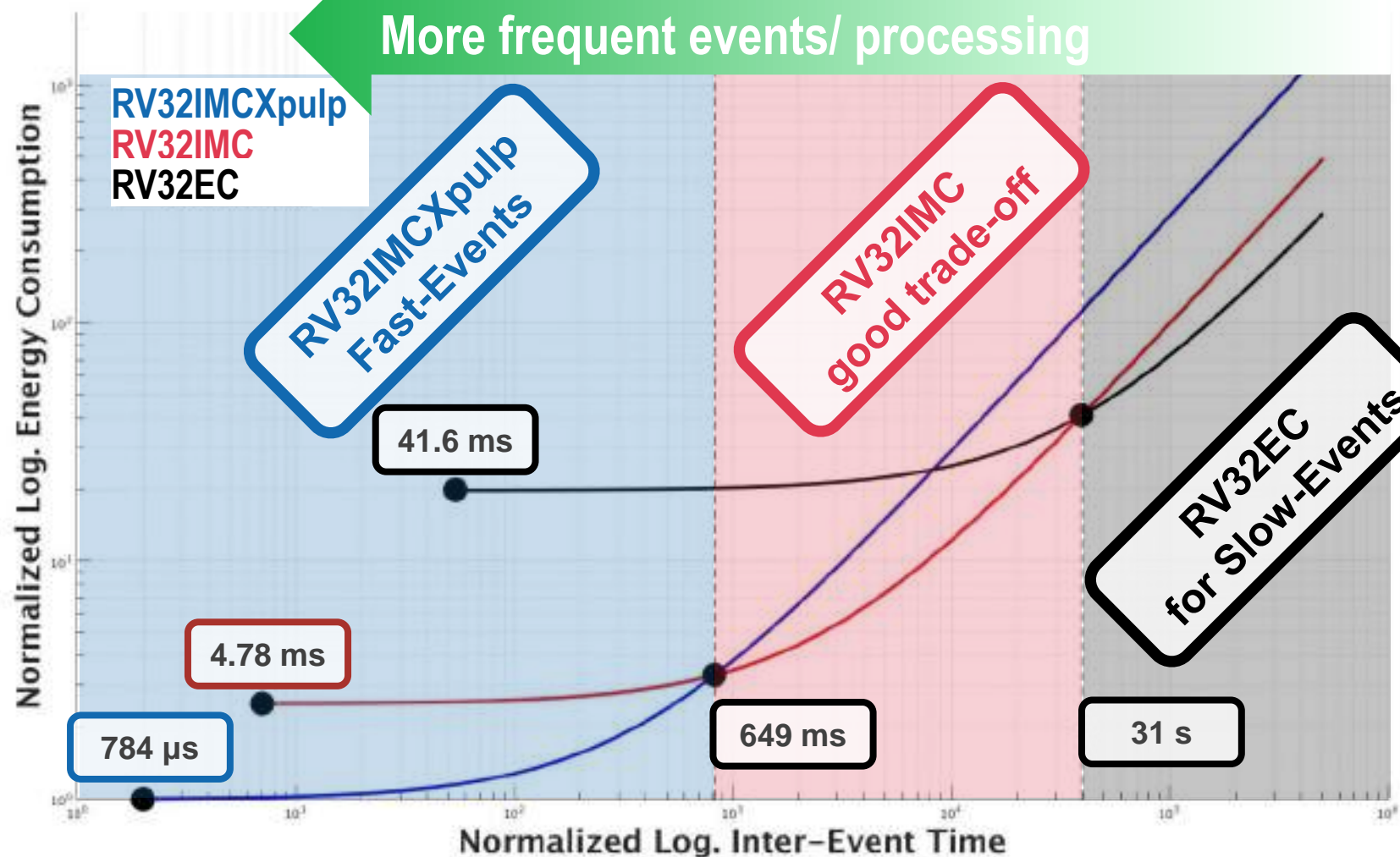
# Different cores for different area budgets



# 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