Ripasso concetti base: ISA RISC-V

Andrea Bartolini <a.bartolini@unibo.it>

(Architettura dei) Calcolatori Elettronici, 2023/2024

Instruction Count and CPI – The Iron Law of processor Performance

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

N°tot di cicli di CK = n° di istruzioni x n° cicli per istruzione

CPU time = n° istruzioni x CPI

Courtesy of Andrea Marongiu

Instruction Count and CPI – The Iron Law of processor Performance

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

Durata del clock

- **Instruction Count** for a program
  - Determined by program, ISA and compiler

- Average **cycles per instruction (CPI)**
  - Determined by CPU hardware
  - If different instructions have different *CPI*
    - Average CPI affected by instruction mix

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A$$

Courtesy of Andrea Marongiu

CPI Example

1 istruz → 2 clock

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \\ \text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}\end{aligned}$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A}$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

Courtesy of Andrea Marongiu

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

Courtesy of Andrea Marongiu

Instruction Set Architecture

CISC, RISC, RISC-V

In the Beginning…

People programmed in assembly and machine code!

- Needed as many addressing modes as possible
- Memory was (and still is) slow

CPUs had relatively few registers

- Register’s were more “expensive” than external mem
- Large number of registers requires many bits to index

Memories were small

- Encouraged highly encoded microcodes as instructions
- Variable length instructions, load/store, conditions, etc

Complex Instruction Set Computer (CISC)

The number of available registers greatly influenced the instruction set architecture (ISA)

Complex Instruction Set Computers were very complex

- Necessary to reduce the number of instructions required to fit a program into memory.
- However, also greatly increased the complexity of the ISA as well.

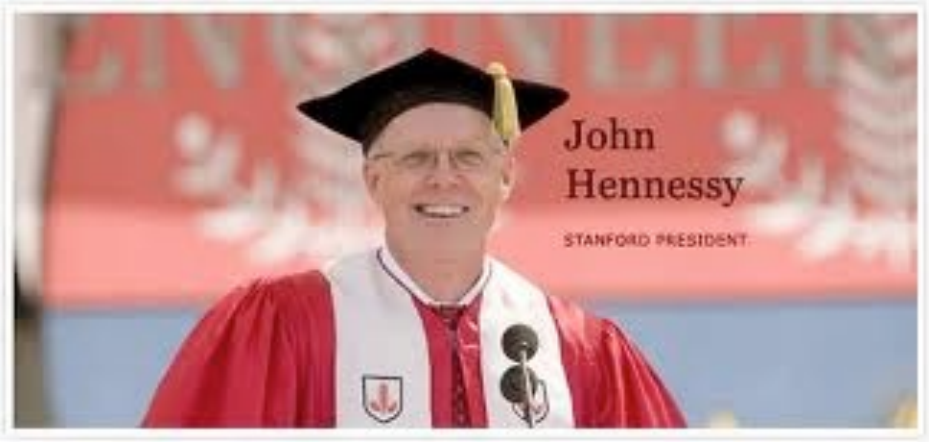
Reduced Instruction Set Computer (RISC)

Dave Patterson

- RISC Project, 1982
- UC Berkeley
- RISC-I: ½ transistors & 3x faster
- Influences: Sun SPARC, namesake of industry

John L. Hennessy

- MIPS, 1981
- Stanford
- Simple, *full* pipeline
- Influences: MIPS computer system, PlayStation, Nintendo



Reduced Instruction Set Computer (RISC)

RISC-V Design Principles

Simplicity favors regularity
- 32 bit instructions
- Same instruction format works at 16/32/64/128-bit data formats

Smaller is faster
- Small register file

Make the common case fast
- Include support for constants

Good design demands good compromises
- Support for different type of interpretations/classes

Developed at UC Berkeley as open ISA starting in 2010

Now managed by the RISC-V Foundation (riscv.org)
- Manages the standardization, trademark, compliancy check

Typical of many modern ISAs

Similar ISAs have a large share of embedded core market
Applications in consumer electronics, network/storage equipment, cameras, printers, ...

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

riscv.org

Why RISC-V?

RISC-V is an open ISA (instruction set architecture) enabling a new era of innovation for processor architectures. The RISC-V Foundation consists of more than 325 member companies. Here are the key benefits of the technology.

- **Software architects / firmware engineers / software developers**
  - RISC-V is much more than an open ISA, it is also a frozen ISA. The base instructions are frozen and optional extensions which have been approved are also frozen. Because of the stability of the ISA, software development can confidently be applied to RISC-V knowing that your investment will be preserved. Software written for RISC-V will run on all similar RISC-V cores forever. The frozen ISA provides a solid foundation that software managers can depend on to preserve their software investments. Because the RISC-V ISA is open, this translates to hardware engineers having more flexibility over the processor implementation. With this power, software architects can become more influential in the final hardware implementation. They can provide input to hardware designers to make the RISC-V core more software centric.
- **CTOs / Chip designers / System Architects**
  - Innovation is the key enabler of RISC-V. Because the ISA is open, it is the equivalent of everyone having a micro architecture license. One can optimize designs for lower power, performance, security, etc. while keeping full compatibility with other designs. Because there is significantly more control over the hardware implementation, all technical recipients of the architecture can make suggestions at a much earlier point than previously was possible. The result is a solution with significantly fewer compromises. RISC-V also supports custom instructions for designs which require particular acceleration or specialty functions.
- **Board designers**
  - In addition to the frozen ISA benefits, RISC-V's open ISA can provide several additional benefits. For example, if engineers are implementing a soft RISC-V core in an FPGA, often the RTL source code is available. Since RISC-V is royalty free this creates significant flexibility to port a RISC-V based design from an FPGA to an ASIC or another FPGA without any software modifications. Designers who are concerned with security from a trust perspective will also appreciate RISC-V. When the RTL source code is available, this enables deep inspection. With the ability to inspect the RTL, one can establish trust.

Courtesy of Andrea Marongiu

RISC-V vs CISC

x86 = Complex Instruction Set Computer (CISC)

- > 1000 instructions, 1 to 15 bytes each
- operands in dedicated registers, general purpose registers, memory, on stack, …
  - can be 1, 2, 4, 8 bytes, signed or unsigned
- 10s of addressing modes
  e.g. Mem[segment + reg + reg*scale + offset]

RISC-V = Reduced Instruction Set Computer (RISC)

- ≈ 200 instructions, 32 bits each, 4 formats
- all operands in registers
  - almost all are 32 bits each
- ≈ 1 addressing mode:
  Mem[reg + imm]

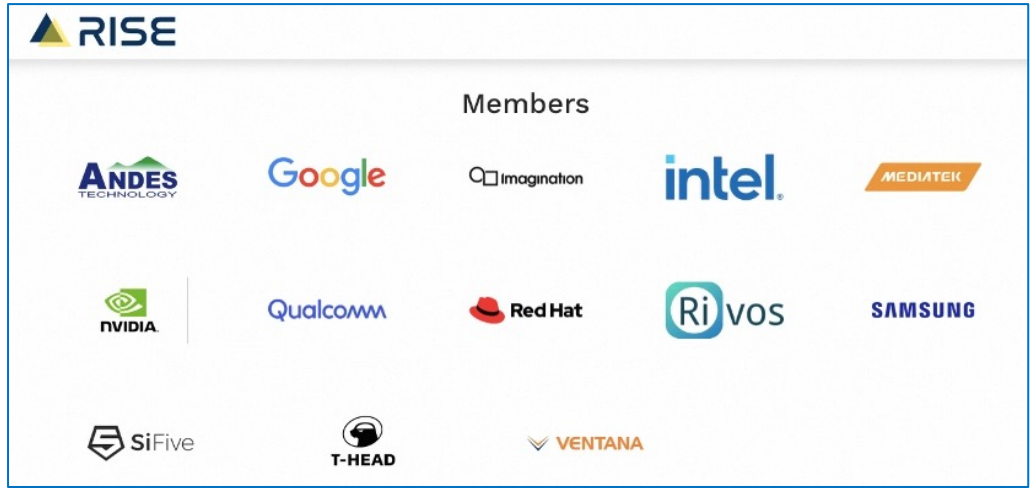
CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per instruction	Low cycles per istruction, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

Why RISC-V

2014 – A cute, open ISA

2023 – Disruptive Force

2023 RISC-V International more than 26% membership growth year-over-year, with over 3,180 members across 70 countries. More than 10 billion RISC-V cores in the market, 10K+ engineers working on RISC-V



Situation in 2021 …

RISC-V® Dashboard

Recently Ratified Extensions

Creato da Jeff Scheel, ultima modifica il mag 26, 2023

If you are looking for documentation on a recently ratified extension that has not yet been merged into the published specifications listed on the RISC-V Specifications page, check the table below. These extensions are completely ratified by RISC-V and will be merged into the final specifications in the coming months.

Specification name	Ratification date	New extension(s) or Profile(s)
"Zihintntl" Non-Temporal Locality Hints	May 2023	Zihintntl
RISC-V Code Size Reduction	April 2023	Zca, Zcb, Zcd, Zce, Zcf, Zcmp, Zcmt
RISC-V Profiles	March 2023	RVA20, RVI20, RVA22
"Zicntr" and "Zihpm" Counters	March 2023	Zicntr, Zihpm
RV32E and RV64E Base Integer Instruction Sets	January 2023	RV32E/RV64E
RISC-V "stimecmp / vstimecmp" Extension	November 2021	Sstc
RISC-V Vector Extension	November 2021	Zve32x, Zve32f, Zve64x, Zve64f, Zve64d, Zve, Zvl32b, Zvl64b, Zvl128b, Zvl256b, Zvl512b, Zvl1024b, Zvl, Zv
The RISC-V Instruction Set Manual Volume II: Privileged Architecture	November 2021	Sm1p12, Ss1p12, Sv57, Hypervisor, Svinval, Svnapot, Svpbmt
"Zfh" and "Zfhmin" Standard Extensions for Half-Precision Floating-Point	November 2021	Zfh, Zfhmin
"Zfinx", "Zdinx", "Zhinx", "Zhinxmin": Standard Extensions for Floating-Point in Integer Registers	November 2021	Zfinx, Zdinx, Zhinx, Zhinxmin
"Zihintpause" Pause Hint	February 2021	Zihintpause

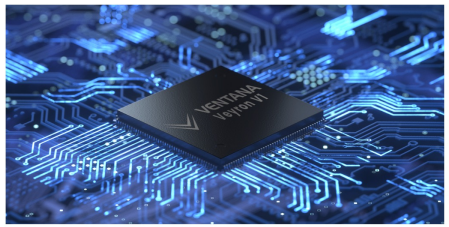
Situation today

VENTANA

PRODUCTS NEWS & INSIGHTS COM

PRESS RELEASES

Ventana Introduces Veyron, World's First Data Center Class RISC-V CPU Product Family



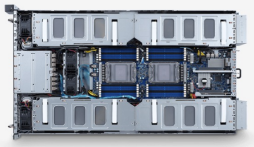
VENTANA PRODUCTS

Veyron V1 Features

- Eight wide, aggressive out-of-order pipeline
- 3.6GHz
- 5nm process technology
- 16 cores per cluster
- High core count multi-cluster scalability up to 128 cores
- 48MB of shared L3 cache per cluster
- Advanced side channel attack mitigations
- Comprehensive RAS features
- Top-down performance tuning methodology
- Provided with IOMMU and Advanced Interrupt Architecture (AIA) system IP
- SDK released with necessary software already ported to Veyron
- Veyron V1 Development Platform available

esperanto.ai

Access up to 16,000 RISC-V processors for exceptional compute efficiency.



SEE LESS

The Esperanto AI /HPC compute server features up to 16,000 RISC-V processors.

Features:

- 8 or 16 ET-SoC-1 PCIe cards, each with more than 1,000 RISC-V compute cores
- 2 Intel Xeon® Gold 6326 16-core or Xeon Platinum 8358P 32-core host processors
- Up to 1 Terabyte of DDR4-3200 main memory
- Esperanto Machine Learning and General Purpose Software Development Kits
- Dozens of Generative AI, Transformer, Computer Vision and Recommendation System AI Models
- Standard 2U 19" rack-mounted chassis for the datacenter environment

RISE (RISC-V Software Ecosystem

https://riseproject.dev

Mission

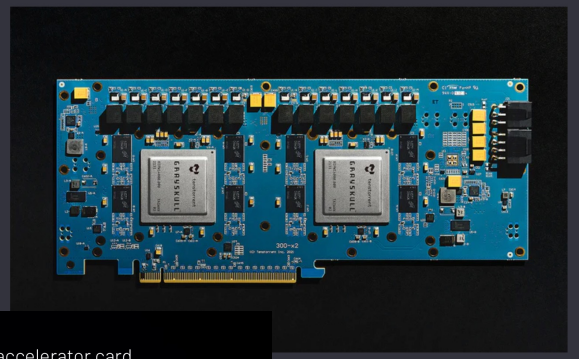
- Accelerate the development of open source software for RISC-V
- Raise the quality of RISC-V Platform implementations
- Push the RISC-V Software ecosystem forward and align ecosystem partners' efforts

Board Members

Google intel Qualcomm MEDIATEK ANDES TECHNOLOGY T-HEAD Red Hat

RiVOS SiFive VENTANA SAMSUNG NVIDIA Imagination

Fast



Grayskull™ e300

3/4 Length PCIe Card

+ Description

− Technical Specs

- 300W
- PCIe Gen 4, 16 lanes
- DRAM 16 GB, 200 GigaByte/sec
- Silicon TOPS 600
- Software enabled TOPS 1700

+ System Requirements

PCIe AI Edge accelerator card
Powered by 1 Metis AIPU

For developers seeking more performance, Axelera AI has created a range of PCIe accelerators. Powered by one Metis AIPU, this card can deliver up to 214 TOPS, enabling it to handle the most demanding vision applications. This performance is enhanced through the exceptional developer experience offered by the Voyager SDK software stack to accelerate application deployment. The PCIe card is priced at 199 EUR, setting a new benchmark for price/performance.

Available for ordering in early 2023.

Up to 214 TOPS | Up to 3.200 FPS for ResNet-50 | From 199 EUR

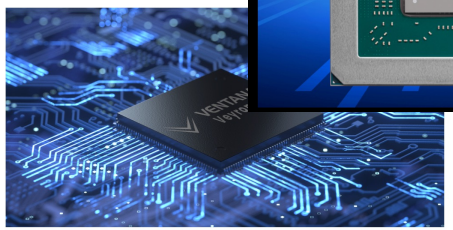
Situation tomorrow

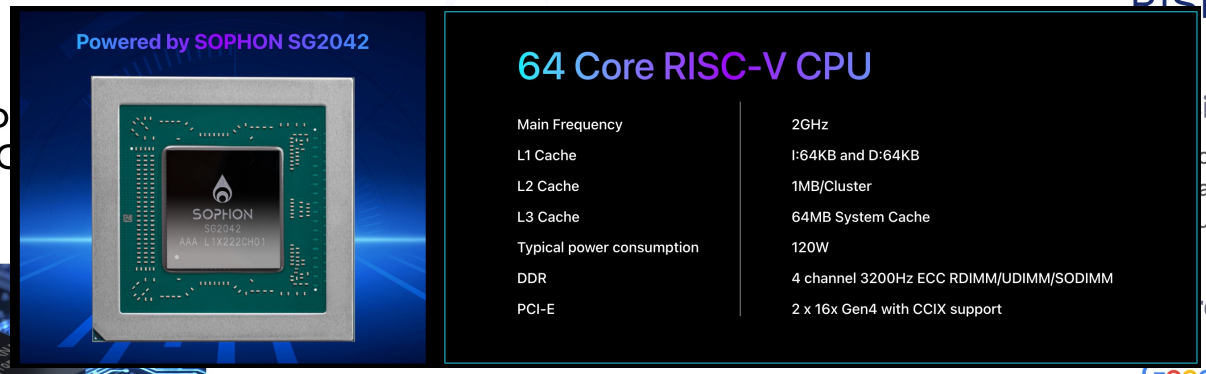
VENTANA

PRODUCTS NEWS & INSIGHTS COM

PRESS RELEASES

Ventana Intro
Data Center C
Family





64 Core RISC-V CPU

Main Frequency	2GHz
L1 Cache	I:64KB and D:64KB
L2 Cache	1MB/Cluster
L3 Cache	64MB System Cache
Typical power consumption	120W
DDR	4 channel 3200Hz ECC RDIMM/UDIMM/SODIMM
PCI-E	2 x 16x Gen4 with CCIX support

- 5nm process technology
- 16 cores per cluster
- High core count multi-cluster scalability up to 128 cores
- 48MB of shared L3 cache per cluster
- Advanced side channel attack mitigations
- Comprehensive RAS features
- Top-down performance tuning methodology
- Provided with IOMMU and Advanced Interrupt Architecture (AIA) system IP
- SDK released with necessary software already ported to Veyron V1 Development Platform available

RISE (RISC-V Software Ecosystem

https://riseproject.dev

ion

ccelerate the development of open source software for RISC-V
aise the quality of RISC-V Platform implementations
ush the RISC-V Software ecosystem forward and align ecosystem partners' efforts

d Me

Google

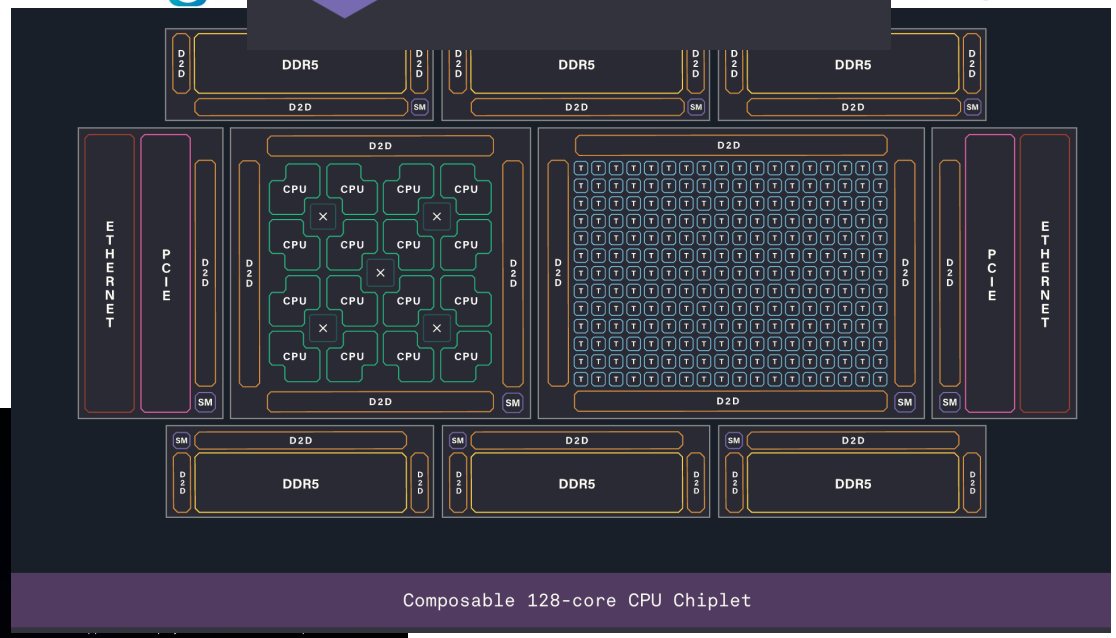
HEAD

Red Hat

Rivos

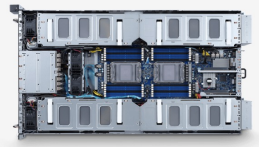
Imagination

tenstorrent



esperanto.ai

Access up to 16,000 RISC-V processors for exceptional compute efficiency.



SEE LESS

The Esperanto AI /HPC compute server features up to 16,000 RISC-V processors.

Features:

- 8 or 16 ET-SoC-1 PCIe cards, each with more than 1,000 RISC-V compute cores
- 2 Intel Xeon® Gold 6326 16-core or Xeon Platinum 8358P 32-core host processors
- Up to 1 Terabyte of DDR4-3200 main memory
- Esperanto Machine Learning and General Purpose Software Development Kits
- Dozens of Generative AI, Transformer, Computer Vision and Recommendation System AI Models
- Standard 2U 19" rack-mounted chassis for the datacenter environment

setting a new benchmark for price/performance.

Available for ordering in early 2023.

Up to 214 TOPS | Up to 3.200 FPS for ResNet-50 | From 199 EUR

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
add a, b, c // a gets b + c
```

- All arithmetic operations have this form

Design Principle 1: **Simplicity favours regularity**
- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

C code:

```
f = (g + h) - (i + j);
```

- f, ..., j in x19, x20, ..., x23

Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Courtesy of Andrea Marongiu

Non si hanno registri specializzati come su Lebarch. Ctipo stack pointer tranne il registro zero in cui è salvato il valore zero e non è modificabile.

Register Operands

› Arithmetic instructions use register operands

32 registri da 64 bit

› RISC-V has a 32 × 64-bit register file
  – Use for frequently accessed data
  – 64-bit data is called a "doubleword"
    › 32 x 64-bit general purpose registers x0 to x31
  – 32-bit data is called a "word"

Design Principle 2: **Smaller is faster**
- c.f. main memory: millions of locations

Dal punto di vista del compilatore ci sono però delle convenzioni x gli indirizzi:

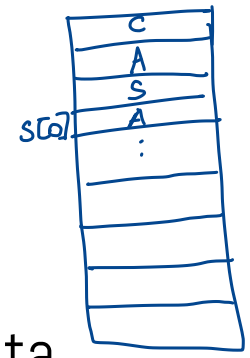
- **x0**: the constant value 0
- **x1**: return address
- **x2**: stack pointer
- **x3**: global pointer
- **x4**: thread pointer
- **x5** – x7, x28 – x31: temporaries
- **x8**: frame pointer
- **x9**, **x18** – **x27**: saved registers
- **x10** – **x11**: function arguments/results
- **x12** – **x17**: function arguments

↓ associati al campo return.

Courtesy of Andrea Marongiu

ISTRUZIONI DI ACCESSO ALLA MEMORIA : si legge / scrive un dato in memoria con load / store.

Memory Operands



In ciascuna cella di memoria c'è 1 byte, ovvero 8 bit.

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian — byte meno significativo all'indirizzo + basso — es. S[4] = casa
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address

C code: — A contiene delle double word

```
A[12] = h + A[8];
```

(8 → 64) — h è in x21 — A è in x22

h in x21, base address of A in x22

A[0] : X22

Compiled RISC-V code:

- *Index 8 requires offset of 64*
- *8 bytes per doubleword* — 32 bit = 4 byte

```
ld    x9, 64(x22)
add   x9, x21, x9
sd    x9, 96(x22)
```

load double (≠ load word (word è a 4 byte)) — 8 byte = 64 bit — reg dest — INM — reg

8·12

RISC-V does not require words to be aligned in memory

Courtesy of Andrea Marongiu

Immediate Operands

RISC V assume di avere operandi immediati rappresentabili in 12 bit (quindi piccoli) $< 2^{12}$

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Courtesy of Andrea Marongiu

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori` `xori FF..F`
Bit-by-bit NOT	~	~	

  - Useful for extracting and inserting groups of bits in a word

More Load/Store Operations: Byte/Halfword

- RISC-V byte/halfword **load/store**
  - Load byte/halfword: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
  - Load byte/halfword unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
  - Store byte/halfword: Store rightmost 8/16 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`

Courtesy of Andrea Marongiu

More Load/Store Operations: Word/Doubleword

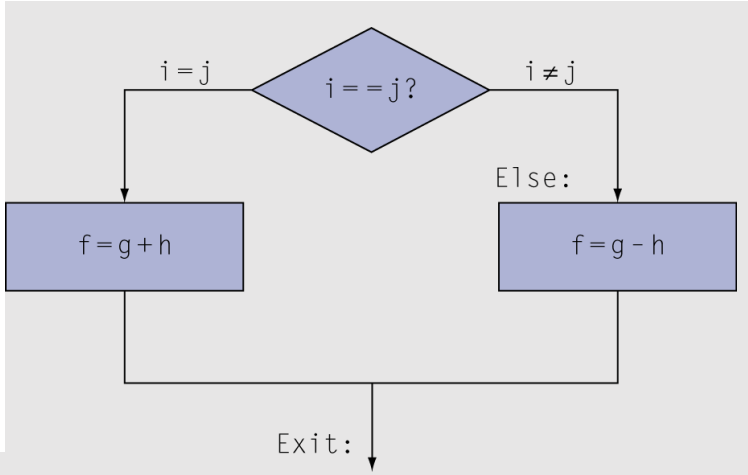
- RISC-V word/doubleword **load/store**
  - Load word/doubleword: Sign extend to 64 bits in rd
    - `lw rd, offset(rs1)`
    - `ld rd, offset(rs1)`
  - Load word/doubleword unsigned: Zero extend to 64 bits in rd
    - `lwu rd, offset(rs1)`
    - `ldu rd, offset(rs1)`
  - Store word/doubleword: Store rightmost 32/64 bits
    - `sw rs2, offset(rs1)`
    - `sd rs2, offset(rs1)`

Courtesy of Andrea Marongiu

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- **`beq rs1, rs2, L1`**
  - if (rs1 == rs2) branch to instruction labeled L1
- **`bne rs1, rs2, L1`**
  - if (rs1 != rs2) branch to instruction labeled L1



C code:

```
if (i==j) f = g+h;
else f = g-h;
```

- *f, g, … in x19, x20, …*

Compiled RISC-V code:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,  x0,  Exit
Else:   sub x19, x20, x21
Exit:   …
```

Assembler calculates addresses

Courtesy of Andrea Marongiu

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - *i in x22, k in x24, address of save in x25*

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

shifto a sx di 3 posizioni → lo sto moltiplicando per 8 e lo salvo in x10

Courtesy of Andrea Marongiu

More Conditional Operations

› **`blt rs1, rs2, L1`**
  – if (rs1 < rs2) branch to instruction labeled L1
› **`bge rs1, rs2, L1`**
  – if (rs1 >= rs2) branch to instruction labeled L1

- Signed comparison: **`blt`**, **`bge`**
- Unsigned comparison: **`bltu`**, **`bgeu`**

C code:

```
if (a > b) a += 1;
```

- *a in x22, b in x23*

Compiled RISC-V code:

```
      bge x23, x22, Exit
      addi x22, x22, 1
Exit: …
```

Courtesy of Andrea Marongiu

RISC-V assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load doubleword	`ld x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	`Memory[x6 + 40] = x5`	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant

Courtesy of Andrea Marongiu

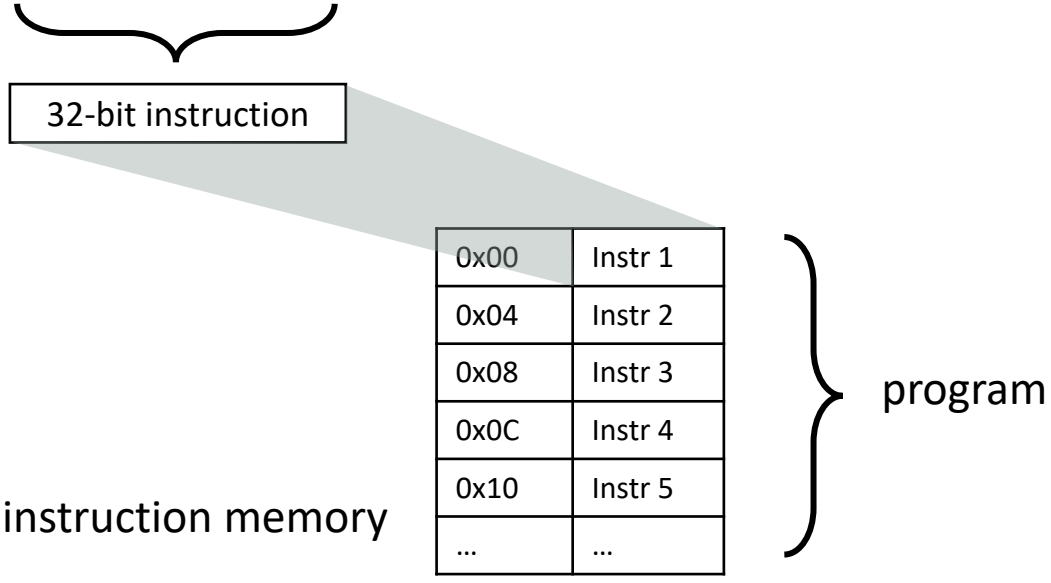
RISC-V assembly language (2)

Category	Instruction	Example	Meaning	Comments
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

Courtesy of Andrea Marongiu

RISC-V Instruction Encoding

R-type	Arithmetic with register operands
I-type	Loads & arithmetic with immediate operand
S-type	Stores
SB-type	Conditional branch
UJ-type	Unconditional jump
U-type	Upper immediate



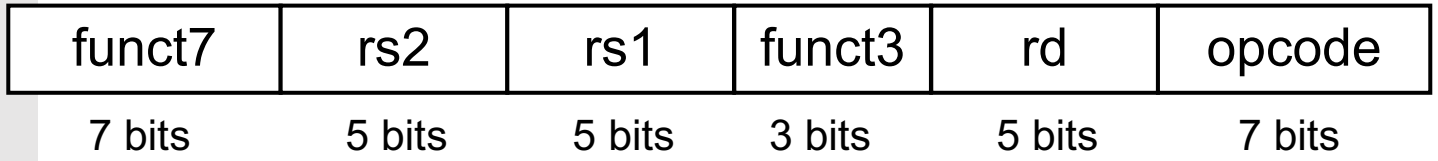
- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

Courtesy of Andrea Marongiu

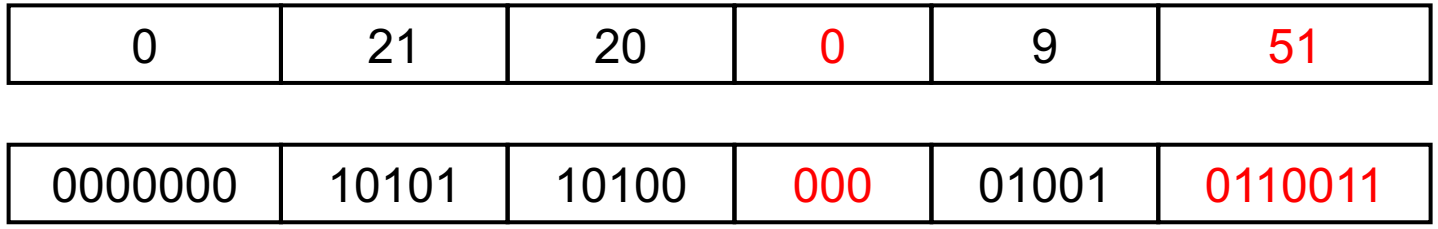
RISC-V **R-format** Instructions

Instruction fields

- *opcode*: operation code
- *rd*: destination register number
- *funct3*: 3-bit function code (additional opcode)
- *rs1*: the first source register number
- *rs2*: the second source register number
- *funct7*: 7-bit function code (additional opcode)



add x9,x20,x21



Courtesy of Andrea Marongiu

RISC-V I-format Instructions

Immediate arithmetic and **load instructions**

- *rs1*: source or base address register number
- *immediate*: constant operand, or offset added to base address
  - 2s-complement, sign extended

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Design Principle 3: **Good design demands good compromises**

- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible

```
addi x10,x10,128   // second operand is an immediate
```

128	10	0	10	19
0000 1000 0000	01010	000	01010	0010011

```
ld x9,64(x22)
```

64	22	3	9	3
0000 0100 0000	1011 0	011	0100 1	000 0011

Courtesy of Andrea Marongiu

Shift Operations

- **I-format** with just 6 bits for immediate
- *immed*: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - **`slli`** by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - **`srli`** by *i* bits divides by 2^i (unsigned only)

Courtesy of Andrea Marongiu

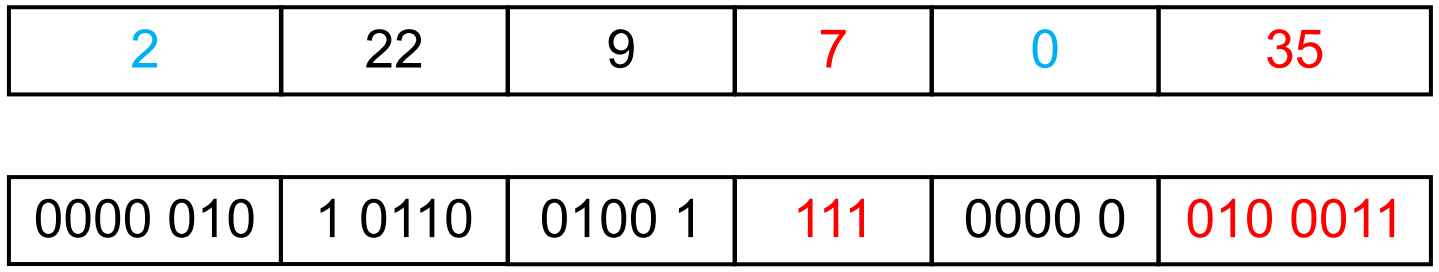
RISC-V **S-format** Instructions

Different immediate format for **store instructions**

- *rs1*: base address register number
- *rs2*: source operand register number
- *immediate*: offset added to base address
  - Split so that *rs1* and *rs2* fields always in the same place

5 least significant bits of $(64)_{10}$

```
sd x9,64(x22)
```

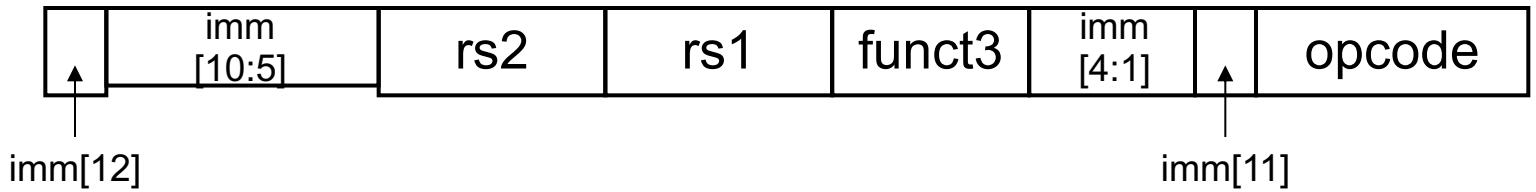


64 >> 5
= 64 / 2^5
= 64/32 = 2

Courtesy of Andrea Marongiu

RISC-V **SB-format** Instructions (branch addressing)

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address = PC + immediate × 2

(Addressing instructions down to halfword)

Courtesy of Andrea Marongiu

RISC-V **UJ-format** Instructions (jump addressing)

- Jump and link (**`jal`**) uses 20-bit immediate for larger range

  - For long jumps, eg, to 32-bit absolute address
    - **`lui`**: load address[31:12] to temp register
    - **`jalr`**: add address[11:0] and jump to target

Courtesy of Andrea Marongiu

More Load/Store Operations: 32-bit Constants

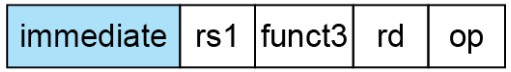
- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant **we need two instructions**:
  1. *Load upper immediate* (`lui`)

     `lui rd, constant`
     - Copies 20-bit constant to bits [31:12] of rd
     - Extends bit 31 to bits [63:32]
     - Clears bits [11:0] of rd to 0
  2. Any other instruction that populates lower bits [11:0]
     - e.g., **`addi`**

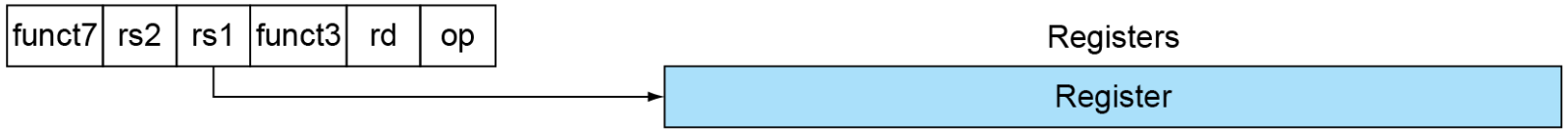
Courtesy of Andrea Marongiu

RISC-V Addressing Summary

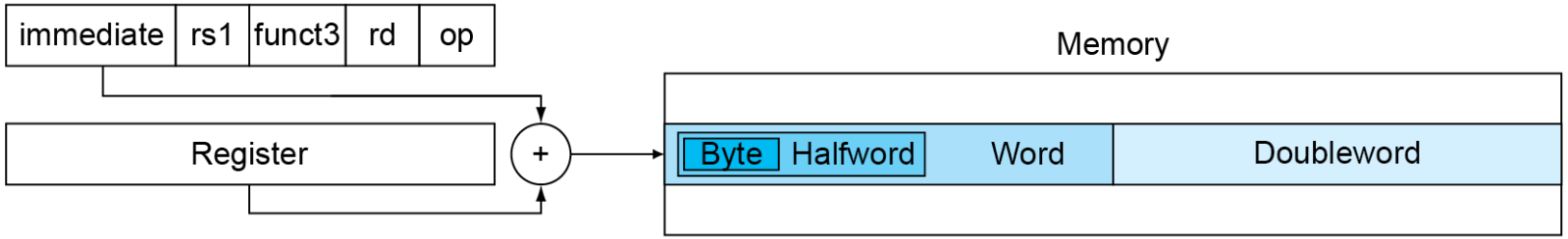
1. Immediate addressing



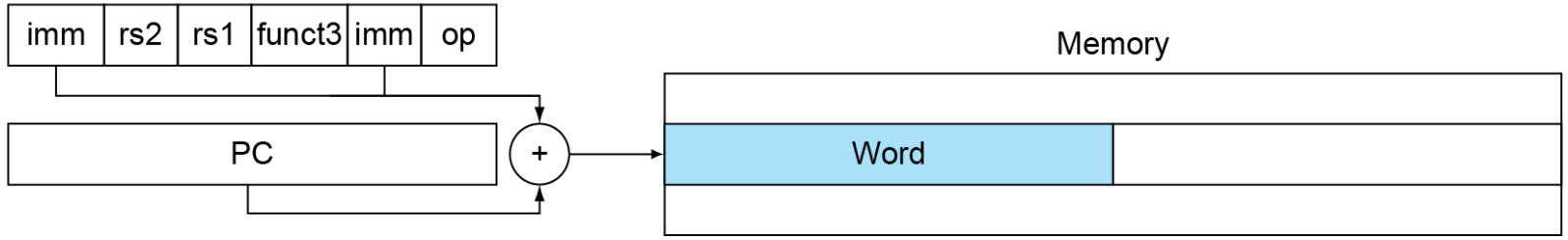
2. Register addressing



3. Base addressing



4. PC-relative addressing



Courtesy of Andrea Marongiu

RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Courtesy of Andrea Marongiu

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100

opcode is same for a *family* of instructions

which get differentiated by the ***funct3*** bits

and the ***funct6/7*** bits

Courtesy of Andrea Marongiu

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.

Courtesy of Andrea Marongiu

RISC-V Encoding Summary

Format	Instruction	Opcode	Funct3	Funct6/7
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

Courtesy of Andrea Marongiu

RISC-V procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Courtesy of Andrea Marongiu

Procedure Call Instructions

- Procedure call: jump and link

```
jal x1, ProcedureLabel
```

  - Address of following instruction put in x1
  - Jumps to target address (ProcedureLabel)
- Procedure return: jump and link register

```
jalr x0, 0(x1)
```

  - Like **jal**, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps

Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

Courtesy of Andrea Marongiu

Leaf Procedure Example

A procedures that doesn't call other procedures

C code:

```
typedef long long int lli;
lli leaf_example (lli g, lli h, lli i, lli j)
  {
    lli f;
    f = (g + h) - (i + j);
    return f;
  }
```

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack (spill to mem)

RISC-V Registers

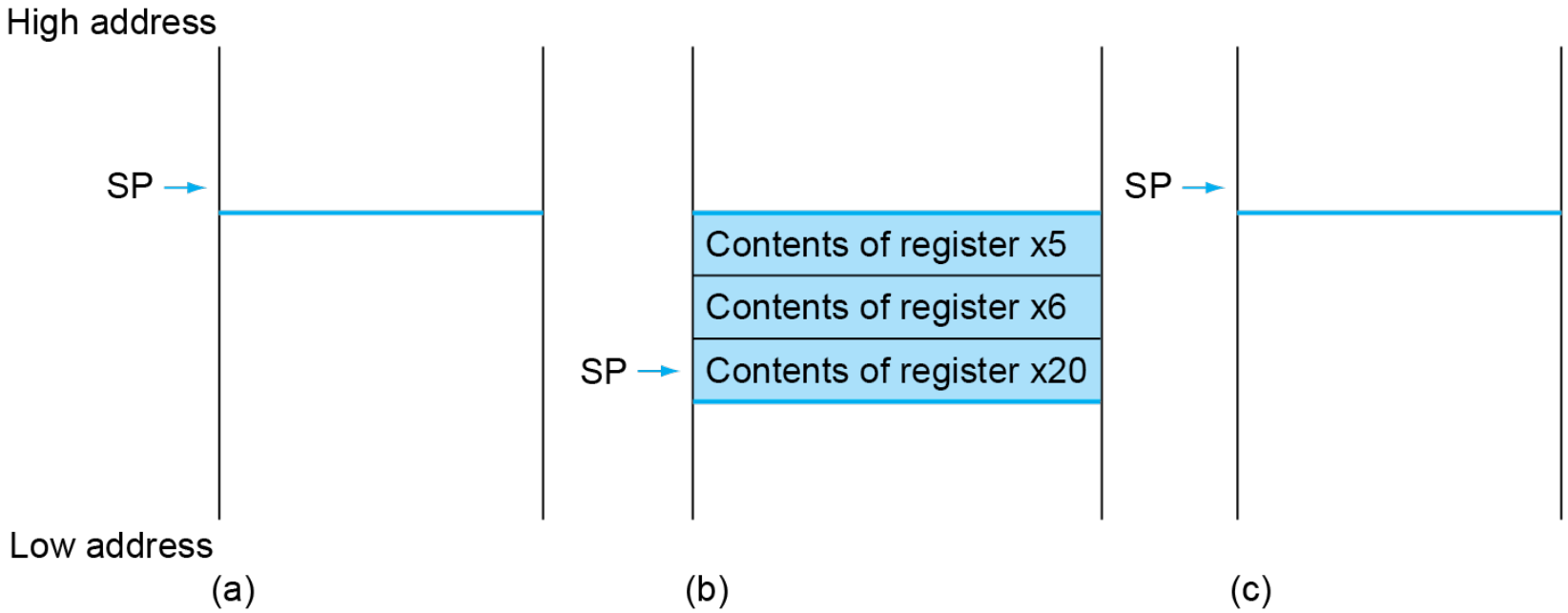
- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

RISC-V code:

```
addi sp,sp,-24
sd   x5,16(sp)   Save x5, x6, x20 on stack
sd   x6,8(sp)
sd   x20,0(sp)
add  x5,x10,x11  x5 = g + h
add  x6,x12,x13  x6 = i + j
sub  x20,x5,x6   f = x5 – x6
addi x10,x20,0   copy f to return register
ld   x20,0(sp)   Restore x5,x6,x20 from stack
ld   x6,8(sp)
ld   x5,16(sp)
addi sp,sp,24
jalr x0,0(x1)    Return to caller
```

Courtesy of Andrea Marongiu

Local Data on the Stack



Courtesy of Andrea Marongiu

Register Usage – Calling convention

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the **callee** (volatile across calls, must be saved by the **caller** if later used)
- x8 – x9, x18 – x27: saved registers
  - Preserved across calls. If used, the **callee** saves and restores them
- In previous example, the stores/loads on x5 and x6 can be dropped

caller: who calls a function
callee: the function itself

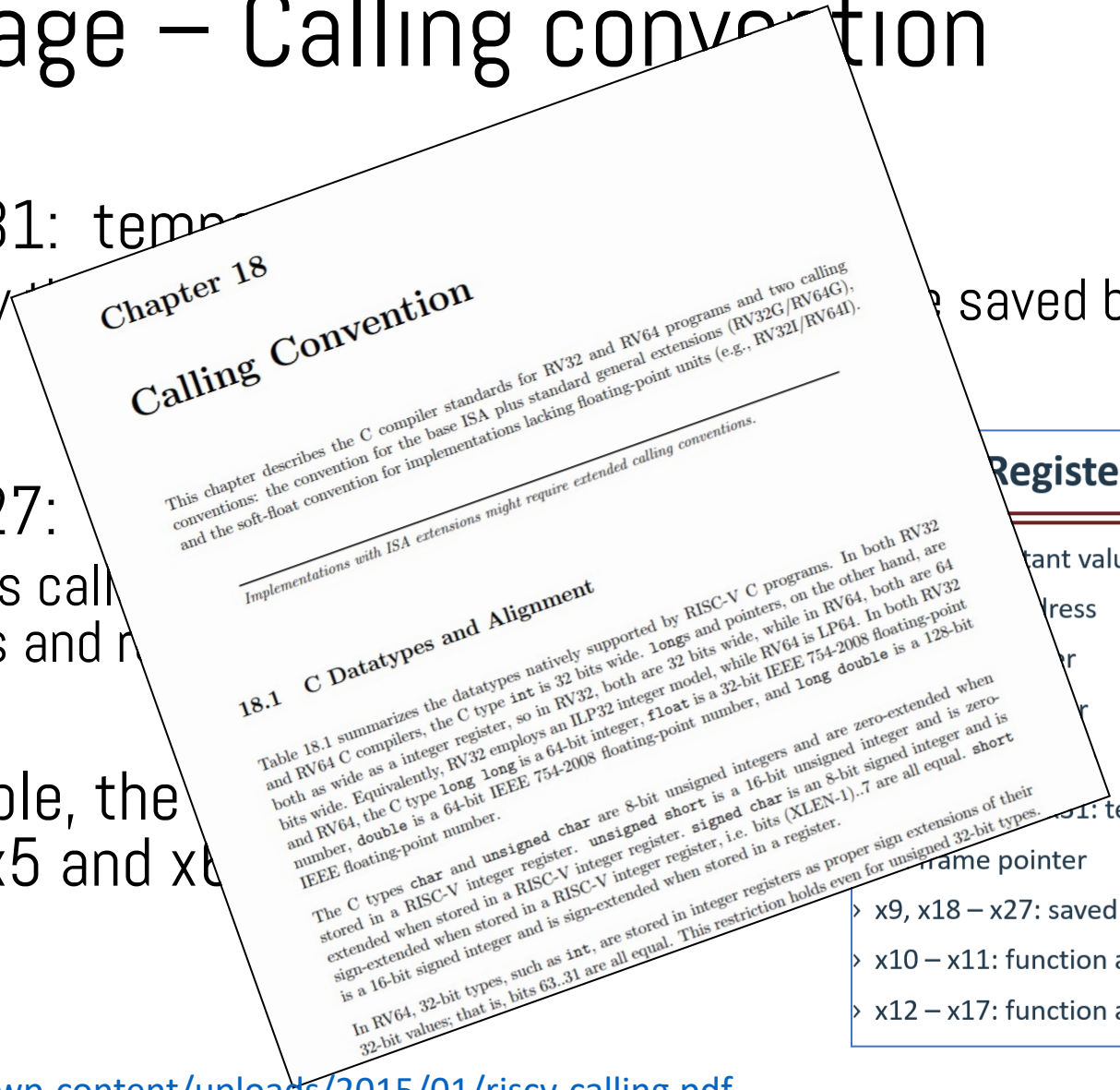
RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Courtesy of Andrea Marongiu

Register Usage – Calling convention

- x5 – x7, x28 – x31: temp
  - Not preserved by t
  - e saved by the **caller** if later used)
- x8 – x9, x18 – x27:
  - Preserved across call
  - the **callee** saves and r
- In previous example, the
  stores/loads on x5 and x
  can be dropped



Chapter 18

Calling Convention

This chapter describes the C compiler standards for RV32 and RV64 programs and two calling conventions: the convention for the base ISA plus standard general extensions (RV32G/RV64G), and the soft-float convention for implementations lacking floating-point units (e.g., RV32I/RV64I).

Implementations with ISA extensions might require extended calling conventions.

18.1 C Datatypes and Alignment

Table 18.1 summarizes the datatypes natively supported by RISC-V C programs. In both RV32 and RV64 C compilers, the C type `int` is 32 bits wide. `long`s and pointers, on the other hand, are both as wide as a integer register, so in RV32, both are 32 bits wide, while in RV64, both are 64 bits wide. Equivalently, RV32 employs an ILP32 integer model, while RV64 is LP64. In both RV32 and RV64, the C type `long long` is a 64-bit integer, `float` is a 32-bit IEEE 754-2008 floating-point number, `double` is a 64-bit IEEE 754-2008 floating-point number, and `long double` is a 128-bit IEEE floating-point number.

The C types `char` and `unsigned char` are 8-bit unsigned integers and are zero-extended when stored in a RISC-V integer register. `unsigned short` is a 16-bit unsigned integer and is zero-extended when stored in a RISC-V integer register. `signed char` is an 8-bit signed integer and is sign-extended when stored in a RISC-V integer register, i.e. bits (XLEN-1)..7 are all equal. `short` is a 16-bit signed integer and is sign-extended when stored in a register.

In RV64, 32-bit types, such as `int`, are stored in integer registers as proper sign extensions of their 32-bit values; that is, bits 63..31 are all equal. This restriction holds even for unsigned 32-bit types.

Registers

tant value 0

dress

r

r

31: temporaries

rame pointer

› x9, x18 – x27: saved registers

› x10 – x11: function arguments/results

› x12 – x17: function arguments

https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf

Courtesy of Andrea Marongiu

FP Instructions in RISC-V

- Separate FP registers: **`f0`**, …, **`f31`**
  - double-precision
  - single-precision values stored in the lower 32 bits

- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact

- FP load and store instructions
  - **`flw, fld`**
  - **`fsw, fsd`**

Courtesy of Andrea Marongiu

FP Instructions in RISC-V

- Single-precision arithmetic
  - **`fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`**
    - e.g., **`fadds.s f2, f4, f6`**
- Double-precision arithmetic
  - **`fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`**
    - e.g., **`fadd.d f2, f4, f6`**
- Single- and double-precision comparison
  - **`feq.s, flt.s, fle.s`**
  - **`feq.d, flt.d, fle.d`**
  - Result is 0 or 1 in integer destination register
    - Use beq, bne to branch on comparison result
- Branch on FP condition code true or false
  - **`B.cond`**

Courtesy of Andrea Marongiu

Backup

Work continues on new RISC-V extensions

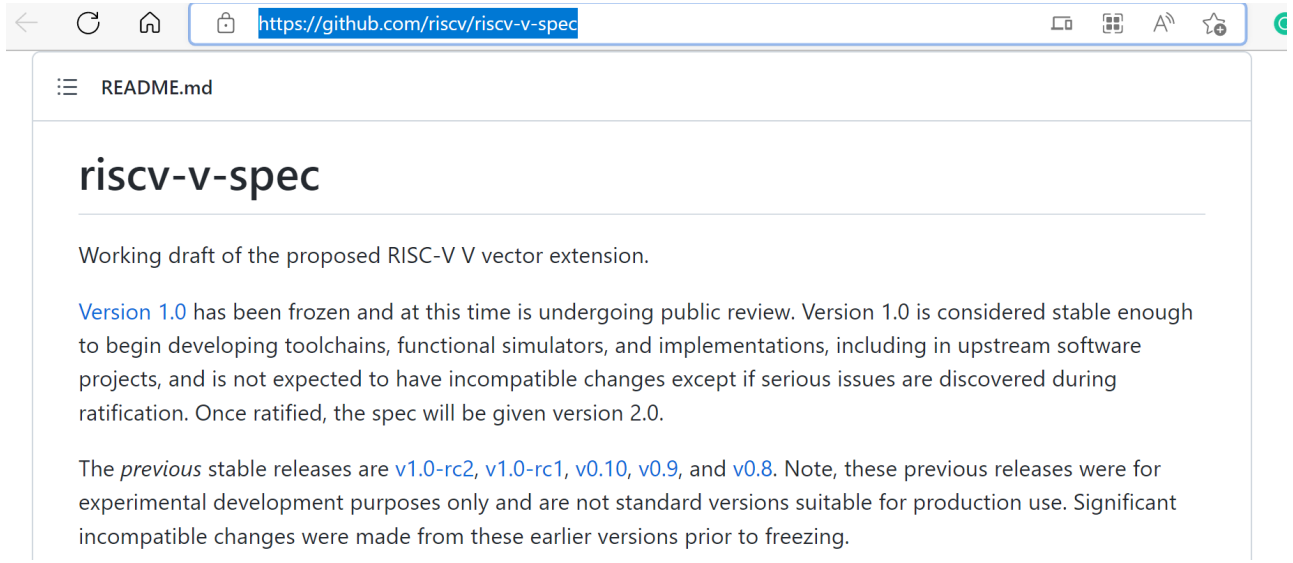
- Foundation members work in task-groups
- Dedicated task-groups
  - Formal specification
  - Memory Model
  - Marketing
  - External Debug Specification

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

For more updated extensions status:
https://en.wikipedia.org/wiki/RISC-V

Work continues on new RISC-V extensions

- Example of the process:
- Vector Operation – V extension
- Frozen, not yet ratified

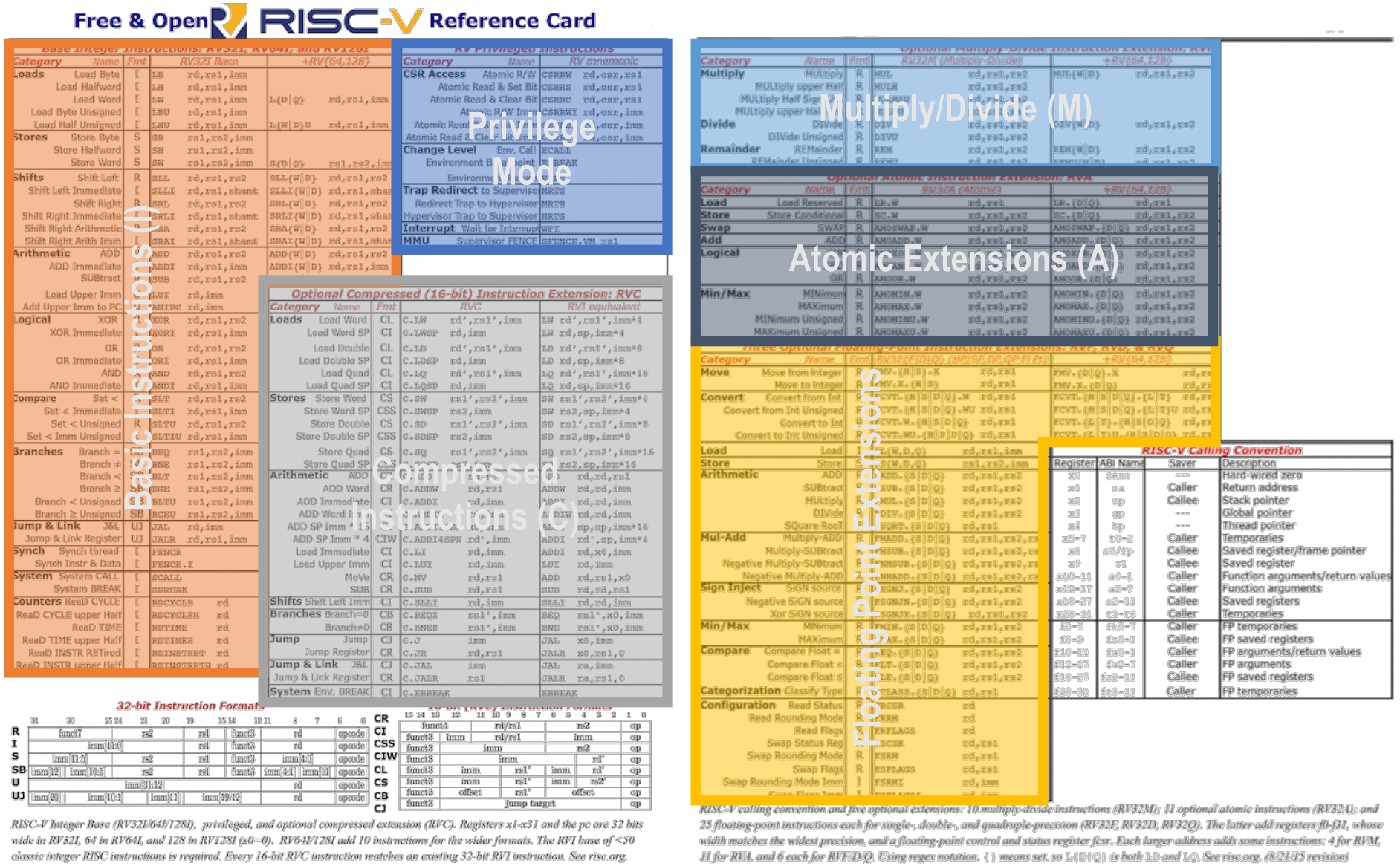


[GitHub - riscv/riscv-v-spec: Working draft of the proposed RISC-V V vector extension]

https://wiki.riscv.org/display/HOME/Recently+Ratified+Extensions

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

Reduced Instruction Set: all in one page



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)

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*	$\geq 80b$

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in `f10`, result in `f10`, literals in global memory space

- Compiled RISC-V code:

```
f2c:
    flw     f0,const5(x3)      // f0 = 5.0f
    flw     f1,const9(x3)      // f1 = 9.0f
    fdiv.s  f0, f0, f1         // f0 = 5.0f / 9.0f
    flw     f1,const32(x3)     // f1 = 32.0f
    fsub.s  f10,f10,f1         // f10 = fahr - 32.0
    fmul.s  f10,f0,f10         // f10 = (5.0f/9.0f) * (fahr-32.0f)
    jalr    x0,0(x1)           // return
```

FP Example: Array Multiplication

- C = C + A × B
  - All 32 × 32 matrices, 64-bit double-precision elements
  - DGEMM (Double precision GEneral Matrix Multiply)
- C code:

```
void mm (double c[][],
         double a[][], double b[][]) {
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1)
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j]
                + a[i][k] * b[k][j];
}
```

  - Addresses of `c`, `a`, `b` in `x10`, `x11`, `x12`, and `i`, `j`, `k` in `x5`, `x6`, `x7`

FP Example: Array Multiplication

- RISC-V code:

```
   mm:...
          li    x28,32        // x28 = 32 (row size/loop end)
          li    x5,0          // i = 0; initialize 1st for loop
L1:    li    x6,0          // j = 0; initialize 2nd for loop
L2:    li    x7,0          // k = 0; initialize 3rd for loop
          slli  x30,x5,5      // x30 = i * 2**5 (size of row of c)
          add   x30,x30,x6    // x30 = i * size(row) + j
          slli  x30,x30,3     // x30 = byte offset of [i][j]
          add   x30,x10,x30   // x30 = byte address of c[i][j]
          fld   f0,0(x30)     // f0 = c[i][j]
L3:    slli  x29,x7,5      // x29 = k * 2**5 (size of row of b)
          add   x29,x29,x6    // x29 = k * size(row) + j
          slli  x29,x29,3     // x29 = byte offset of [k][j]
          add   x29,x12,x29   // x29 = byte address of b[k][j]
          fld   f1,0(x29)     // f1 = b[k][j]
```

FP Example: Array Multiplication

…

```
slli    x29,x5,5      // x29 = i * 2**5 (size of row of a)
add     x29,x29,x7    // x29 = i * size(row) + k
slli    x29,x29,3     // x29 = byte offset of [i][k]
add     x29,x11,x29   // x29 = byte address of a[i][k]
fld     f2,0(x29)     // f2 = a[i][k]
fmul.d f1, f2, f1     // f1 = a[i][k] * b[k][j]
fadd.d f0, f0, f1     // f0 = c[i][j] + a[i][k] * b[k][j]
addi    x7,x7,1       // k = k + 1
bltu    x7,x28,L3     // if (k < 32) go to L3
fsd     f0,0(x30)     // c[i][j] = f0
addi    x6,x6,1       // j = j + 1
bltu    x6,x28,L2     // if (j < 32) go to L2
addi    x5,x5,1       // i = i + 1
bltu    x5,x28,L1     // if (i < 32) go to L1
```

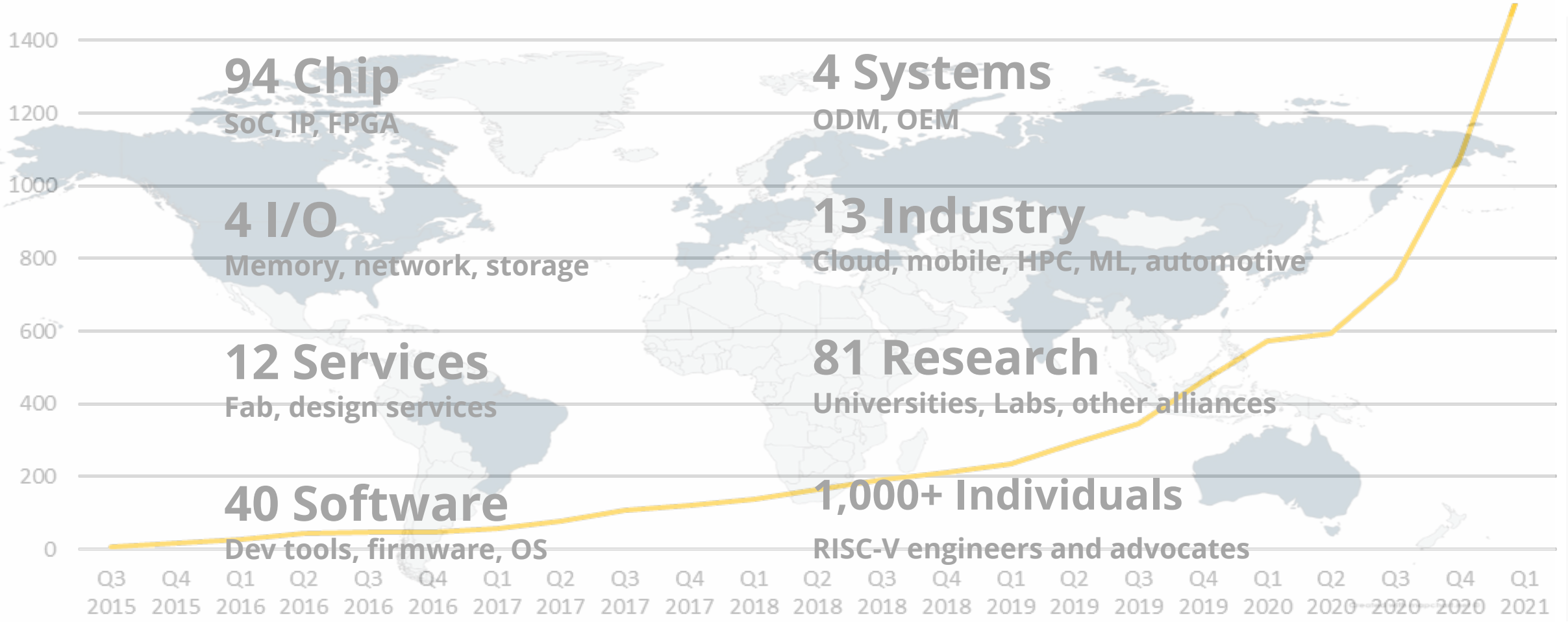
Pitfall: Right Shift and Division

- Left shift by i places multiplies an integer by 2^i
- Right shift divides by 2^i?
  - Only for unsigned integers
- For signed integers
  - Arithmetic right shift: replicate the sign bit
  - e.g., −5 / 4
    - $11111011_2 >> 2 = 11111110_2 = -2$
    - Rounds toward −∞
  - c.f. $11111011_2 >> 2 = 00111110_2 = +62$

What is one advantage of a CISC ISA?

A. It naturally supports a faster clock.
B. Instructions are easier to decode.
C. The static footprint of the code will be smaller.
D. The code is easier for a compiler to optimize.
E. You have a lot of registers to use.

More than 1,500 RISC-V Members
across 70 Countries



94 Chip
SoC, IP, FPGA

4 Systems
ODM, OEM

4 I/O
Memory, network, storage

13 Industry
Cloud, mobile, HPC, ML, automotive

12 Services
Fab, design services

81 Research
Universities, Labs, other alliances

40 Software
Dev tools, firmware, OS

1,000+ Individuals
RISC-V engineers and advocates

In 2020, RISC-V membership grew 133%