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

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>

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

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

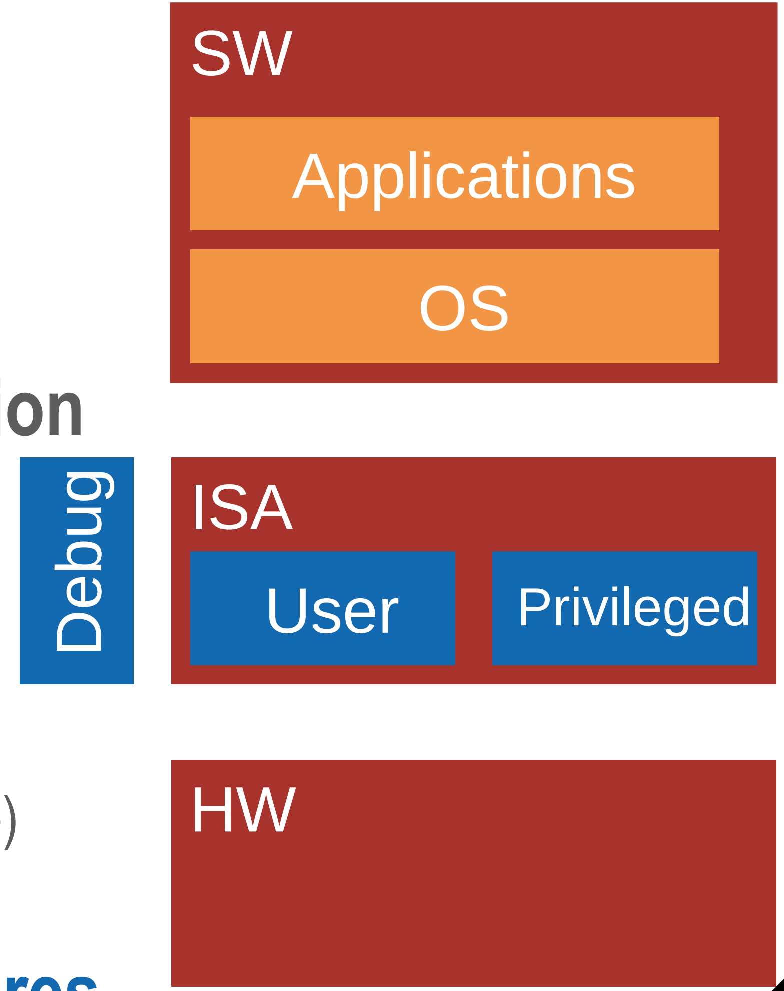
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Frank K. Gürkaynak
Senior Scientist

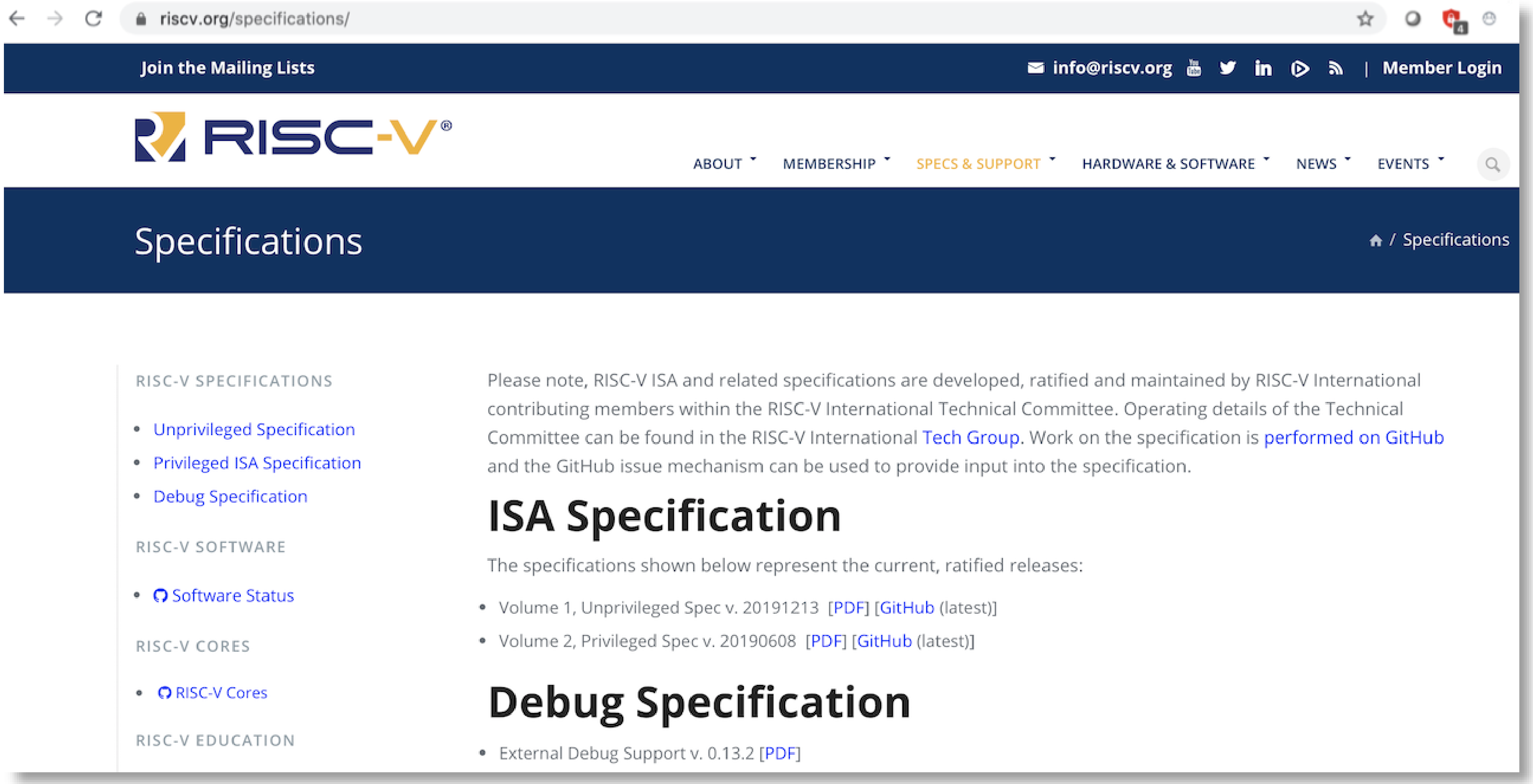
ETH Zürich
Integrated Systems Laboratory
Gloriastrasse 35
8092 Zürich, Switzerland
kgf@iis.ee.ethz.ch

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



riscv.org/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

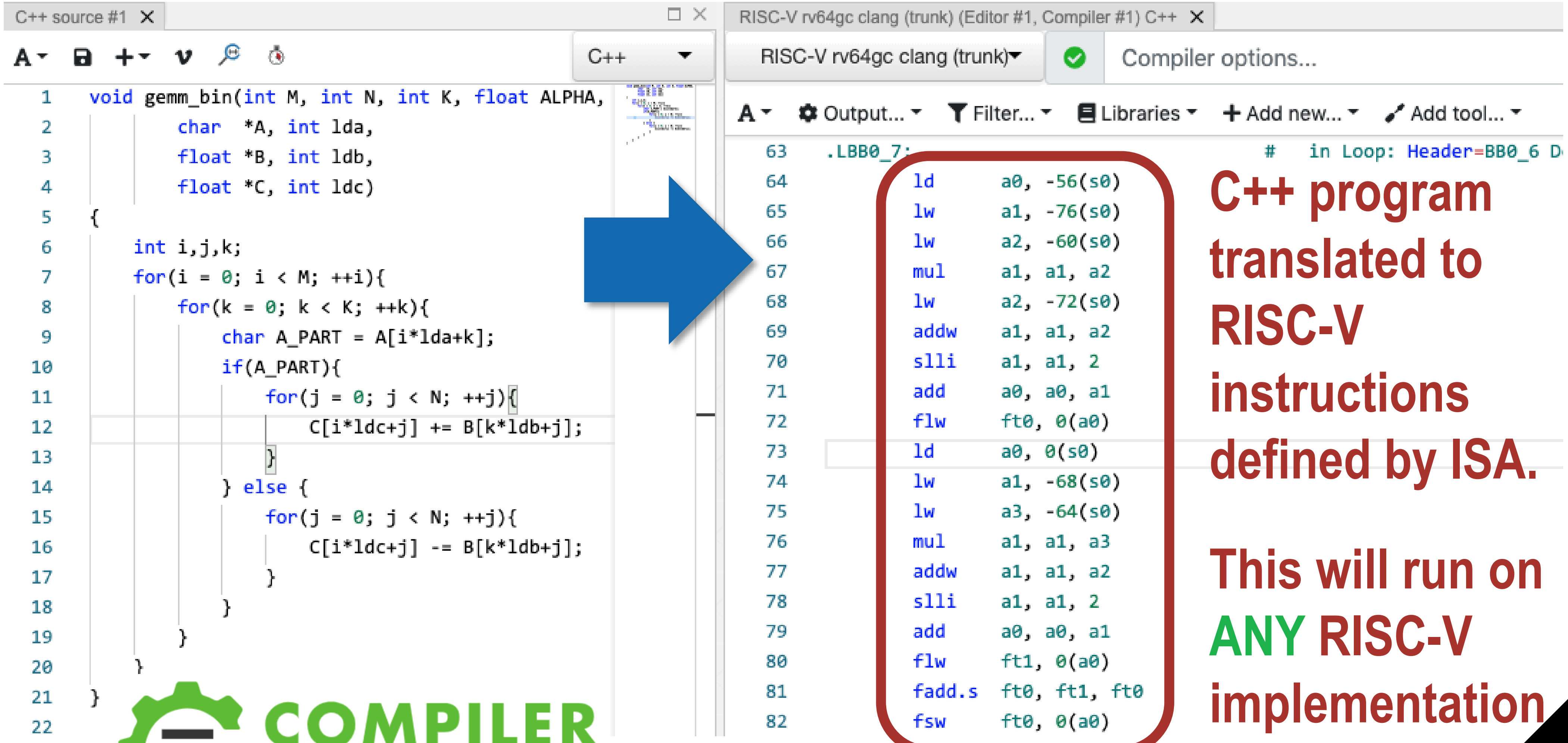
The specifications shown below represent the current, ratified releases:

- 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
C++
RISC-V rv64gc clang (trunk) (Editor #1, Compiler #1) C++
RISC-V rv64gc clang (trunk)
Compiler options...
Output...
Filter...
Libraries
Add new...
Add tool...

```cpp
void gemm_bin(int M, int N, int K, float ALPHA,
          char  *A, int lda,
          float *B, int ldb,
          float *C, int ldc)
{
    int i,j,k;
    for(i = 0; i < M; ++i){
        for(k = 0; k < K; ++k){
            char A_PART = A[i*lda+k];
            if(A_PART){
                for(j = 0; j < N; ++j){
                    C[i*ldc+j] += B[k*ldb+j];
                }
            } else {
                for(j = 0; j < N; ++j){
                    C[i*ldc+j] -= B[k*ldb+j];
                }
            }
        }
    }
}
```

```asm
.LBB0_7:                                # in Loop: Header=BB0_6 D
        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
        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

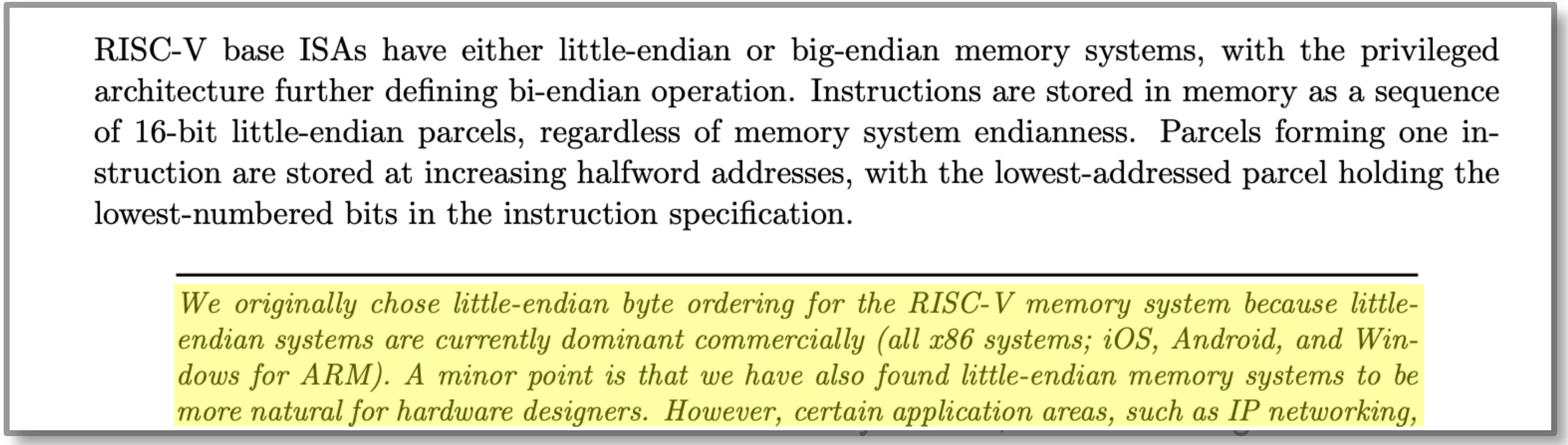
- **Kept very simple and extendable**
  - Wide range of applications from IoT to HPC
- **RV + word-width + extensions**
  - RV32IMC: 32bit, integer, multiplication, compressed
- **User specification:**
  - Separated into extensions, only I is mandatory
- **Privileged Specification (WIP):**
  - Governs 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

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.

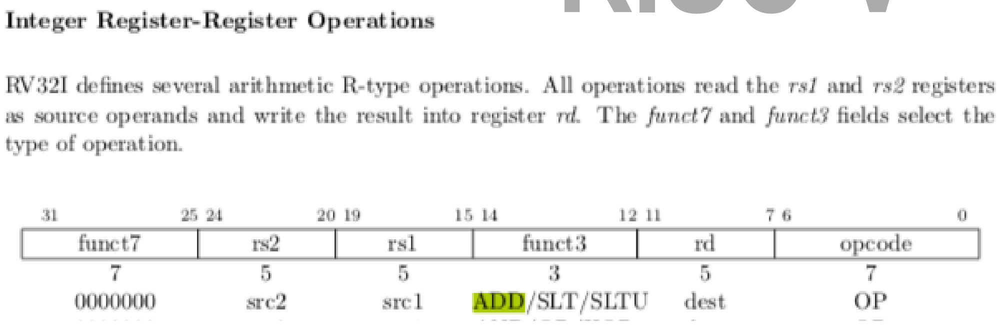
We originally chose little-endian byte ordering for the RISC-V memory system because little-endian systems are currently dominant commercially (all x86 systems; iOS, Android, and Windows for ARM). A minor point is that we have also found little-endian memory systems to be more natural for hardware designers.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

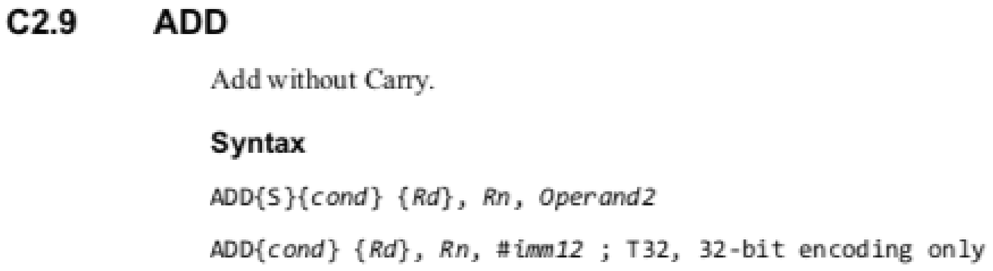


Integer Register-Register Operations

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
```

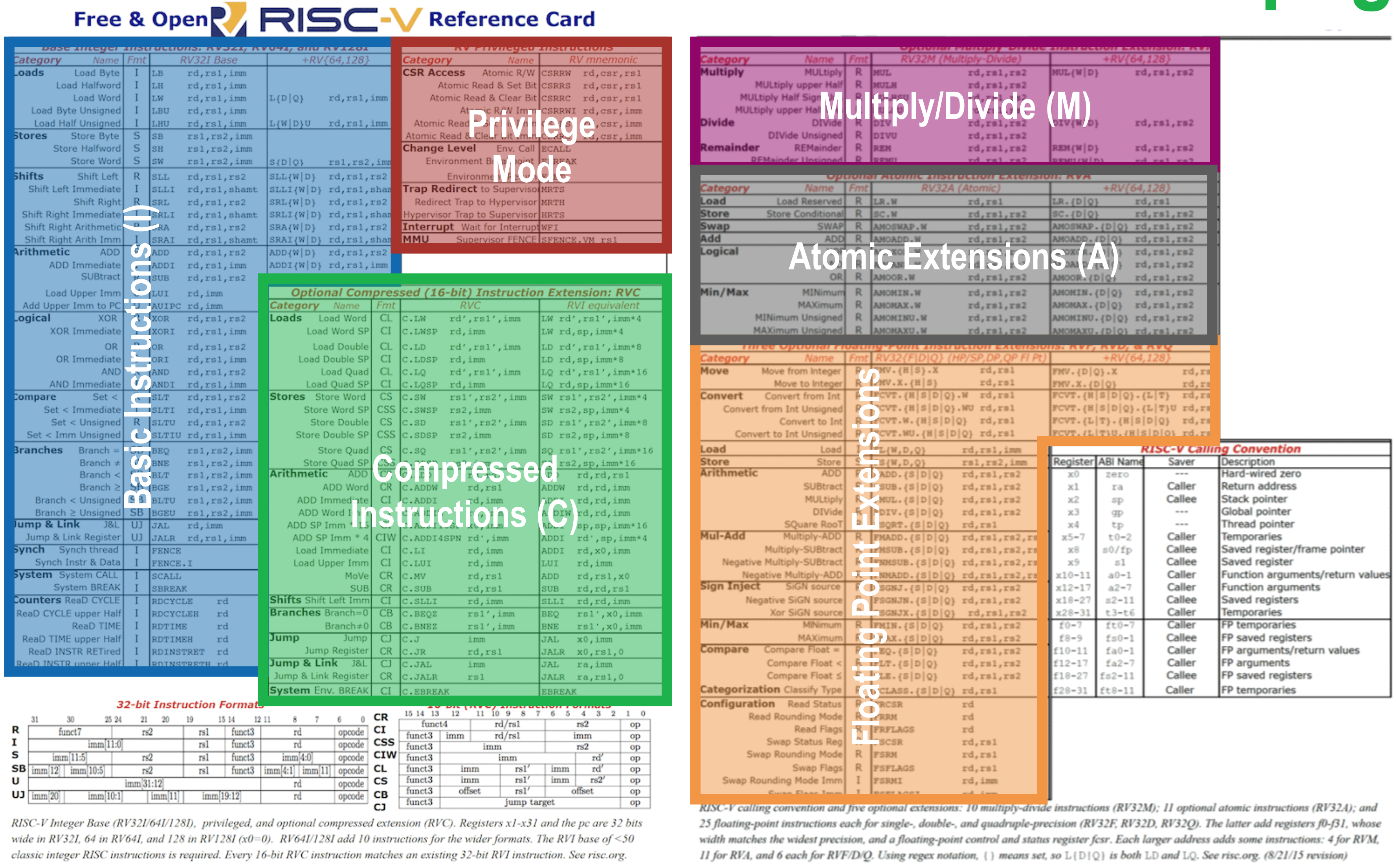
Are RISC-V processors better than XYZ?

- 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 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



Basic Instructions (I)

Privilege Mode

Compressed Instructions (C)

Multiply/Divide (M)

Atomic Extensions (A)

Floating Point Extensions

Free & Open RISC-V Reference Card

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base		+RV{64,128}	
Loads	Load Byte	I	LB	rd,rs1,imm		
	Load Halfword	I	LH	rd,rs1,imm		
	Load Word	I	LW	rd,rs1,imm	L{D|Q}	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm		
	Load Half Unsigned	I	LHU	rd,rs1,imm	L{W|D}U	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm		
	Store Halfword	S	SH	rs1,rs2,imm		
	Store Word	S	SW	rs1,rs2,imm	S{D|Q}	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2	SLL{W|D}	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt	SLLI{W|D}	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2	SRL{W|D}	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt	SRLI{W|D}	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2	SRA{W|D}	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt	SRAI{W|D}	rd,rs1,shamt
Arithmetic	ADD	[illegible]	ADD	rd,rs1,rs2	ADD{W|D}	rd,rs1,rs2
	ADD Immediate	[illegible]	ADDI	rd,rs1,imm	ADDI{W|D}	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2		
	Load Upper Imm	[illegible]	LUI	rd,imm		
	Add Upper Imm to PC	U	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	[illegible]	ORI	rd,rs1,imm		
	AND	R	AND	rd,rs1,rs2		
	AND Immediate	I	ANDI	rd,rs1,imm		
Compare	Set <	R	SLT	rd,rs1,rs2		
	Set < Immediate	[illegible]	SLTI	rd,rs1,imm		
	Set < Unsigned	R	SLTU	rd,rs1,rs2		
	Set < Imm Unsigned	[illegible]	SLTIU	rd,rs1,imm		
Branches	Branch =	SB	BEQ	rs1,rs2,imm		
	Branch ≠	[illegible]	BNE	rs1,rs2,imm		
	Branch <	SB	BLT	rs1,rs2,imm		
	Branch ≥	SB	BGE	rs1,rs2,imm		
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm		
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm		
Jump & Link	J&L	UJ	JAL	rd,imm		
	Jump & Link Register	UJ	JALR	rd,rs1,imm		
Synch	Synch thread	I	FENCE			
	Synch Instr & Data	I	FENCE.I			
System	System CALL	I	SCALL			
	System BREAK	I	SBREAK			
Counters	ReaD CYCLE	I	RDCYCLE	rd		
	ReaD CYCLE upper Half	I	RDCYCLEH	rd		
	ReaD TIME	I	RDTIME	rd		
	ReaD TIME upper Half	I	RDTIMEH	rd		
	ReaD INSTR RETired	I	RDINSTRET	rd		
	ReaD INSTR upper Half	I	RDINSTRETH	rd		

RV Privileged Instructions

Category	Name	RV mnemonic	
CSR Access	Atomic R/W	CSRRW	rd,csr,rs1
	Atomic Read & Set Bit	CSRRS	rd,csr,rs1
	Atomic Read & Clear Bit	CSRRC	rd,csr,rs1
	Atomic R/W Imm	CSRRWI	rd,csr,imm
	Atomic Read & Set Bit Imm	CSRRSI	rd,csr,imm
	Atomic Read & Clear Bit Imm	CSRRCI	rd,csr,imm
Change Level	Env. Call	ECALL	
	Environment Breakpoint	EBREAK	
	Environment Return	ERET	
Trap Redirect	to Supervisor	MRTS	
	Redirect Trap to Hypervisor	MRTH	
	Hypervisor Trap to Supervisor	HRTS	
Interrupt	Wait for Interrupt	WFI	
MMU	Supervisor FENCE	SFENCE.VM	rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC		RVI equivalent	
Loads	Load Word	CL	C.LW	rd′,rs1′,imm	LW	rd′,rs1′,imm*4
	Load Word SP	CI	C.LWSP	rd,imm	LW	rd,sp,imm*4
	Load Double	CL	C.LD	rd′,rs1′,imm	LD	rd′,rs1′,imm*8
	Load Double SP	CI	C.LDSP	rd,imm	LD	rd,sp,imm*8
	Load Quad	CL	C.LQ	rd′,rs1′,imm	LQ	rd′,rs1′,imm*16
	Load Quad SP	CI	C.LQSP	rd,imm	LQ	rd,sp,imm*16
Stores	Store Word	CS	C.SW	rs1′,rs2′,imm	SW	rs1′,rs2′,imm*4
	Store Word SP	CSS	C.SWSP	rs2,imm	SW	rs2,sp,imm*4
	Store Double	CS	C.SD	rs1′,rs2′,imm	SD	rs1′,rs2′,imm*8
	Store Double SP	CSS	C.SDSP	rs2,imm	SD	rs2,sp,imm*8
	Store Quad	CS	C.SQ	rs1′,rs2′,imm	SQ	rs1′,rs2′,imm*16
	Store Quad SP	CSS	C.SQSP	rs2,imm	SQ	rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD	rd,rs1	ADD	rd,rd,rs1
	ADD Word	CR	C.ADDW	rd,rs1	ADDW	rd,rd,imm
	ADD Immediate	CI	C.ADDI	rd,imm	ADDI	rd,rd,imm
	ADD Word Imm	CI	C.ADDIW	rd,imm	ADDIW	rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP	x0,imm	ADDI	sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN	rd′,imm	ADDI	rd′,sp,imm*4
	Load Immediate	CI	C.LI	rd,imm	ADDI	rd,x0,imm
	Load Upper Imm	CI	C.LUI	rd,imm	LUI	rd,imm
	MoVe	CR	C.MV	rd,rs1	ADD	rd,rs1,x0
	SUB	CR	C.SUB	rd,rs1	SUB	rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI	rd,imm	SLLI	rd,rd,imm
Branches	Branch=0	CB	C.BEQZ	rs1′,imm	BEQ	rs1′,x0,imm
	Branch≠0	CB	C.BNEZ	rs1′,imm	BNE	rs1′,x0,imm
Jump	Jump	CJ	C.J	imm	JAL	x0,imm
	Jump Register	CR	C.JR	rd,rs1	JALR	x0,rs1,0
Jump & Link	J&L	CJ	C.JAL	imm	JAL	ra,imm
	Jump & Link Register	CR	C.JALR	rs1	JALR	ra,rs1,0
System	Env. BREAK	CI	C.EBREAK		EBREAK	

32-bit Instruction Formats

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

16-bit (RVC) Instruction Formats

	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	funct4		rd/rs1		rs2		op
CI	funct3	imm	rd/rs1		imm		op
CSS	funct3	imm			rs2		op
CIW	funct3	imm				rd′	op
CL	funct3	imm		rs1′	imm	rd′	op
CS	funct3	imm		rs1′	imm	rs2′	op
CB	funct3	offset		rs1′	offset		op
CJ	funct3	jump target					op

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.

Optional Multiply-Divide Instruction Extension: RVM

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/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	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 Instruction Extension: RVA

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	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1
Load	Load	I	FL{W,D,Q}	rd,rs1,imm		
Store	Store	S	FS{W,D,Q}	rs1,rs2,imm		
Arithmetic	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		
	SQuare RooT	R	FSQRT.{S|D|Q}	rd,rs1		
Mul-Add	Multiply-ADD	R	FMADD.{S|D|Q}	rd,rs1,rs2,rs3		
	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 Inject	SiGN source	R	FSGNJ.{S|D|Q}	rd,rs1,rs2		
	Negative SiGN source	R	FSGNJN.{S|D|Q}	rd,rs1,rs2		
	Xor SiGN source	R	FSGNJX.{S|D|Q}	rd,rs1,rs2		
Min/Max	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		
Categorization	Classify Type	R	FCLASS.{S|D|Q}	rd,rs1		
Configuration	Read Status	R	FRCSR	rd		
	Read Rounding Mode	R	FRRM	rd		
	Read Flags	R	FRFLAGS	rd		
	Swap Status Reg	R	FSCSR	rd,rs1		
	Swap Rounding Mode	R	FSRM	rd,rs1		
	Swap Flags	R	FSFLAGS	rd,rs1		
	Swap Rounding Mode Imm	I	FSRMI	rd,imm		
	Swap Flags Imm	I	FSFLAGSI	rd,imm		

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	t0-2	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	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

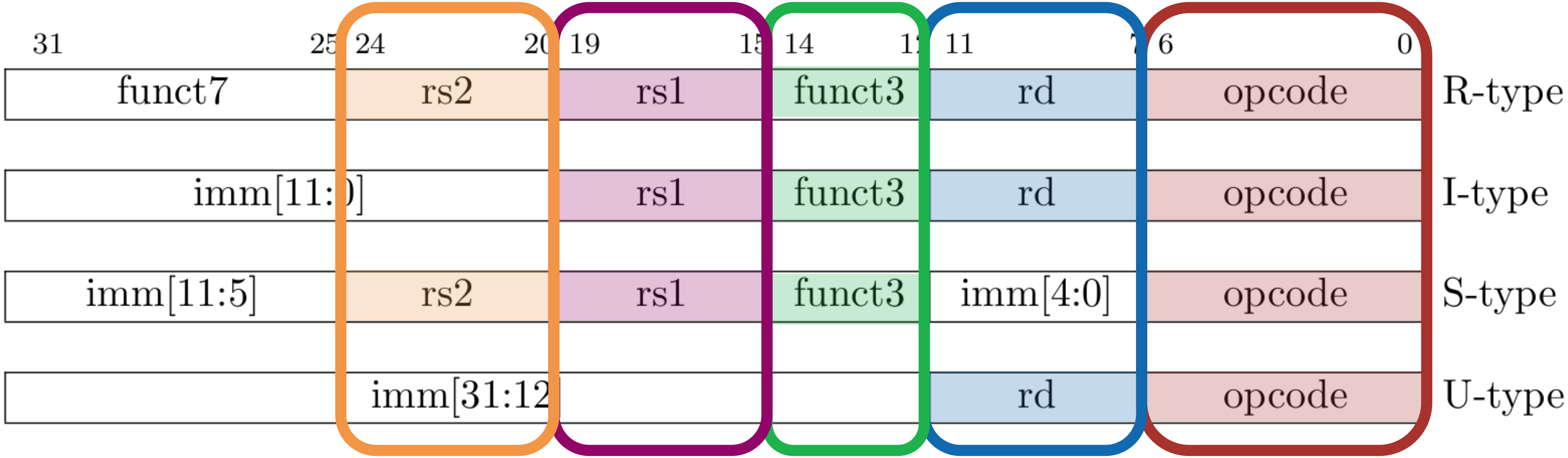
RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the 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

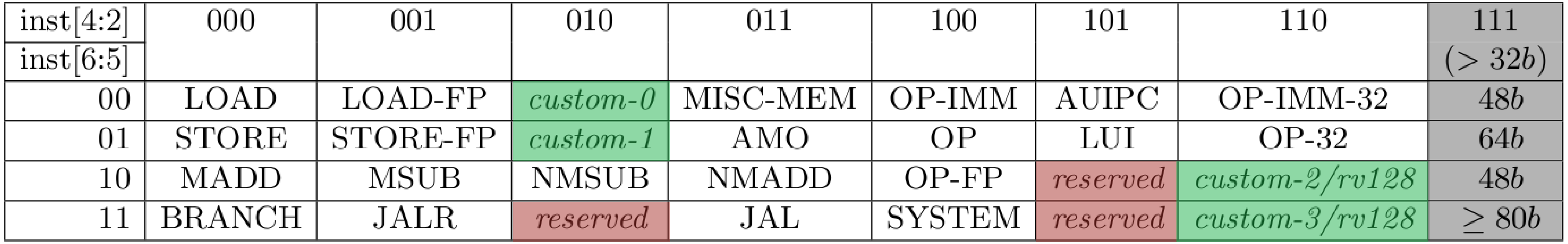
- **R** register to register operations
- **I** operations with immediate/constant values
- **S / SB** operations with two source registers
- **U / UJ** operations with large immediate/constant value



31 25	24 20	19 15	14 12	11 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[31:12]				rd	opcode	U-type

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]	000	001	010	011	100	101	110	111
inst[6:5]								$(> 32b)$
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*	$\geq 80b$

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

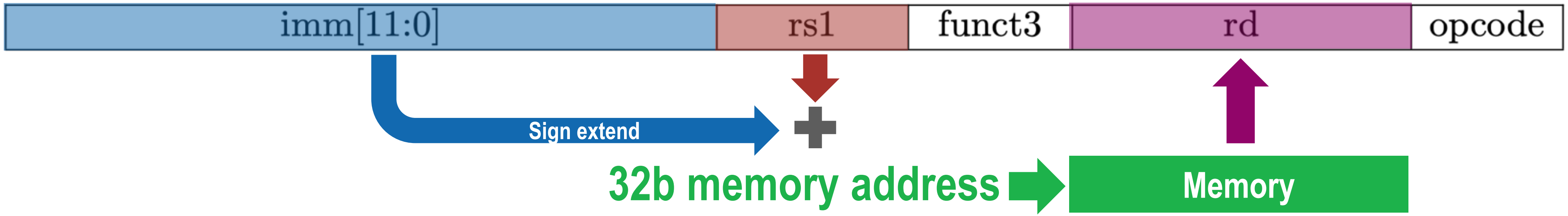
Constants (Immediates) in Instructions

- **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

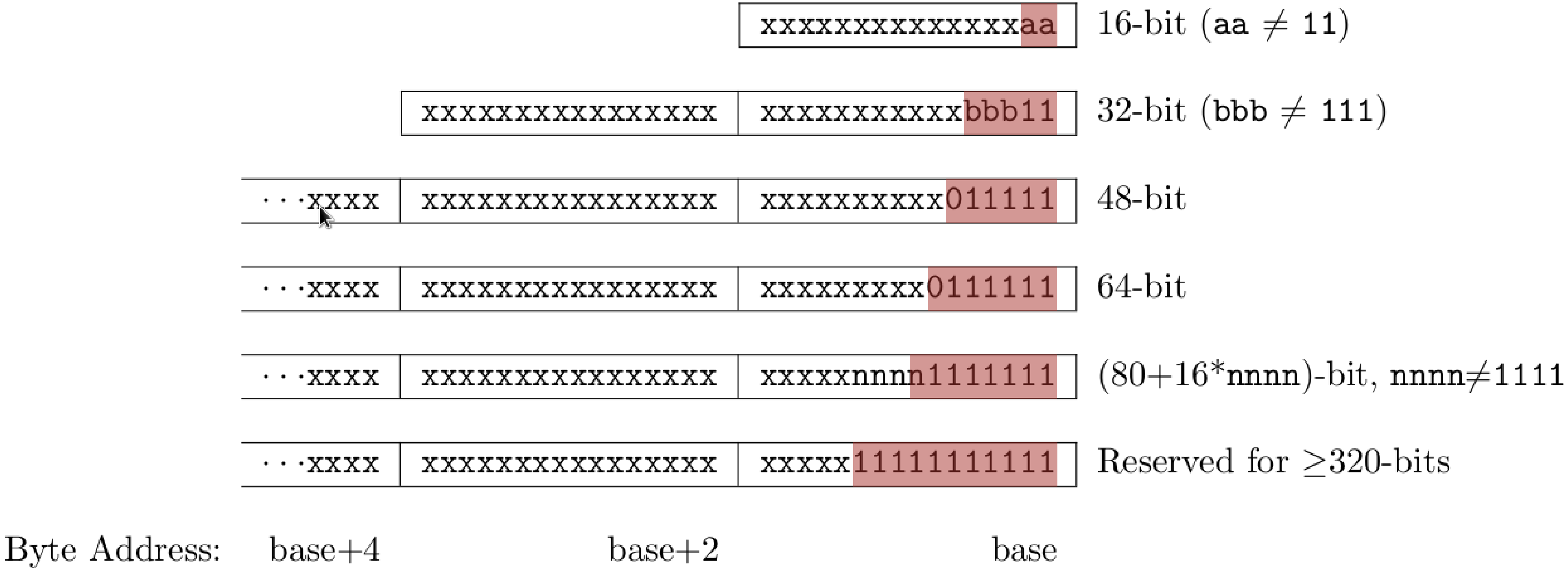
```
bne x10, x11, 2000 // if x10 != x11, jump 2000 ahead
```

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

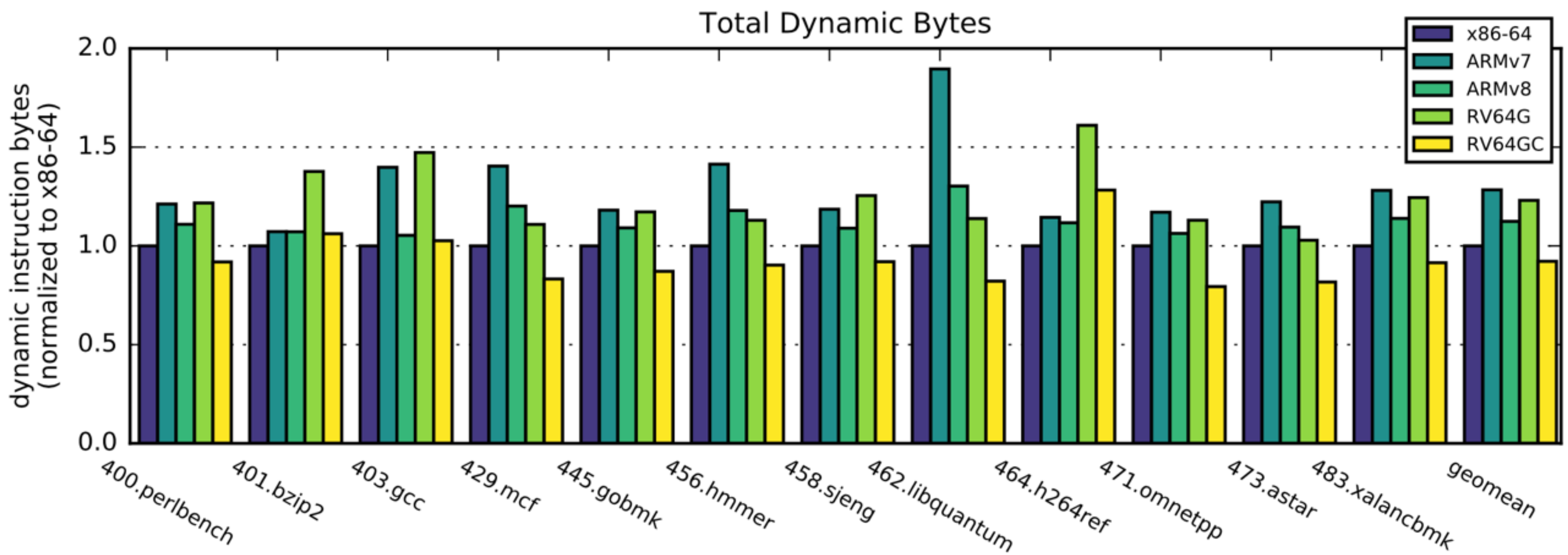
- LSB of the instruction tells how long the instruction is
- Supports instructions of 16, 32, 48, 64, 80, 96, … , 320 bit
  - Allows RISC-V to have Compressed instructions



Byte Address:	base+4	base+2	base	
			xxxxxxxxxxxxxxaa	16-bit (aa ≠ 11)
		xxxxxxxxxxxxxxxx	xxxxxxxxxxxbbb11	32-bit (bbb ≠ 111)
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxxx011111	48-bit
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxxxxx0111111	64-bit
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxxnnnn1111111	(80+16*nnnn)-bit, nnnn≠1111
	· · ·xxxx	xxxxxxxxxxxxxxxx	xxxxx11111111111	Reserved for ≥320-bits

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 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-CE	- **RI5CY** - RV32-ICMFX - SIMD - HW loops - Bit manipulation - Fixed point	- **Snitch** - RV32-ICMDFX	- **Ariane** - RV64-IC(MA) - Full privileged specification

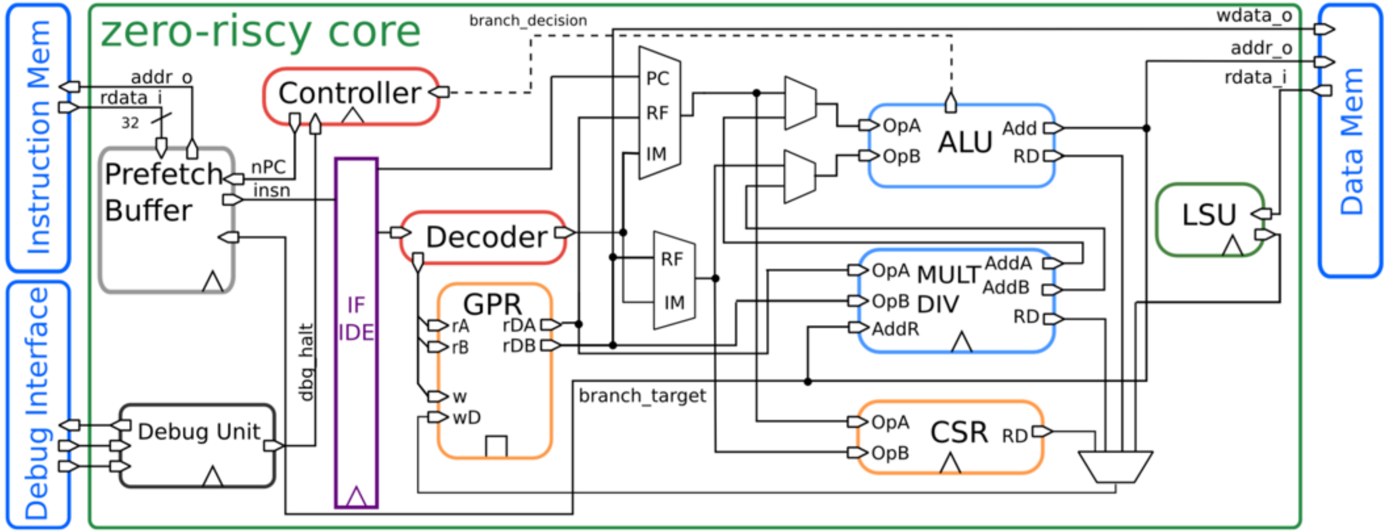
Ibex by LowRISC

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

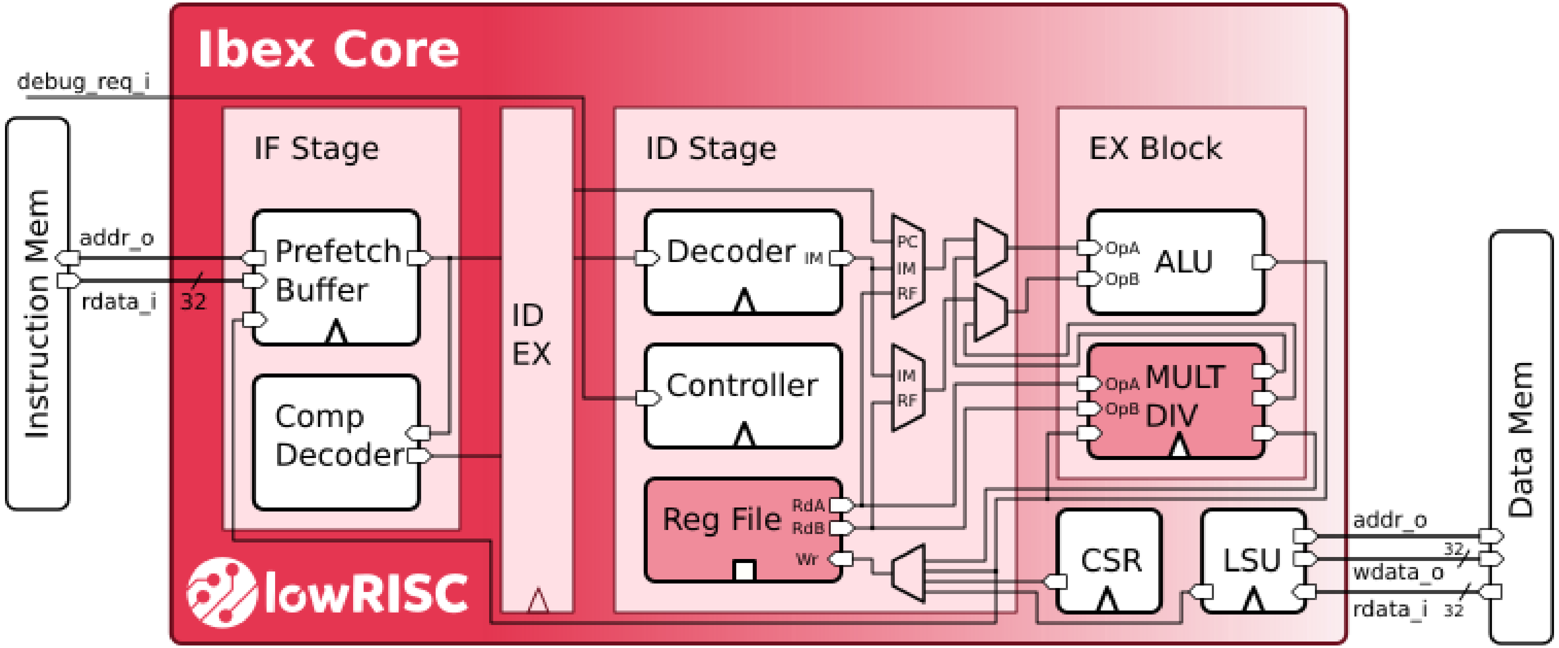


- **Two Configurations:**
  - **Zero-riscy**: RV32IMC (2,44 Coremark/MHz)
    - 32 registers, hardware multiplier
  - **Micro-riscy** : RV32EC (0,91 Coremark/MHz)
    - 16 registers (E), software emulated multiplier

Ibex continues to grow with LowRISC

40+	Contributors
680	Pull Requests
314	GitHub Issues

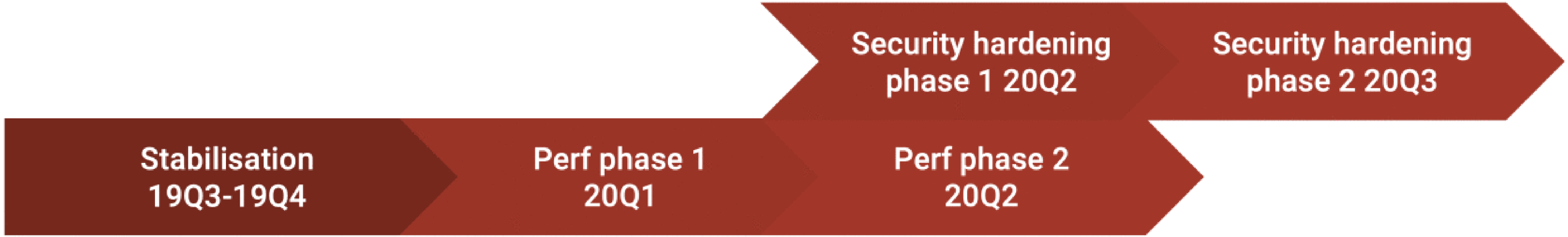
lowRISC



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



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

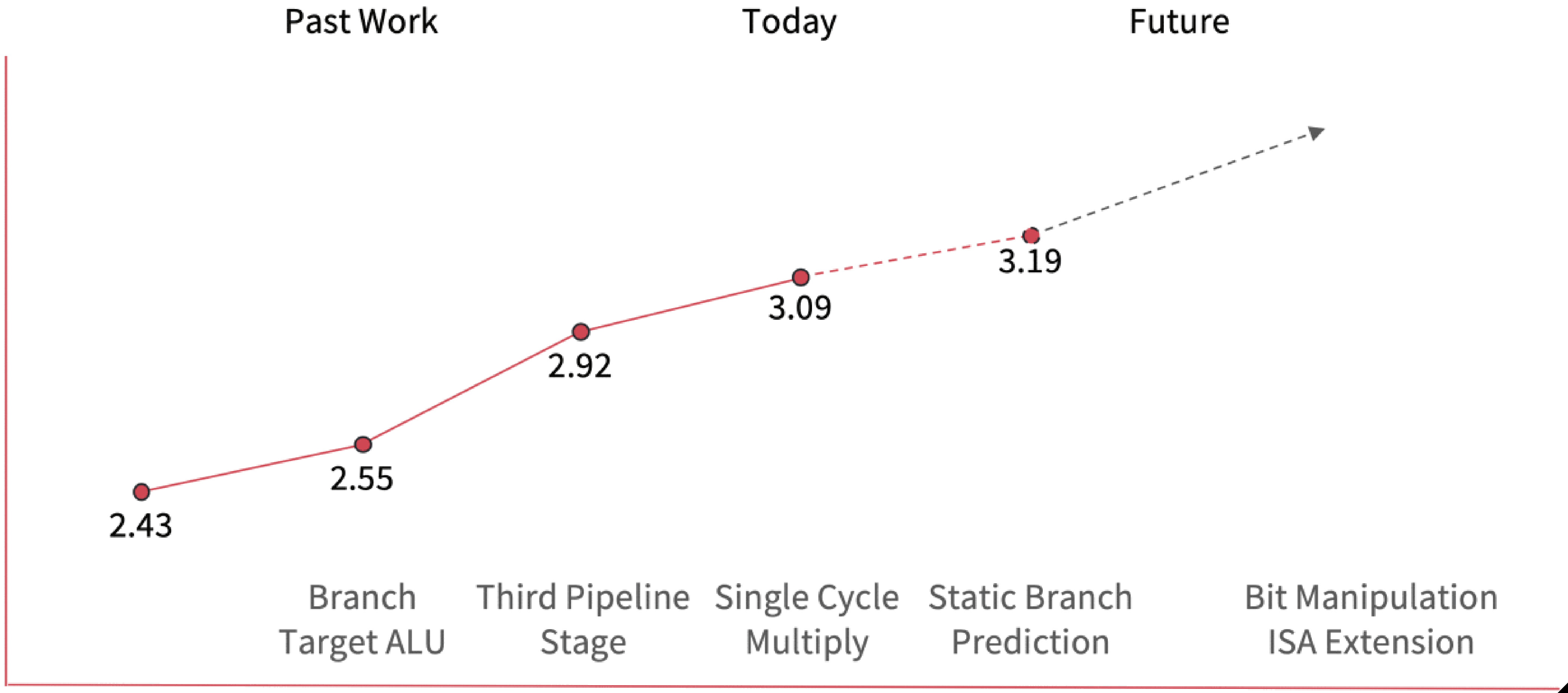
Security hardening phase 1 20Q2

- 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

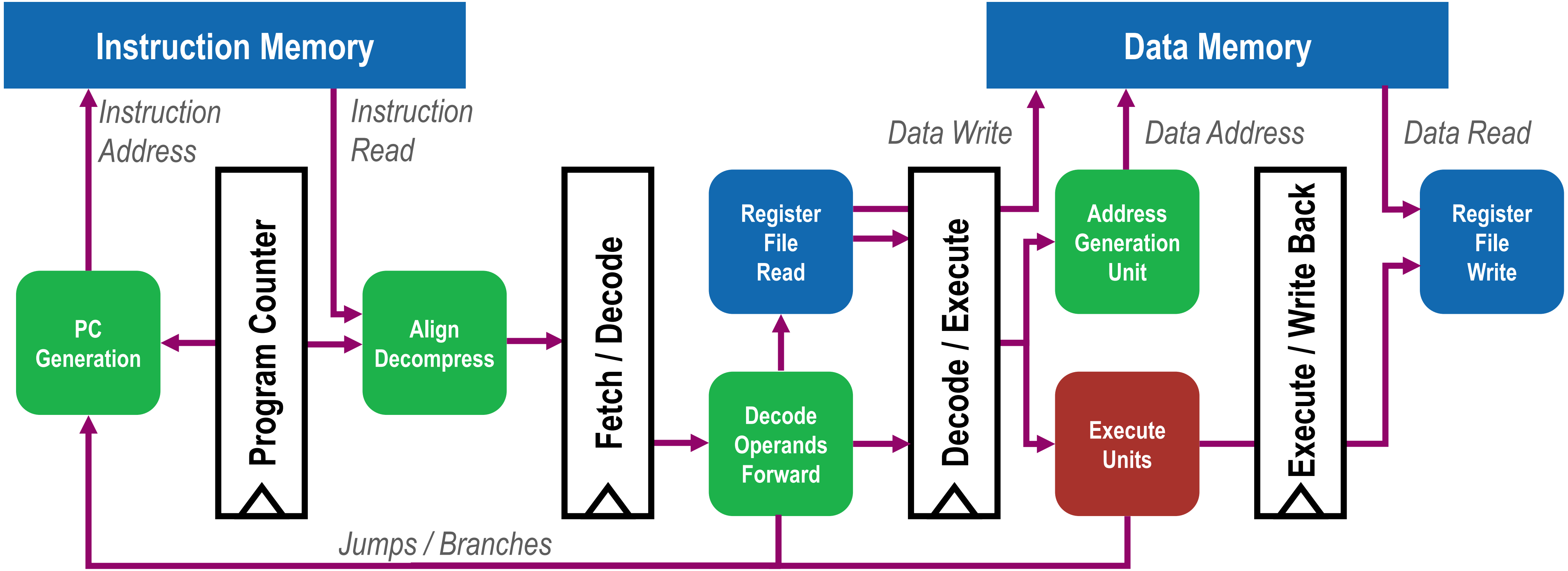
Growth of Ibex measured with Coremark/MHz



RI5CY / CV32E40P our main 32bit RISC-V core

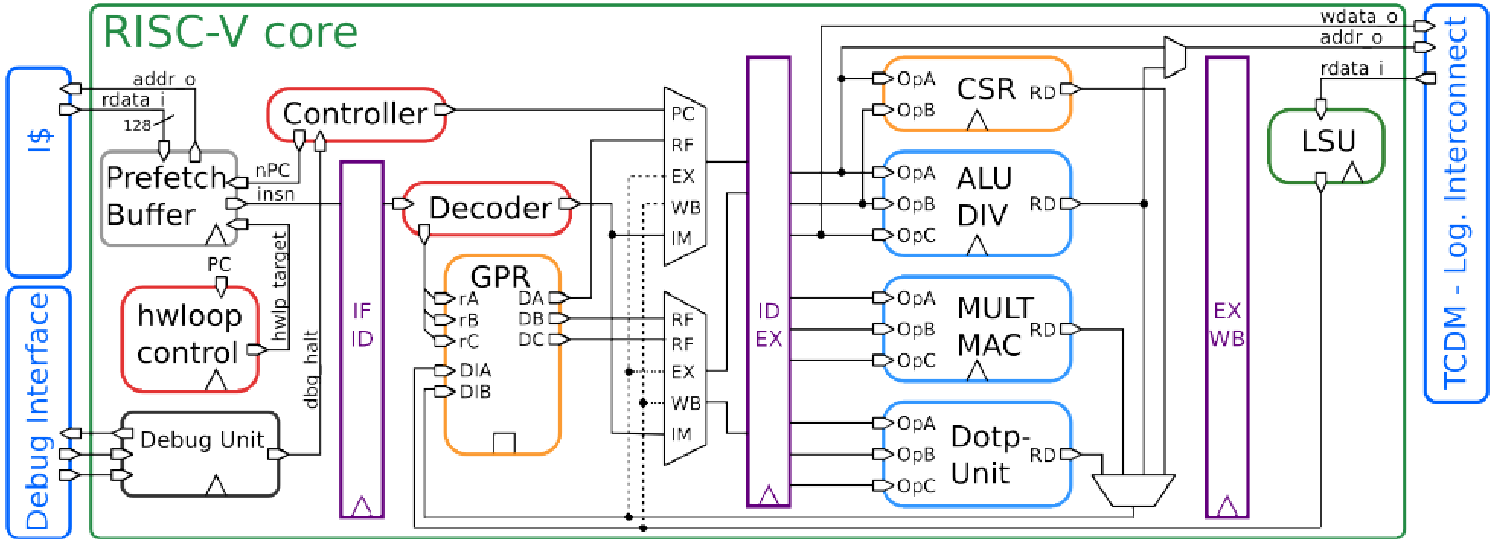
- Zero-riscy / Ibex is suitable for simple applications
  - Control applications, book-keeping
- For our research we need more capable cores
  - Mainly used in clusters for signal processing / machine learning applications
- Tuned for energy efficiency
  - Not necessarily low power
- Make use of custom 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 (`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!)
```

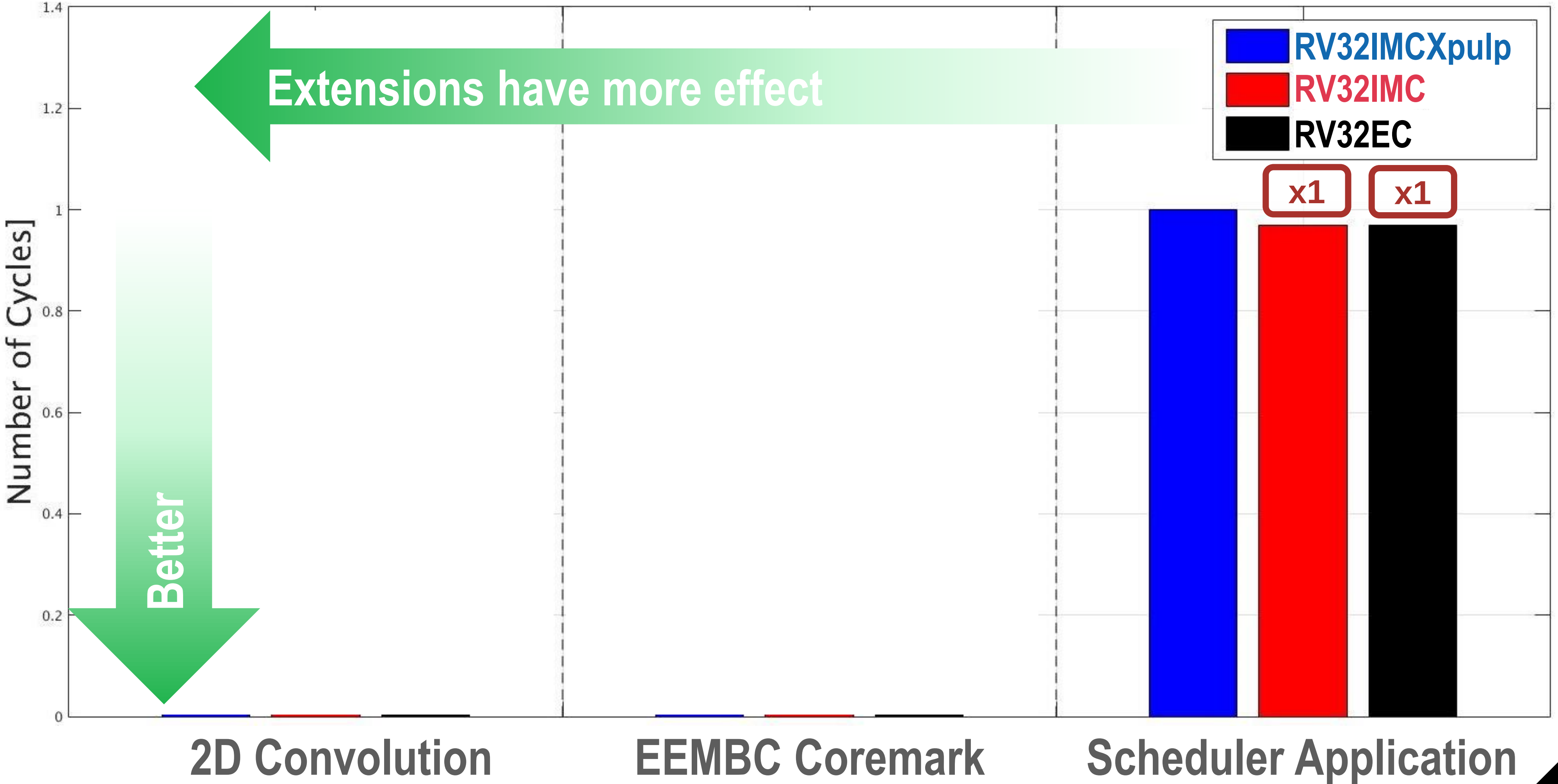
5 cycles/output

Packed-SIMD

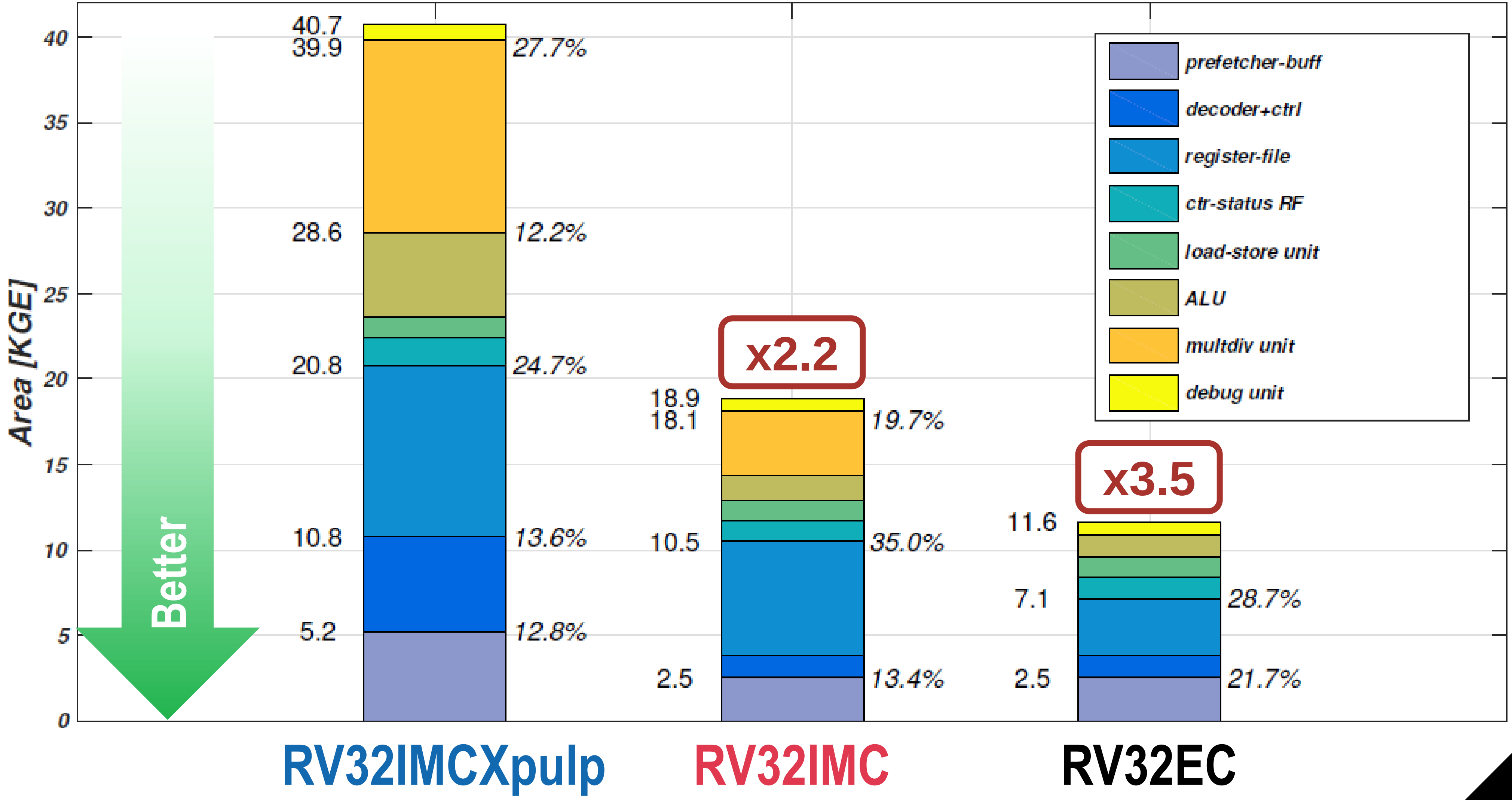
```
lp.setupi 25, Lend
   lw    x2, 0(x10!)
   lw    x3, 0(x11!)
   pv.add.b x2, x3, x2
Lend: sw x2, 0(x12!)
```

1,25 cycles/output

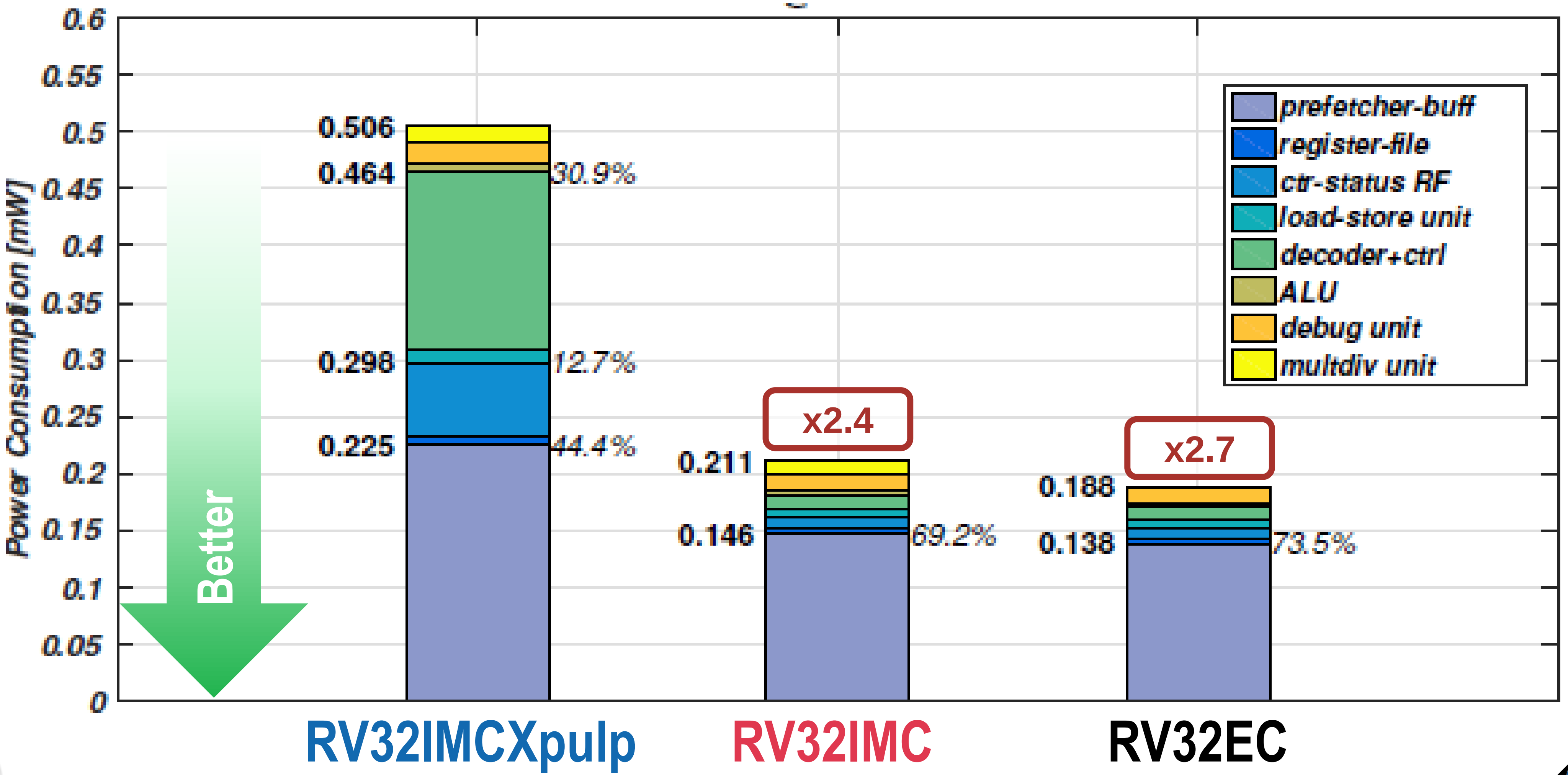
Runtime for three different applications



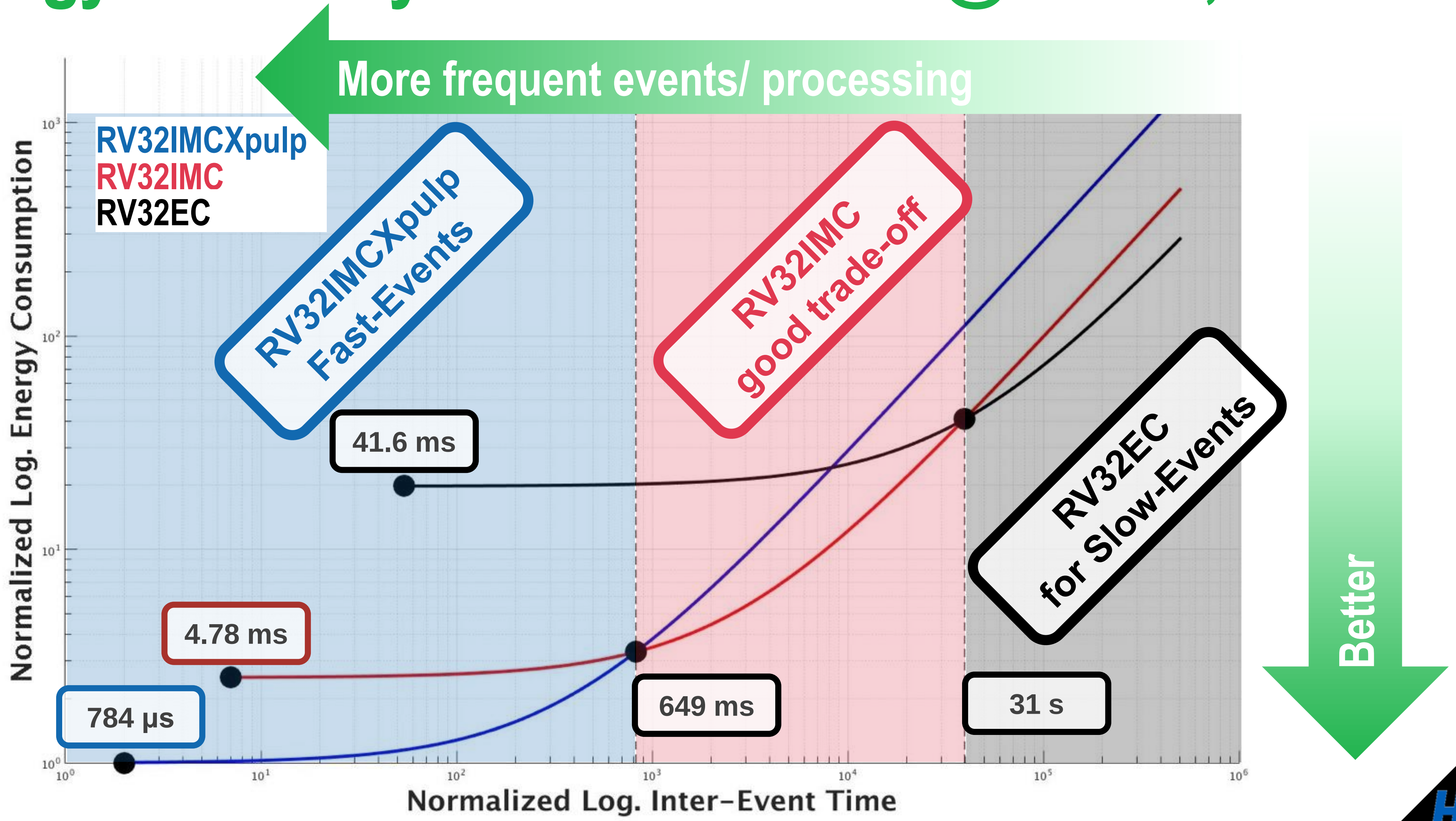
Different cores for different area budgets



Different cores for different power budgets



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



More frequent events/ processing

RV32IMCXpulp Fast-Events

RV32IMC good trade-off

RV32EC for Slow-Events

41.6 ms

4.78 ms

784 µs

649 ms

31 s

Better

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

PULP

Parallel Ultra Low Power

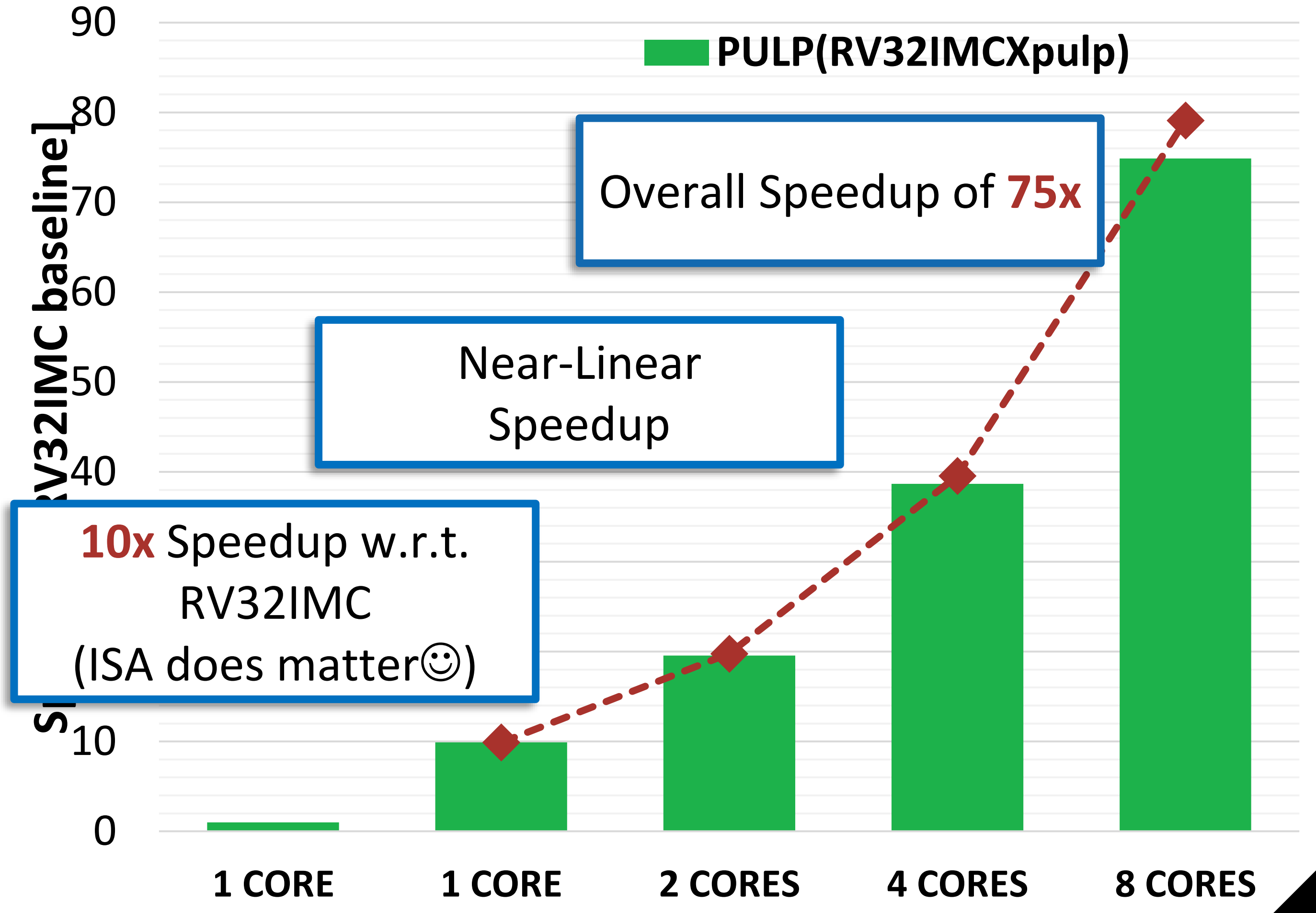
Luca Benini, Davide Rossi, Andrea Borghesi, Michele Magno, Simone Benatti, Francesco Conti, Francesco Beneventi, Daniele Palossi, Giuseppe Tagliavini, Antonio Pullini, Germain Haugou, Manuele Rusci, Florian Glaser, Fabio Montagna, Bjoern Forsberg, Pasquale Davide Schiavone, Alfio Di Mauro, Victor Javier 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

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



Overall Speedup of 75x

Near-Linear Speedup

10x Speedup w.r.t. RV32IMC (ISA does matter☺)