European
Processor
Initiative

# A RISC-V vector CPU for HPC:

## architecture, platforms and tools to make it happen

Filippo Mantovani, Barcelona Supercomputing Center (BSC)

European
Processor
Initiative

# Introduction to RISC-V

# The value of having “standards”

# The value of having “standards”

**DISCLAIMER**: Apple users may not fully understand this slide

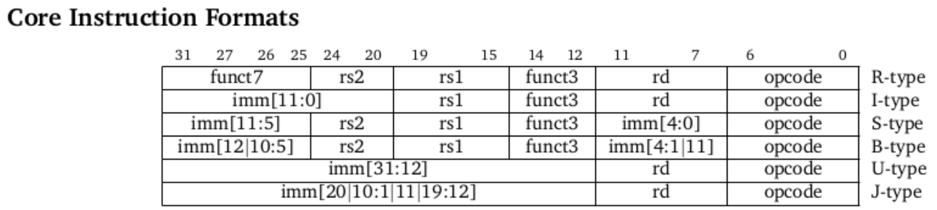
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# What is RISC-V?

- ➩ Simple and modular Instruction Set Architecture (ISA)
- ➩ Research project started at Berkeley in 2010
- ➩ In 2014 ISA ratified
- ➩ RISC-V (pronounced "risc five", as it is
  the fifth generation of RISC ISA at Berkeley)

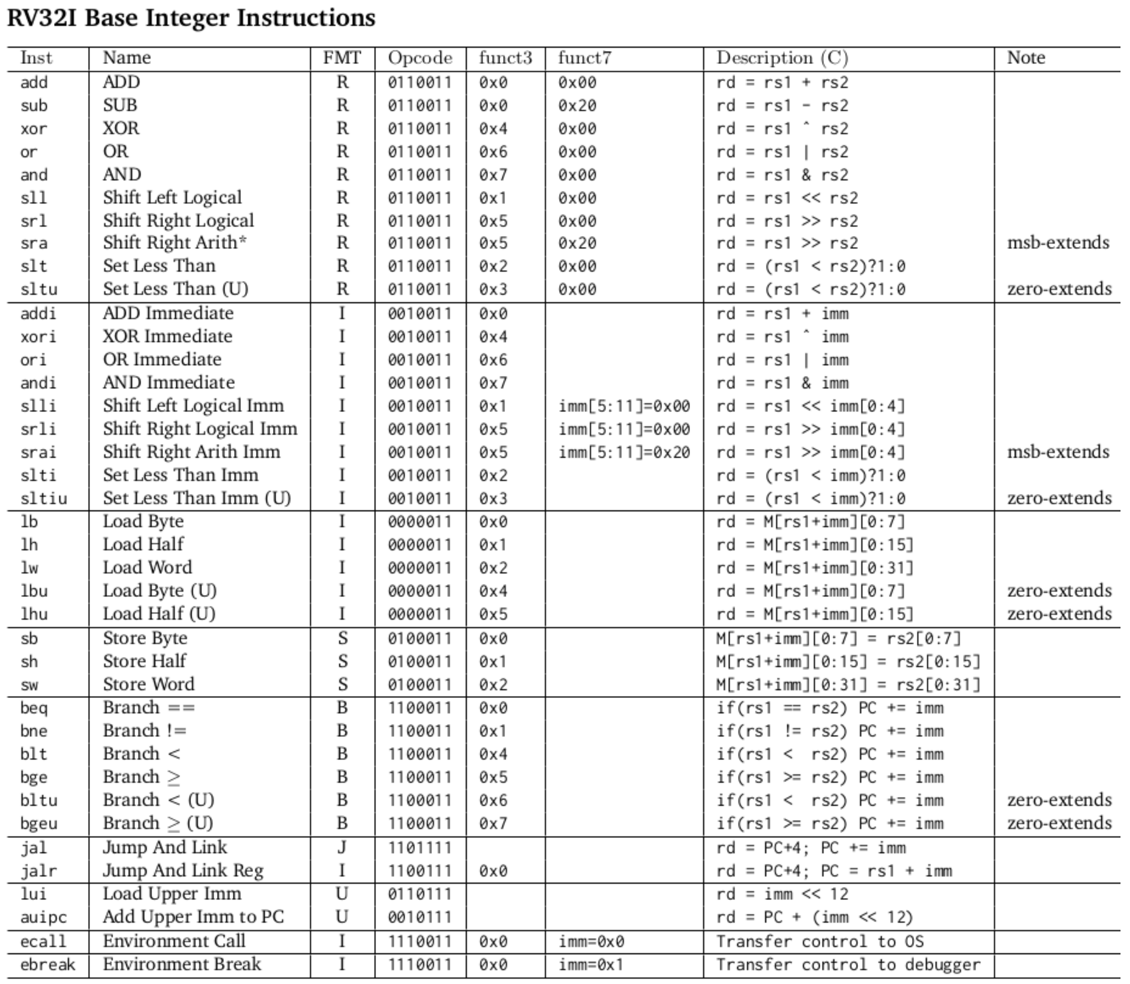
📙 Waterman, Andrew., Patterson, David A.. **The RISC-V Reader: An Open Architecture Atlas**. United States: Strawberry Canyon LLC, 2017.

📙 Patterson, David A.., Hennessy, John L.. **Computer Organization and Design RISC-V Edition: The Hardware Software Interface**. Netherlands: Elsevier Science, 2017.



**Core Instruction Formats**

| 31 27 | 26 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[12\|10:5] | | rs2 | rs1 | funct3 | imm[4:1\|11] | opcode | B-type |
| imm[31:12] | | | | | rd | opcode | U-type |
| imm[20\|10:1\|11\|19:12] | | | | | rd | opcode | J-type |



**RV32I Base Integer Instructions**

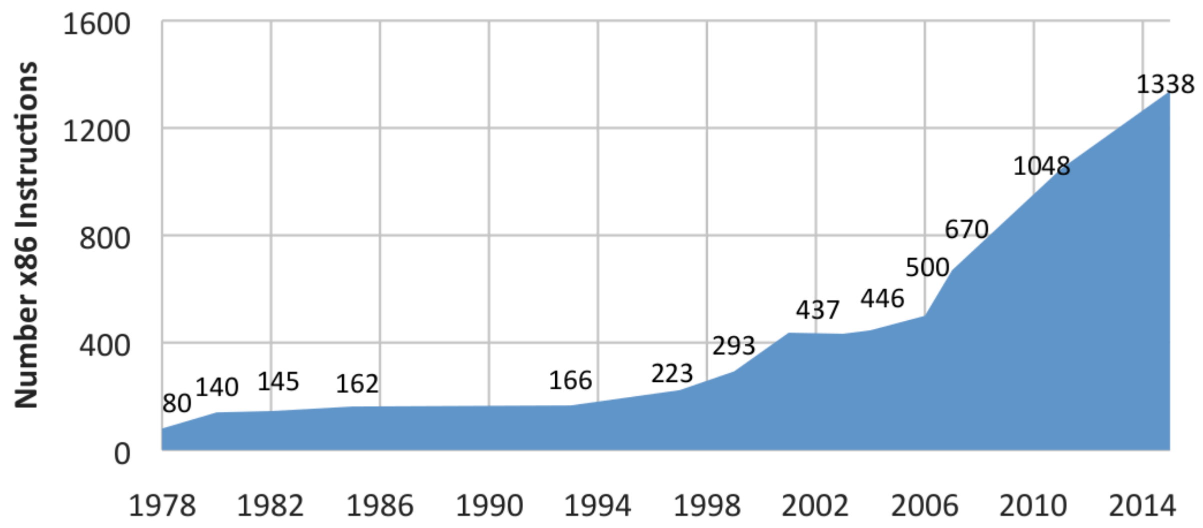
| Inst | Name | FMT | Opcode | funct3 | funct7 | Description (C) | Note |
|---|---|---|---|---|---|---|---|
| add | ADD | R | 0110011 | 0x0 | 0x00 | rd = rs1 + rs2 | |
| sub | SUB | R | 0110011 | 0x0 | 0x20 | rd = rs1 - rs2 | |
| xor | XOR | R | 0110011 | 0x4 | 0x00 | rd = rs1 ˆ rs2 | |
| or | OR | R | 0110011 | 0x6 | 0x00 | rd = rs1 \| rs2 | |
| and | AND | R | 0110011 | 0x7 | 0x00 | rd = rs1 & rs2 | |
| sll | Shift Left Logical | R | 0110011 | 0x1 | 0x00 | rd = rs1 << rs2 | |
| srl | Shift Right Logical | R | 0110011 | 0x5 | 0x00 | rd = rs1 >> rs2 | |
| sra | Shift Right Arith* | R | 0110011 | 0x5 | 0x20 | rd = rs1 >> rs2 | msb-extends |
| slt | Set Less Than | R | 0110011 | 0x2 | 0x00 | rd = (rs1 < rs2)?1:0 | |
| sltu | Set Less Than (U) | R | 0110011 | 0x3 | 0x00 | rd = (rs1 < rs2)?1:0 | zero-extends |
| addi | ADD Immediate | I | 0010011 | 0x0 | | rd = rs1 + imm | |
| xori | XOR Immediate | I | 0010011 | 0x4 | | rd = rs1 ˆ imm | |
| ori | OR Immediate | I | 0010011 | 0x6 | | rd = rs1 \| imm | |
| andi | AND Immediate | I | 0010011 | 0x7 | | rd = rs1 & imm | |
| slli | Shift Left Logical Imm | I | 0010011 | 0x1 | imm[5:11]=0x00 | rd = rs1 << imm[0:4] | |
| srli | Shift Right Logical Imm | I | 0010011 | 0x5 | imm[5:11]=0x00 | rd = rs1 >> imm[0:4] | |
| srai | Shift Right Arith Imm | I | 0010011 | 0x5 | imm[5:11]=0x20 | rd = rs1 >> imm[0:4] | msb-extends |
| slti | Set Less Than Imm | I | 0010011 | 0x2 | | rd = (rs1 < imm)?1:0 | |
| sltiu | Set Less Than Imm (U) | I | 0010011 | 0x3 | | rd = (rs1 < imm)?1:0 | zero-extends |
| lb | Load Byte | I | 0000011 | 0x0 | | rd = M[rs1+imm][0:7] | |
| lh | Load Half | I | 0000011 | 0x1 | | rd = M[rs1+imm][0:15] | |
| lw | Load Word | I | 0000011 | 0x2 | | rd = M[rs1+imm][0:31] | |
| lbu | Load Byte (U) | I | 0000011 | 0x4 | | rd = M[rs1+imm][0:7] | zero-extends |
| lhu | Load Half (U) | I | 0000011 | 0x5 | | rd = M[rs1+imm][0:15] | zero-extends |
| sb | Store Byte | S | 0100011 | 0x0 | | M[rs1+imm][0:7] = rs2[0:7] | |
| sh | Store Half | S | 0100011 | 0x1 | | M[rs1+imm][0:15] = rs2[0:15] | |
| sw | Store Word | S | 0100011 | 0x2 | | M[rs1+imm][0:31] = rs2[0:31] | |
| beq | Branch == | B | 1100011 | 0x0 | | if(rs1 == rs2) PC += imm | |
| bne | Branch != | B | 1100011 | 0x1 | | if(rs1 != rs2) PC += imm | |
| blt | Branch < | B | 1100011 | 0x4 | | if(rs1 < rs2) PC += imm | |
| bge | Branch ≥ | B | 1100011 | 0x5 | | if(rs1 >= rs2) PC += imm | |
| bltu | Branch < (U) | B | 1100011 | 0x6 | | if(rs1 < rs2) PC += imm | zero-extends |
| bgeu | Branch ≥ (U) | B | 1100011 | 0x7 | | if(rs1 >= rs2) PC += imm | zero-extends |
| jal | Jump And Link | J | 1101111 | | | rd = PC+4; PC += imm | |
| jalr | Jump And Link Reg | I | 1100111 | 0x0 | | rd = PC+4; PC = rs1 + imm | |
| lui | Load Upper Imm | U | 0110111 | | | rd = imm << 12 | |
| auipc | Add Upper Imm to PC | U | 0010111 | | | rd = PC + (imm << 12) | |
| ecall | Environment Call | I | 1110011 | 0x0 | imm=0x0 | Transfer control to OS | |
| ebreak | Environment Break | I | 1110011 | 0x0 | imm=0x1 | Transfer control to debugger | |

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Technical difference: incremental vs modular ISA

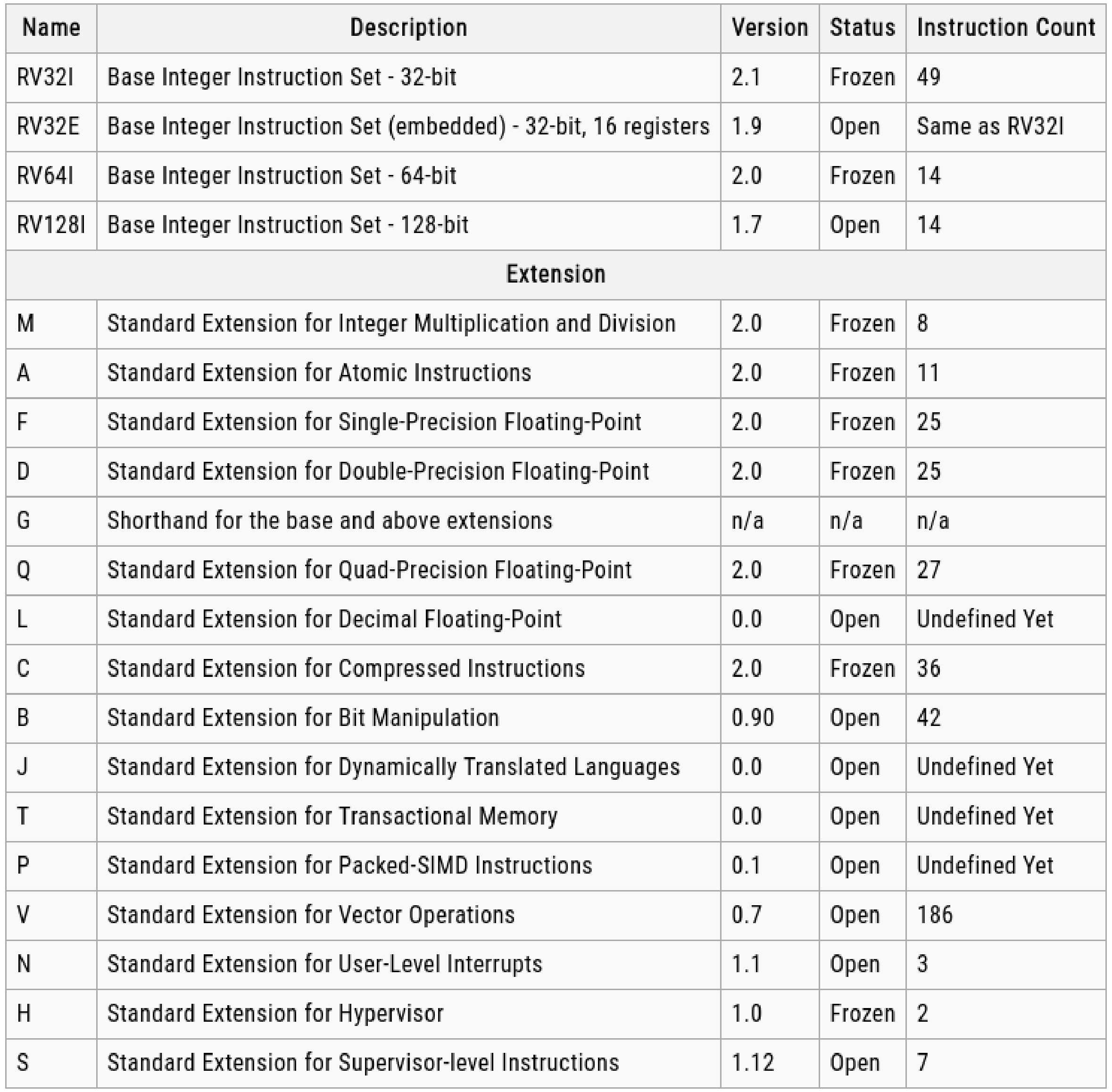
⇨ Intel x86 is an incremental ISA. Each new release:

- Maintain backward compatibility
- Carry on new instructions (also for marketing reasons)



📙 Waterman, Andrew., Patterson, David A.. **The RISC-V Reader: An Open Architecture Atlas**. United States: Strawberry Canyon LLC, 2017.

🏷 https://en.wikichip.org/wiki/risc-v/standard_extensions



| Name | Description | Version | Status | Instruction Count |
|---|---|---|---|---|
| RV32I | Base Integer Instruction Set - 32-bit | 2.1 | Frozen | 49 |
| RV32E | Base Integer Instruction Set (embedded) - 32-bit, 16 registers | 1.9 | Open | Same as RV32I |
| RV64I | Base Integer Instruction Set - 64-bit | 2.0 | Frozen | 14 |
| RV128I | Base Integer Instruction Set - 128-bit | 1.7 | Open | 14 |
| | Extension | | | |
| M | Standard Extension for Integer Multiplication and Division | 2.0 | Frozen | 8 |
| A | Standard Extension for Atomic Instructions | 2.0 | Frozen | 11 |
| F | Standard Extension for Single-Precision Floating-Point | 2.0 | Frozen | 25 |
| D | Standard Extension for Double-Precision Floating-Point | 2.0 | Frozen | 25 |
| G | Shorthand for the base and above extensions | n/a | n/a | n/a |
| Q | Standard Extension for Quad-Precision Floating-Point | 2.0 | Frozen | 27 |
| L | Standard Extension for Decimal Floating-Point | 0.0 | Open | Undefined Yet |
| C | Standard Extension for Compressed Instructions | 2.0 | Frozen | 36 |
| B | Standard Extension for Bit Manipulation | 0.90 | Open | 42 |
| J | Standard Extension for Dynamically Translated Languages | 0.0 | Open | Undefined Yet |
| T | Standard Extension for Transactional Memory | 0.0 | Open | Undefined Yet |
| P | Standard Extension for Packed-SIMD Instructions | 0.1 | Open | Undefined Yet |
| V | Standard Extension for Vector Operations | 0.7 | Open | 186 |
| N | Standard Extension for User-Level Interrupts | 1.1 | Open | 3 |
| H | Standard Extension for Hypervisor | 1.0 | Frozen | 2 |
| S | Standard Extension for Supervisor-level Instructions | 1.12 | Open | 7 |

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Non-technical differences: another business models

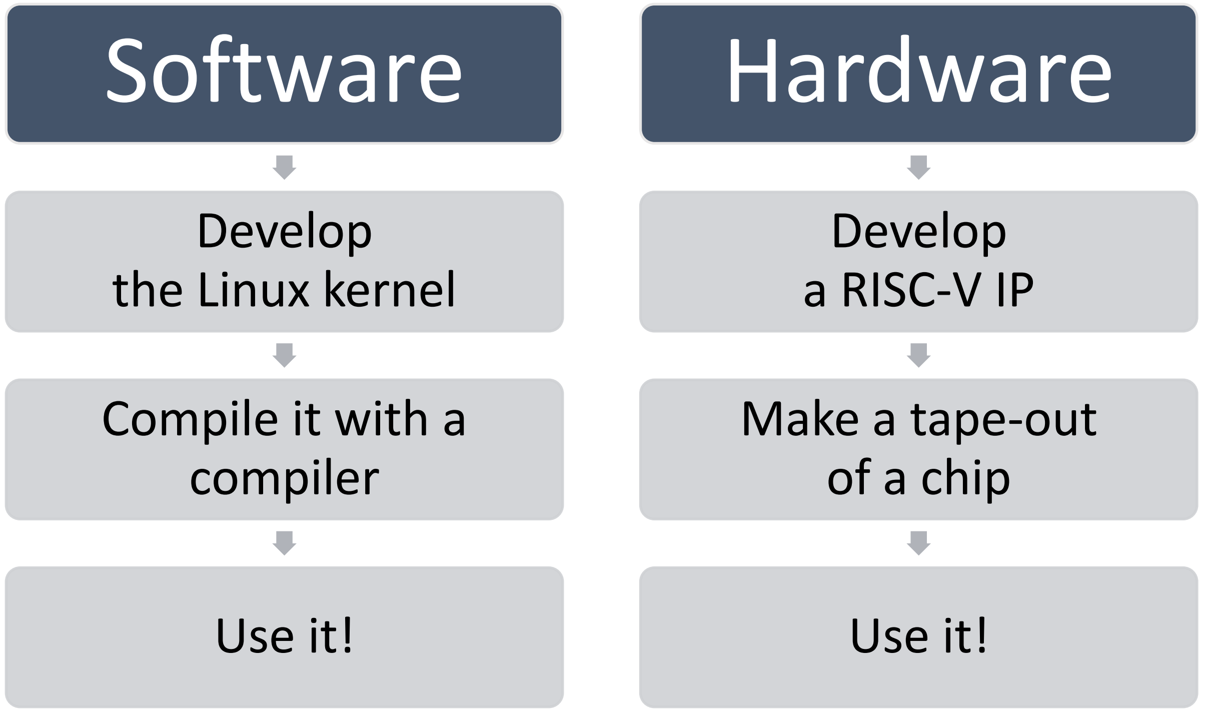
| | intel | arm | RISC-V |
|---|---|---|---|
| Open ISA | ☐ | ☐ | ☑ |
| Adopting the ISA is free | ☐ [1] | ☐ [2] | ☑ |
| It allows development of commercial IPs | ☐ | ☑ | ☑ |
| Everybody can develop commercial IPs | ☐ | ☐ [3] | ☑ |
| It allows access to extension | ☐ | ☐ | ☑ |
| It allows development of open-source IPs | ☐ | ☐ | ☑ |

[1] Adoption of the ISA “de facto” not possible (unless you are AMD)
[2] Adoption of the ISA is possible (under a fee)
[3] Only partners can develop commercial IPs (under a fee)

# False myth: it is not like Linux for software

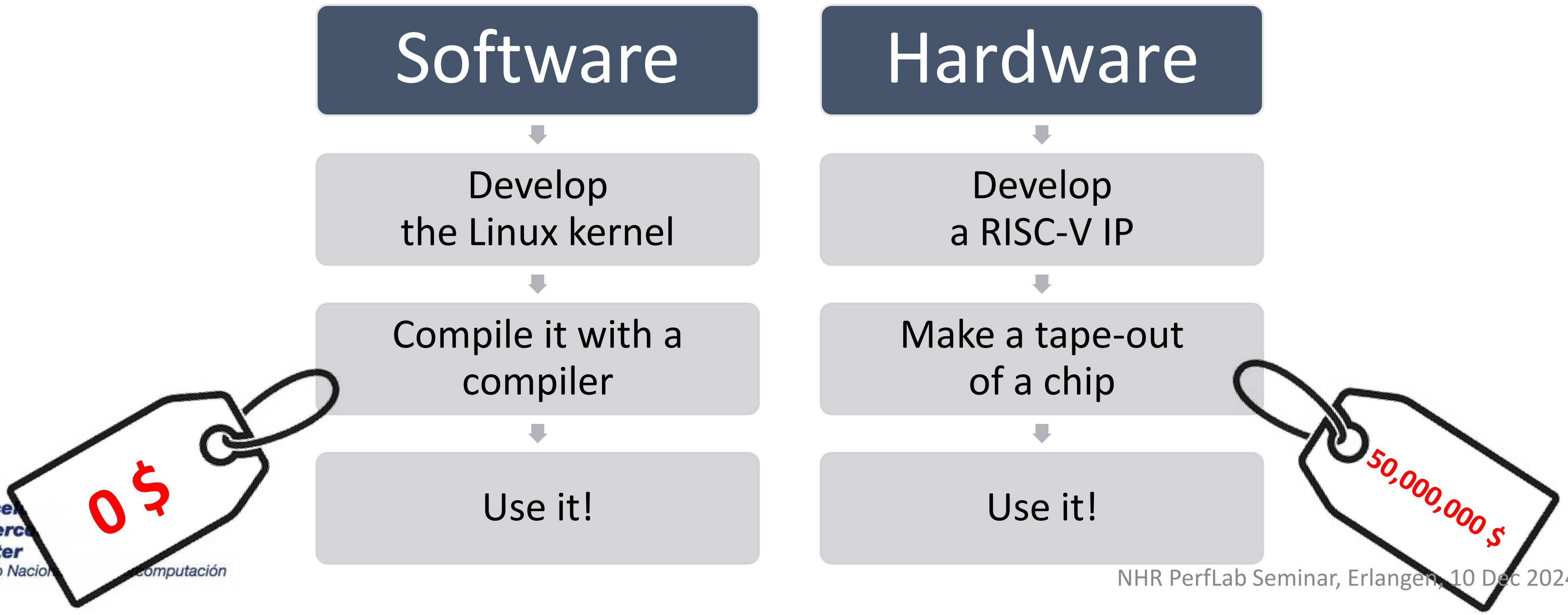
⇨ A shallow analysis often uses the analogy

- "It is the same idea of Linux but in hardware"
- "RISC-V will do to the hardware what Linux did to software"



# False myth: it is not like Linux for software

⇨ A shallow analysis often uses the analogy
- "It is the same idea of Linux but in hardware"
- "RISC-V will do to the hardware what Linux did to software"



## Take-home message

🤓

🎥 Krste Asanović, SiFive, **Advancing HPC with RISC-V** (Supercomputing 2022, invited talk) https://www.youtube.com/watch?v=iFlcJFcOJKk

⇨RISC-V defines an open, free and standard ISA

- Simple and modular (as opposed to incremental)

⇨It defines a new business model

- ISA + extensions remains free
- Implementations can be closed (and sold)
- Implementations can be open

⇨Likely to work as a standard/universal ISA

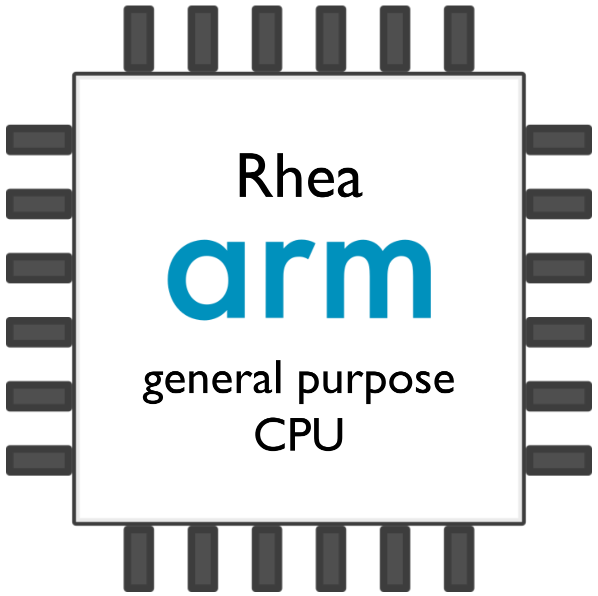
- Independent on market fluctuations (war, bans, …)

European Processor Initiative

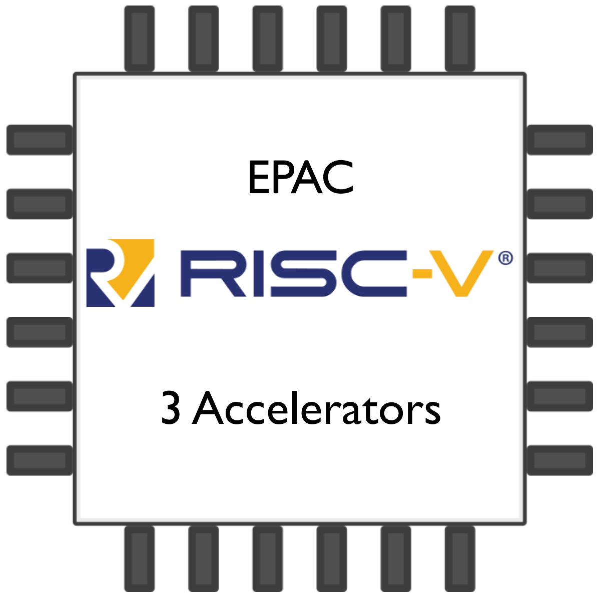
## European Processor Initiative (EPI)

## EPI Main Objective

- To develop European microprocessor and accelerator technology
- Strengthen competitiveness of EU industry and science



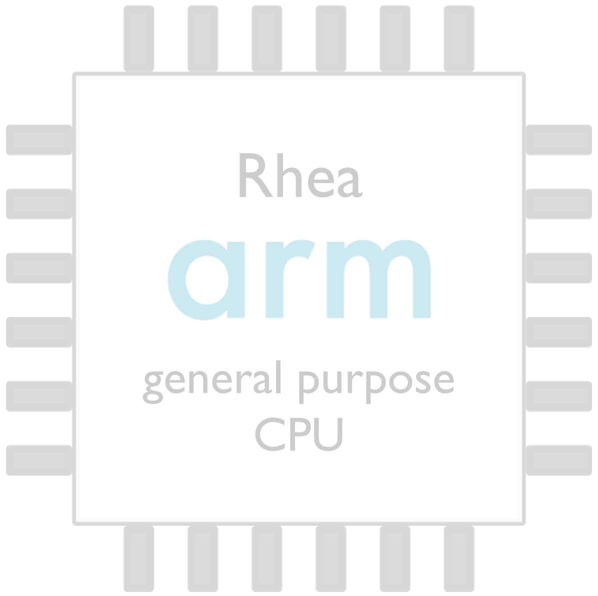
SiPearl, Atos, CEA, UniBo, E4, UniPi, P&R



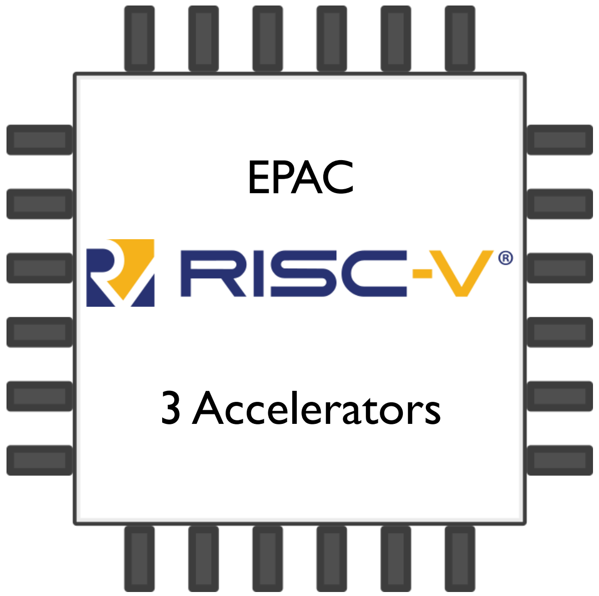
BSC, SemiDynamics, EXTOLL, FORTH, ETHZ, UniBo, UniZG, Chalmers, CEA, E4

## EPI Main Objective

- To develop European microprocessor and accelerator technology
- Strengthen competitiveness of EU industry and science



SiPearl, Atos, CEA, UniBo, E4, UniPi, P&R



BSC, SemiDynamics, EXTOLL, FORTH, ETHZ, UniBo, UniZG, Chalmers, CEA, E4

# EPAC: EPI Accelerator v1.5

GF22FDX, 27 mm2, 0.3 Btr Tape out Mar 2023, Bring up Oct 2023

## VEC tile

General purpose RISC-V CPU
Avispado Core (16 kI$, 32 kD$)
with dedicated VPU
Up to 256 DP element vector length

## VRP tile

General purpose RISC-V CPU supporting variable precision arithmetic up to 256 bit elements

## STX tile

RISC-V many-core machine learning accelerator targeting stencil and tensor arithmetics.

## L2-HN tile

Distributed L2 cache (256 kB/slice) and Coherence Home Node

## CHI NoC and SerDes

On-chip high-speed network based on multiple CHI cross points (XP).

Off-chip link based on SerDes.

Physical design by Fraunhofer
Prototype board integration by E4 COMPUTER ENGINEERING

# EPAC: EPI Accelerator v1.5

GF22FDX, 27 mm2, 0.3 Btr Tape out Mar 2023, Bring up Oct 2023

European Processor Initiative

## VEC tile

General purpose RISC-V CPU
Avispado Core (16 kI$, 32 kD$)
with dedicated VPU
Up to 256 DP element vector length

## VRP tile

General purpose RISC-V CPU supporting variable precision arithmetic up to 256 bit elements

## STX tile

RISC-V many-core machine learning accelerator targeting stencil and tensor arithmetics.

## L2-HN tile

Distributed L2 cache (256 kB/slice) and Coherence Home Node

## CHI NoC and SerDes

On-chip high-speed network based on multiple CHI cross points (XP).

Off-chip link based on SerDes.

Physical design by Fraunhofer
Prototype board integration by E4 COMPUTER ENGINEERING

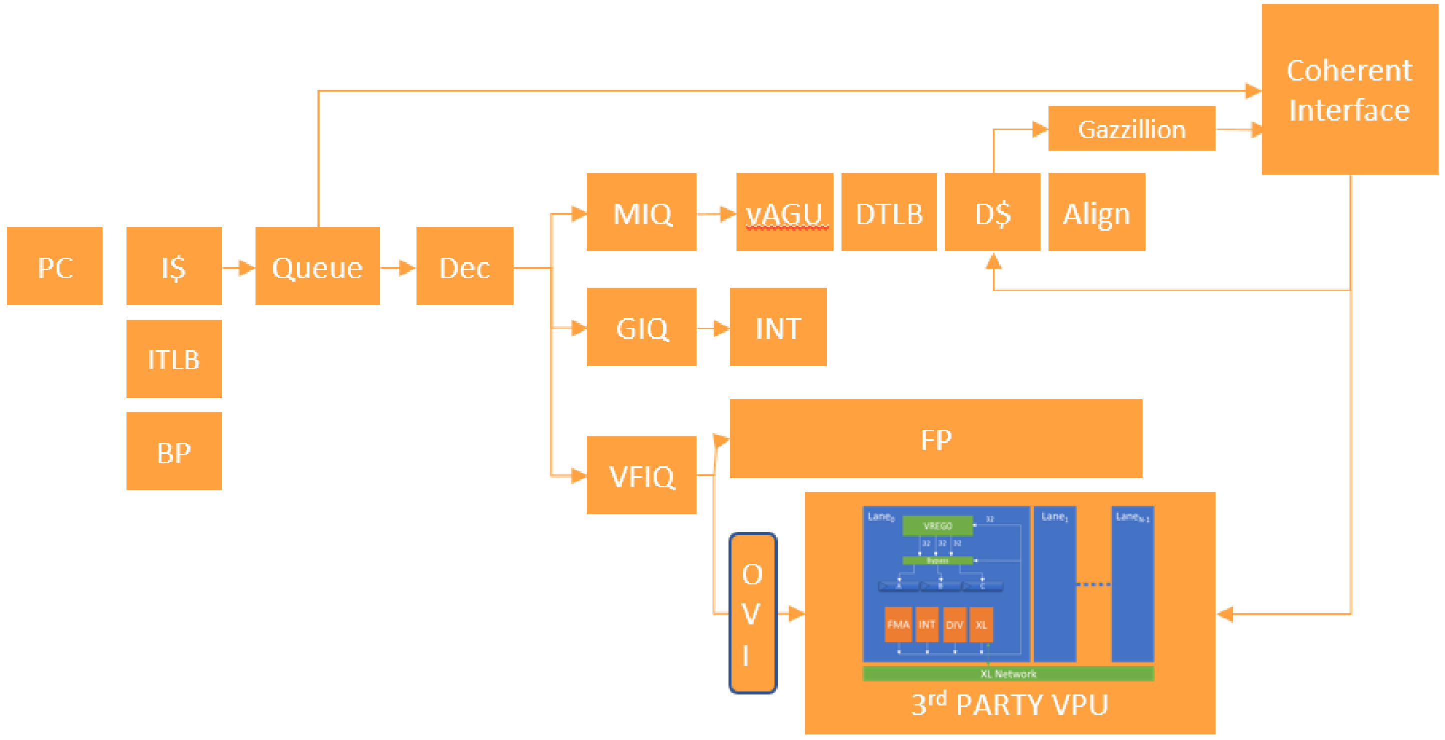
## What’s special in EPAC – VEC?

| | |
|---|---|
| The “Avispado” RISC-V core | semidynamics |
| The Vector Processing Unit (VPU) | BSC Barcelona Supercomputing Center Centro Nacional de Supercomputación |

# What’s special?

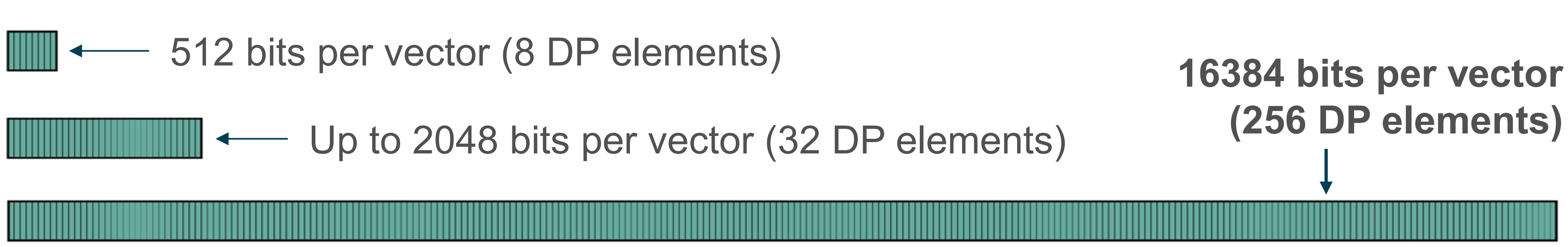
- It boots Linux
- The scalar in-order RISC-V core can release several requests of cache lines to the main memory
- The core is connected to a Vector Processing Unit (VPU) with very wide vector registers (16kb)

  - 16 kB instruction cache
  - 32 kB data cache
  - Decodes v0.7, v1.0 vector extension
  - Full hardware support for unaligned accesses
  - Cache coherent (CHI)
  - Vector memory accesses (vle, vlse, vlxe, vse, …) processed by a dedicated queue (MIQ/LSU)



Courtesy: semidynamics silicon design and verification services

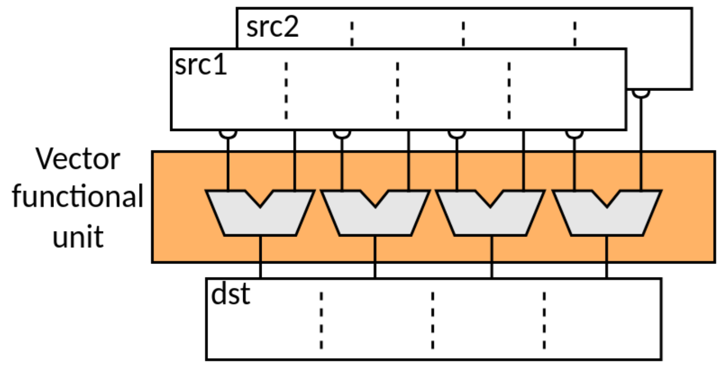
# VPU with Long Vector Length (VL) support



intel – AVX512
512 bits per vector (8 DP elements)
arm – SVE
Up to 2048 bits per vector (32 DP elements)
NEC / RISC-V
16384 bits per vector (256 DP elements)

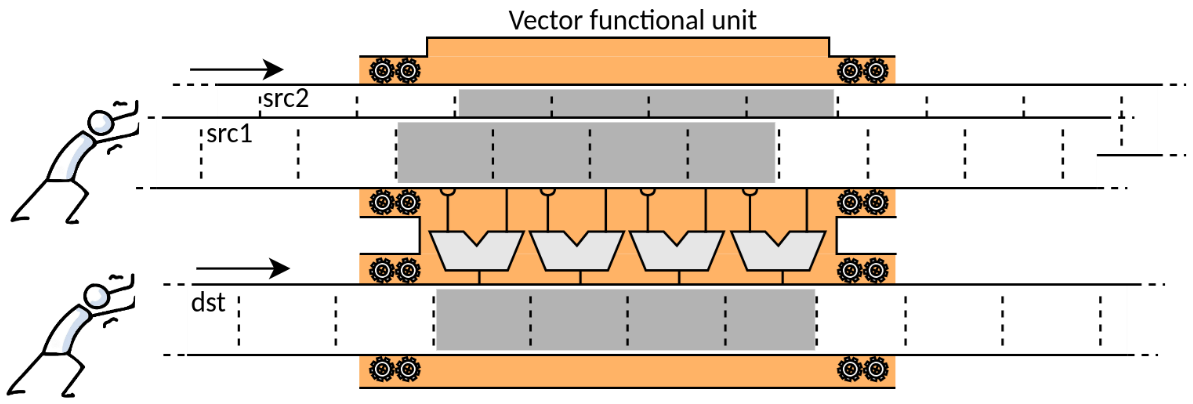
## Short VL

- As many Functional Units as VL.
- Vector instructions executed in 1 cycle



## Long VL

- Cannot afford (area, power, cost) hundreds of Functional Units
- Vector instructions are executed on multiple cycles



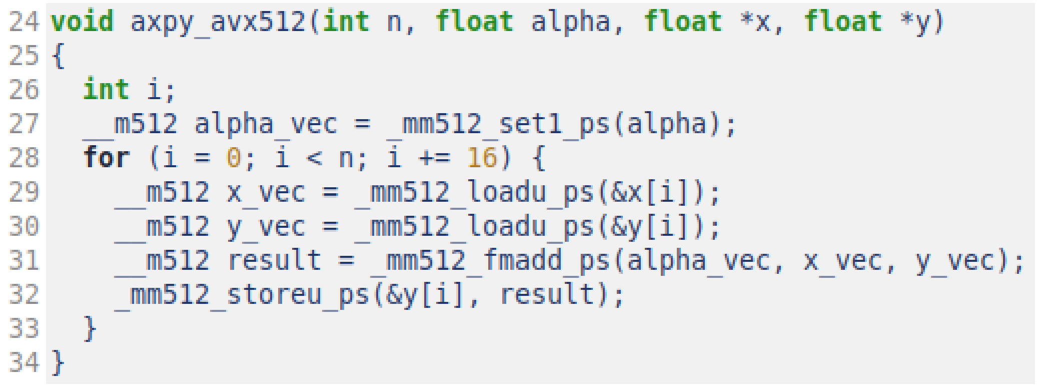
# An example: AXPY with x86 intrinsics

```c
void axpy(int n, float alpha, float *x, float *y)
{
  for (int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

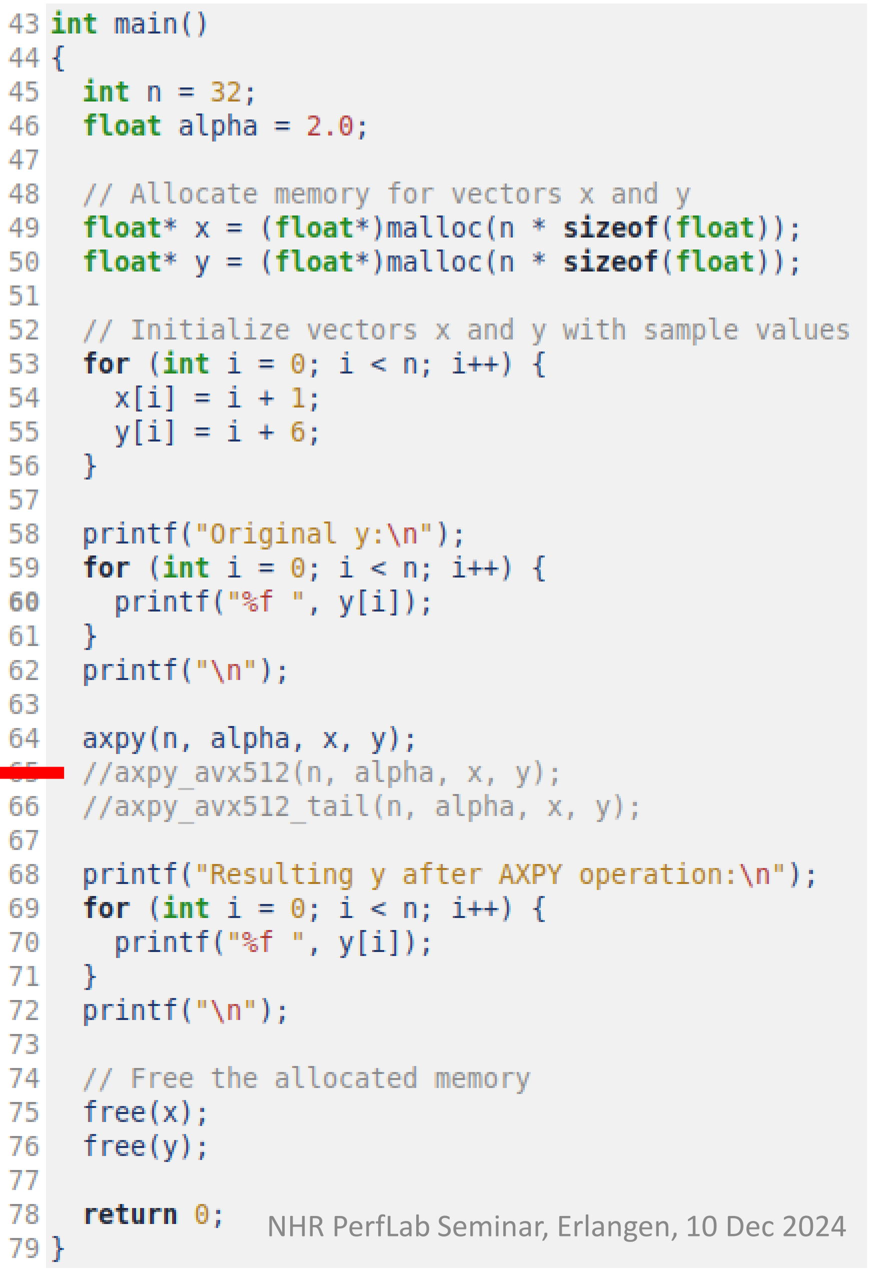
```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# An example: AXPY with x86 intrinsics



```c
void axpy_avx512(int n, float alpha, float *x, float *y)
{
  int i;
  __m512 alpha_vec = _mm512_set1_ps(alpha);
  for (i = 0; i < n; i += 16) {
    __m512 x_vec = _mm512_loadu_ps(&x[i]);
    __m512 y_vec = _mm512_loadu_ps(&y[i]);
    __m512 result = _mm512_fmadd_ps(alpha_vec, x_vec, y_vec);
    _mm512_storeu_ps(&y[i], result);
  }
}
```



```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

# An example: AXPY with x86 intrinsics

For a generic size of X and Y, we must handle “loop tails”

```c
int main()
{
  int n = 32;
  float alpha = 2.0;

  // Allocate memory for vectors x and y
  float* x = (float*)malloc(n * sizeof(float));
  float* y = (float*)malloc(n * sizeof(float));

  // Initialize vectors x and y with sample values
  for (int i = 0; i < n; i++) {
    x[i] = i + 1;
    y[i] = i + 6;
  }

  printf("Original y:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  axpy(n, alpha, x, y);
  //axpy_avx512(n, alpha, x, y);
  //axpy_avx512_tail(n, alpha, x, y);

  printf("Resulting y after AXPY operation:\n");
  for (int i = 0; i < n; i++) {
    printf("%f ", y[i]);
  }
  printf("\n");

  // Free the allocated memory
  free(x);
  free(y);

  return 0;
}
```

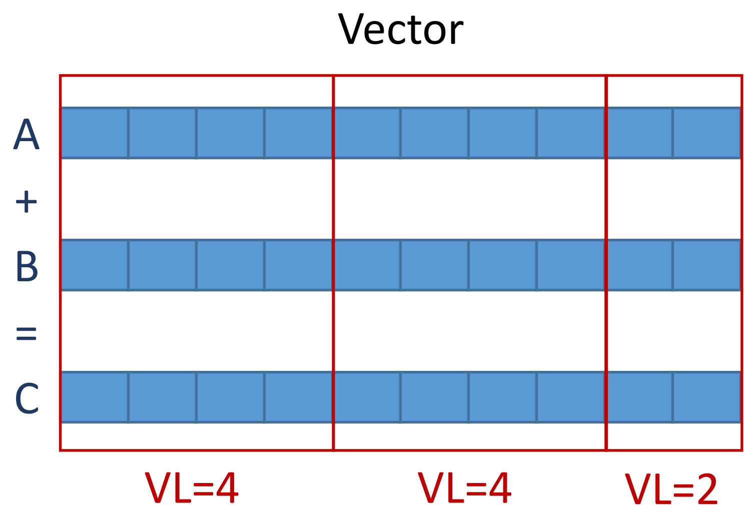
```c
void axpy_avx512_tail(int n, float alpha, float *x, float *y)
{
  int i;
  __m512 alpha_vec = _mm512_set1_ps(alpha);
  int avx512_loop_size = n - (n % 16);

  for (i = 0; i < avx512_loop_size; i += 16) {
    __m512 x_vec = _mm512_loadu_ps(&x[i]);
    __m512 y_vec = _mm512_loadu_ps(&y[i]);
    __m512 result = _mm512_fmadd_ps(alpha_vec, x_vec, y_vec);
    _mm512_storeu_ps(&y[i], result);
  }

  for (; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

# A bit more elegant: Variable Vector Length

⇨ Vector length (VL) register limits the max number of elements to be processed by a vector instruction

- VL is loaded prior to executing the vector instruction with a special instruction
- No need to handle “loop tails”
- Makes the code “vector length agnostic”

```c
void axpy(double a, double *dx, double *dy, int n) {
  int i;

  long gvl = __builtin_epi_vsetvl(n, __epi_e64, __epi_m1);
  __epi_1xf64 v_a = _MM_SET_f64(a, gvl);

  for (i = 0; i < n; i += gvl) {
    gvl = __builtin_epi_vsetvl(n - i, __epi_e64, __epi_m1);
    __epi_1xf64 v_dx = _MM_LOAD_f64(&dx[i], gvl);
    __epi_1xf64 v_dy = _MM_LOAD_f64(&dy[i], gvl);
    __epi_1xf64 v_res = _MM_MACC_f64(v_dy, v_a, v_dx, gvl);
    _MM_STORE_f64(&dy[i], v_res, gvl);
  }
```

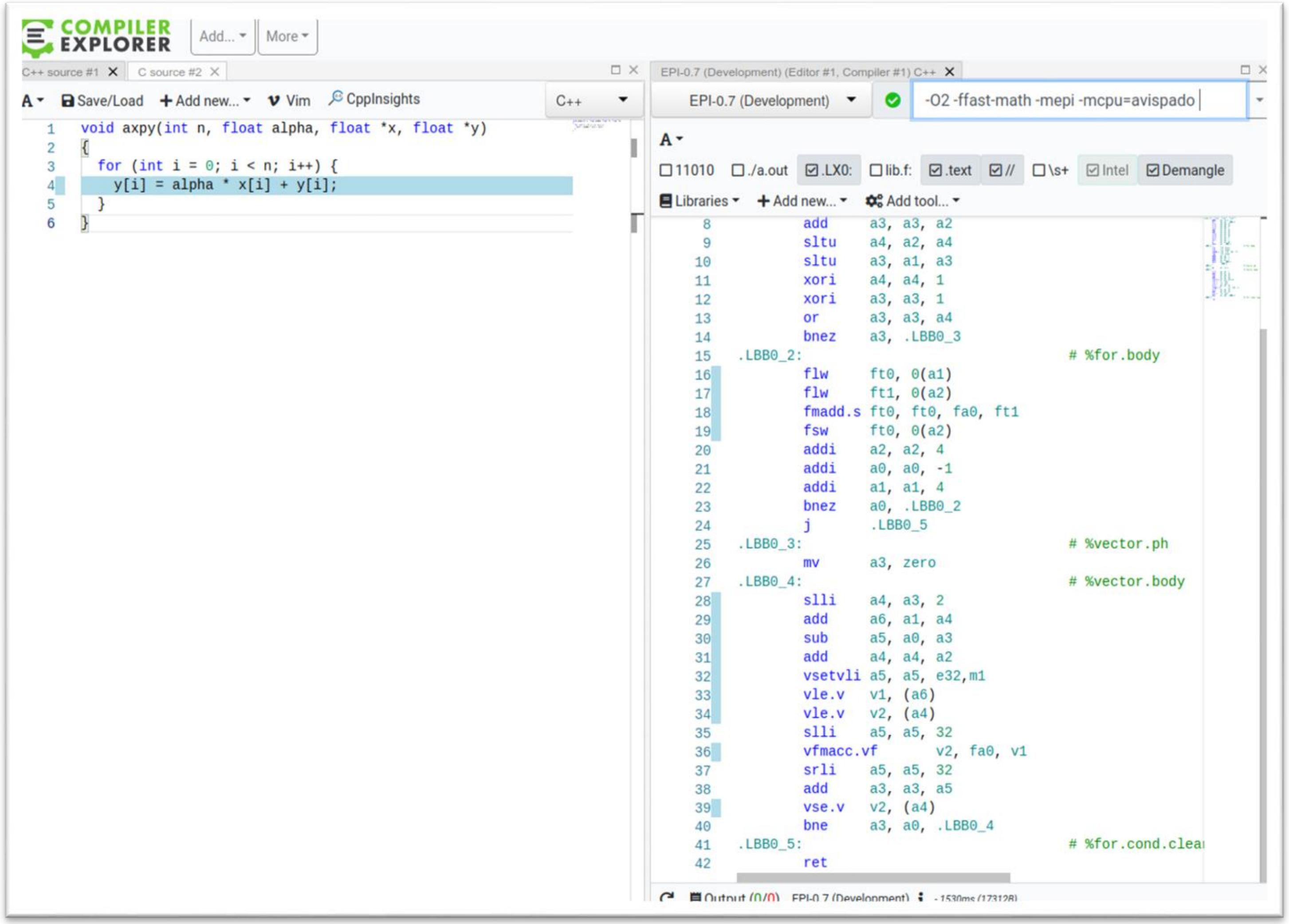


VL can have any value < VL_max
It does not work only with intrinsics

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Try it yourself

- ⇨ "Compiler Explorer" allows developers to write and compile code in various programming languages, including C++, C, Rust, and others.
- ⇨ Web-based interface for quickly testing and experimenting with code snippets, especially in the context of compiler optimizations.
- ⇨ https://repo.hca.bsc.es/epic/



Compiler: EPI-0.7 (Development), options: `-O2 -ffast-math -mepi -mcpu=avispado`

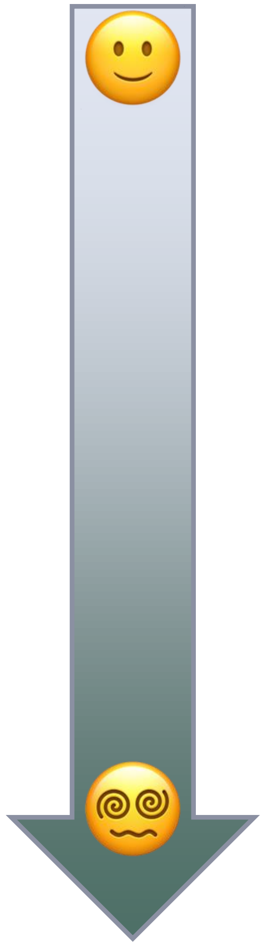
```c
void axpy(int n, float alpha, float *x, float *y)
{
  for (int i = 0; i < n; i++) {
    y[i] = alpha * x[i] + y[i];
  }
}
```

```
        add     a3, a3, a2
        sltu    a4, a2, a4
        sltu    a3, a1, a3
        xori    a4, a4, 1
        xori    a3, a3, 1
        or      a3, a3, a4
        bnez    a3, .LBB0_3
.LBB0_2:                                # %for.body
        flw     ft0, 0(a1)
        flw     ft1, 0(a2)
        fmadd.s ft0, ft0, fa0, ft1
        fsw     ft0, 0(a2)
        addi    a2, a2, 4
        addi    a0, a0, -1
        addi    a1, a1, 4
        bnez    a0, .LBB0_2
        j       .LBB0_5
.LBB0_3:                                # %vector.ph
        mv      a3, zero
.LBB0_4:                                # %vector.body
        slli    a4, a3, 2
        add     a6, a1, a4
        sub     a5, a0, a3
        add     a4, a4, a2
        vsetvli a5, a5, e32,m1
        vle.v   v1, (a6)
        vle.v   v2, (a4)
        slli    a5, a5, 32
        vfmacc.vf       v2, fa0, v1
        srli    a5, a5, 32
        add     a3, a3, a5
        vse.v   v2, (a4)
        bne     a3, a0, .LBB0_4
.LBB0_5:                                # %for.cond.clea
        ret
```

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

https://www.microcontrollertips.com/risc-v-vs-arm-vs-x86-whats-the-difference/

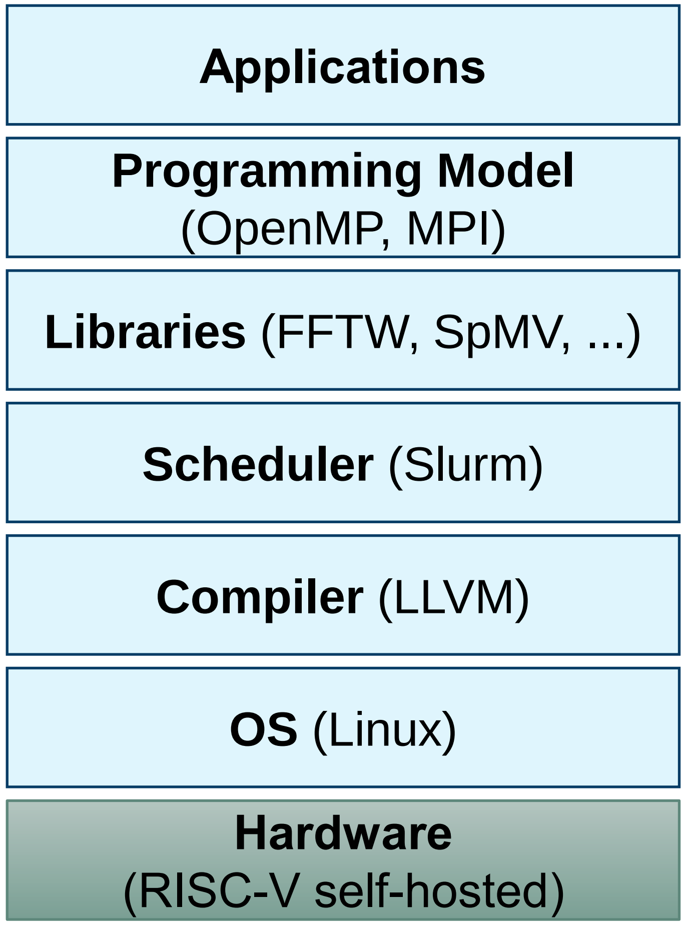
## How do I program EPAC - VEC?



- **Autovectorization**
  - Leave it to the compiler
- **#pragma omp simd** (aka “Guided vectorization”)
  - Relies on vectorization capabilities of the compiler
    - Usually works but gets complicated if the code calls functions
  - Also usable in Fortran
- **C/C++ builtins** (aka “Intrinsics”)
  - Low-level mapping to the instructions
  - Allows embedding it into an existing C/C++ codebase
  - Allows relatively quick experimentation
- **Assembler**
  - Always a valid option but not the most pleasant

## How do I use EPAC - VEC?

- Like a standard HPC system!
- Compile your code
  - We give you a compiler
- Link libraries
- Write/Submit a job script
  - SLURM
- Wait for the results
- Analyse execution traces and study how well your code is vectorized


**Applications**
**Programming Model** (OpenMP, MPI)
**Libraries** (FFTW, SpMV, ...)
**Scheduler** (Slurm)
**Compiler** (LLVM)
**OS** (Linux)
**Hardware** (RISC-V self-hosted)

## Take home message

⇨**EPI is developing:**

- Arm-based CPU (not part of this talk/workshop)
- RISC-V-based Accelerator

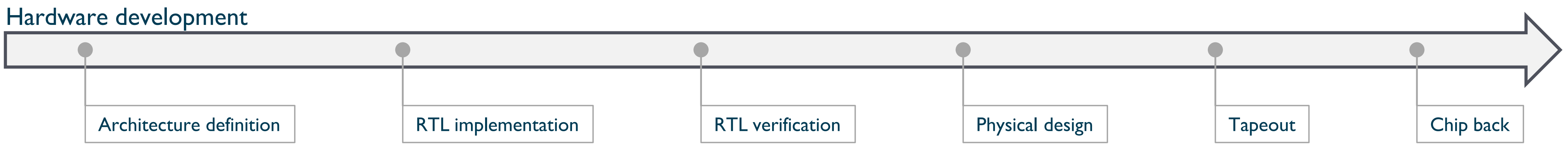
⇨**We focus on the RISC-V vector accelerator (VEC) that:**

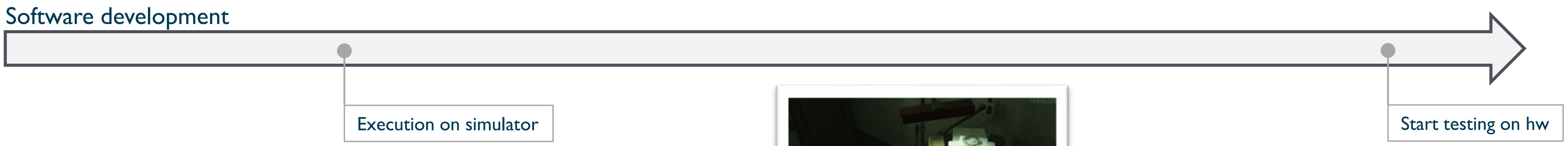
- Can be self-hosted
- Support variable vector length
- Is vector length agnostic
- Uses long vectors (256 DP elements, 32x larger than x86)

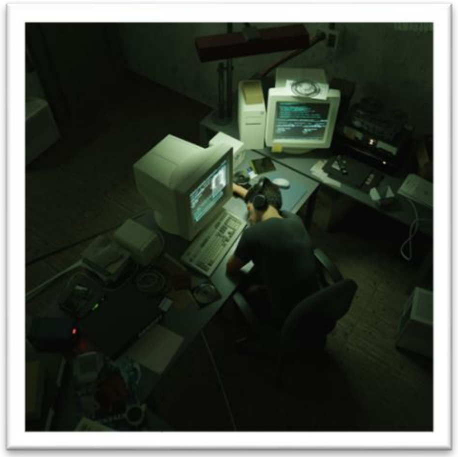
European
Processor
Initiative

## Software Development Vehicles (SDV)

# What to do until the hardware is ready?

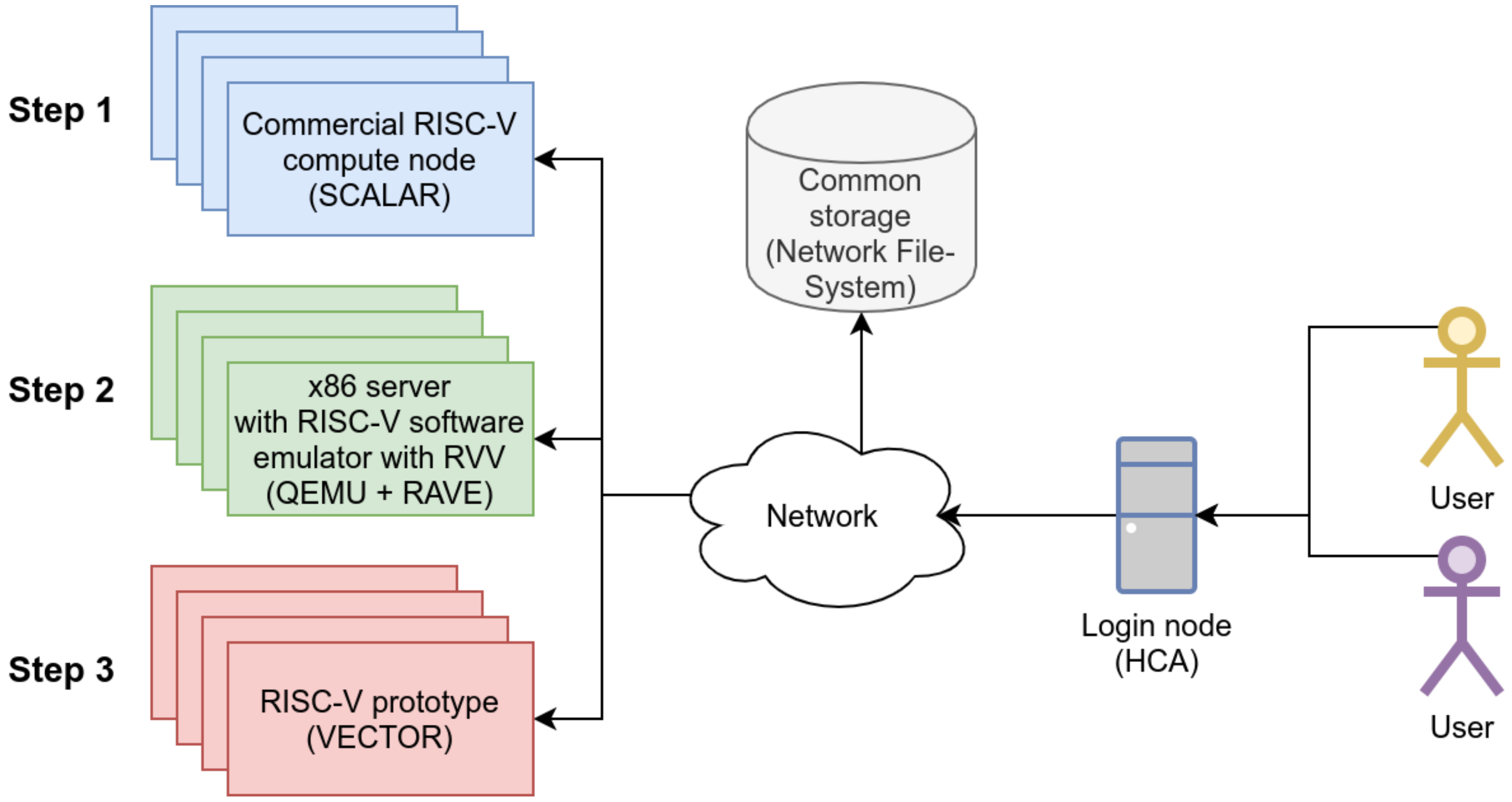




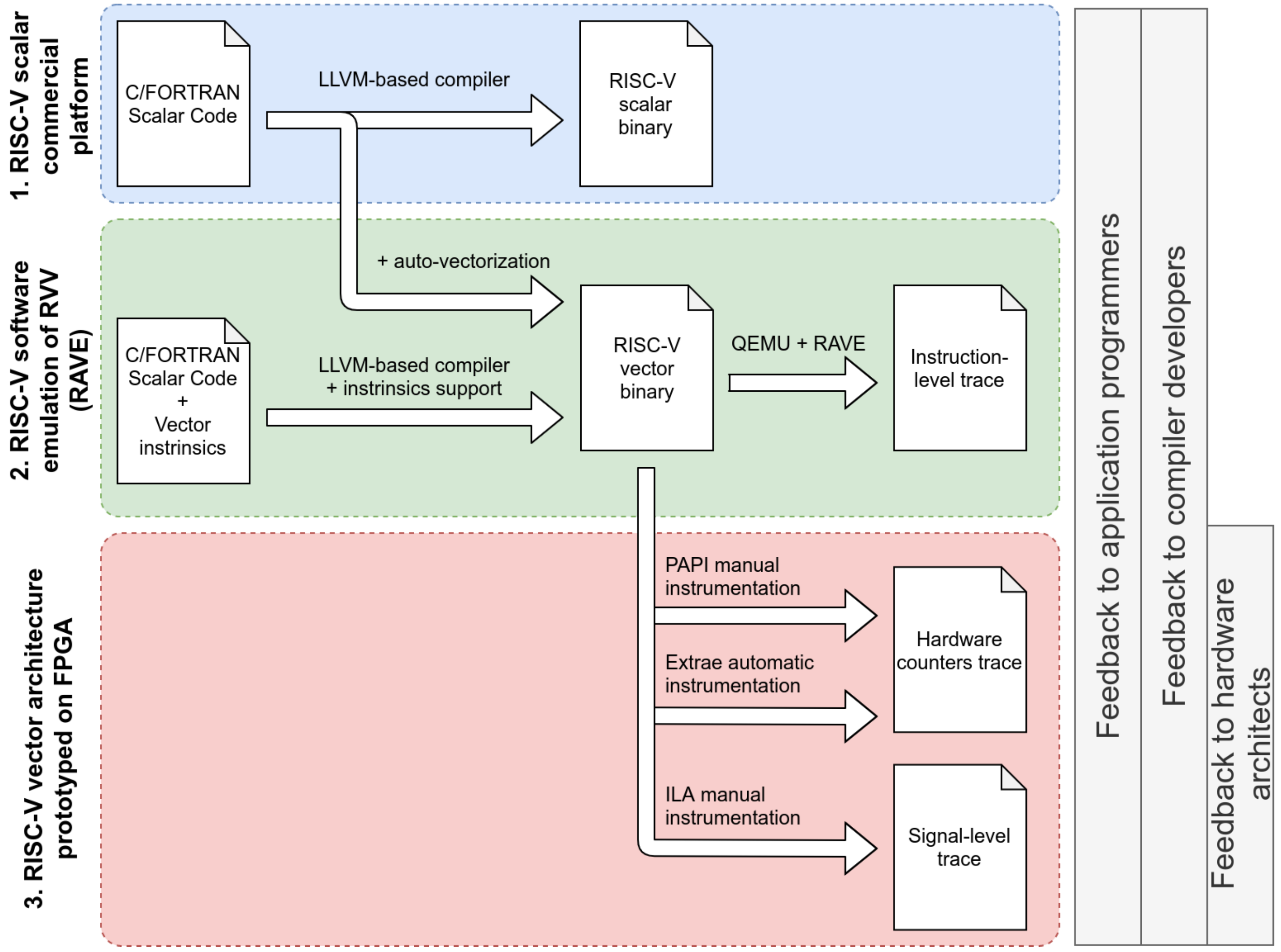


Wake up Neo…
Follow the Software Development Vehicles

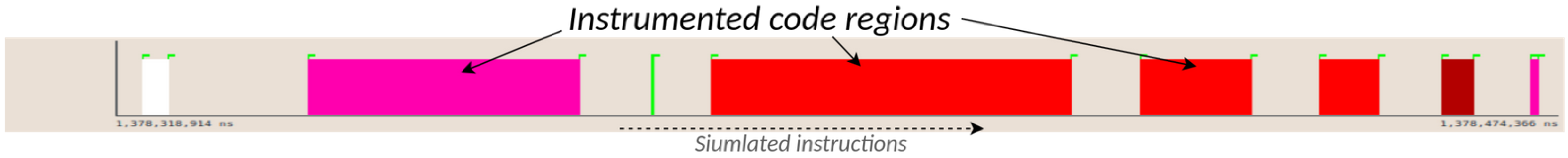
# Software Development Vehicles (SDV)

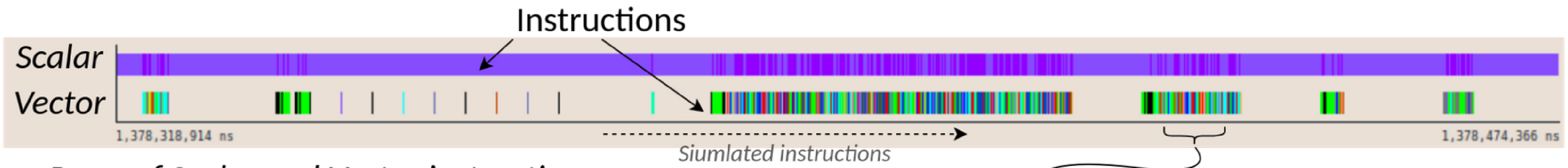

Step 1
Commercial RISC-V compute node (SCALAR)
Step 2
x86 server with RISC-V software emulator with RVV (QEMU + RAVE)
Step 3
RISC-V prototype (VECTOR)
Common storage (Network File-System)
Network
Login node (HCA)
User
User

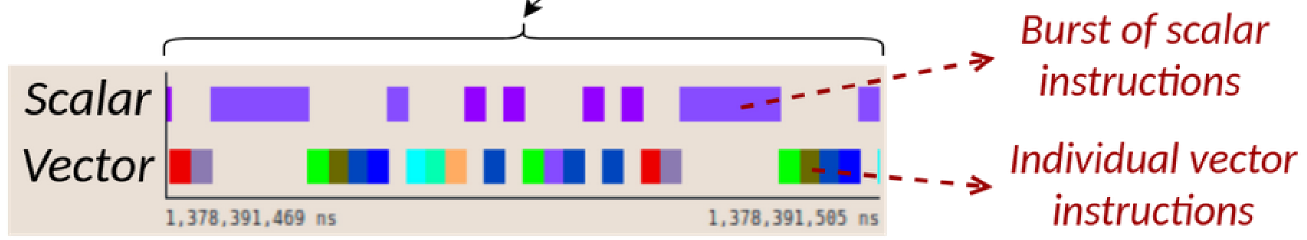
# Co-design with SDV


1. RISC-V scalar commercial platform
2. RISC-V software emulation of RVV (RAVE)
3. RISC-V vector architecture prototyped on FPGA
C/FORTRAN Scalar Code
LLVM-based compiler
RISC-V scalar binary
+ auto-vectorization
C/FORTRAN Scalar Code + Vector instrinsics
LLVM-based compiler + instrinsics support
RISC-V vector binary
QEMU + RAVE
Instruction-level trace
PAPI manual instrumentation
Extrae automatic instrumentation
Hardware counters trace
ILA manual instrumentation
Signal-level trace
Feedback to application programmers
Feedback to compiler developers
Feedback to hardware architects

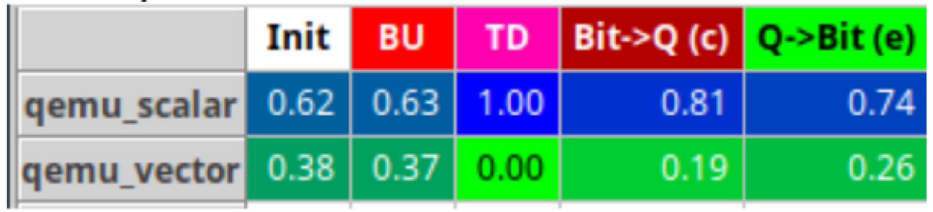
# Navigate, visualize and quantify





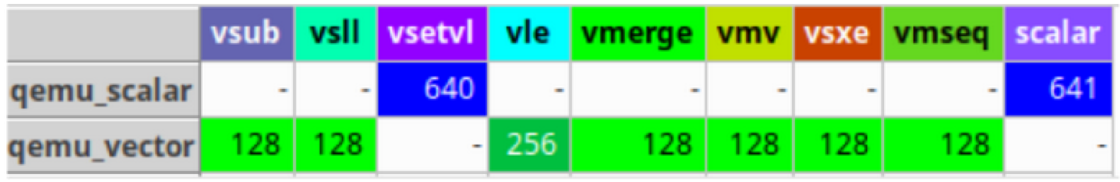


*Prop. of Scalar and Vector instructions*



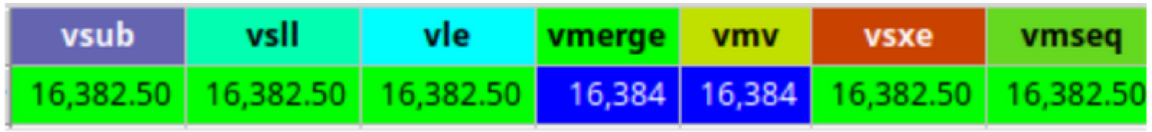
| | Init | BU | TD | Bit->Q (c) | Q->Bit (e) |
|---|---|---|---|---|---|
| qemu_scalar | 0.62 | 0.63 | 1.00 | 0.81 | 0.74 |
| qemu_vector | 0.38 | 0.37 | 0.00 | 0.19 | 0.26 |

*Assembly of instructions (In phase Init)*



| | vsub | vsll | vsetvl | vle | vmerge | vmv | vsxe | vmseq | scalar |
|---|---|---|---|---|---|---|---|---|---|
| qemu_scalar | - | - | 640 | - | - | - | - | - | 641 |
| qemu_vector | 128 | 128 | - | 256 | 128 | 128 | 128 | 128 | - |

*Average bits per instruction (In phase Init)*

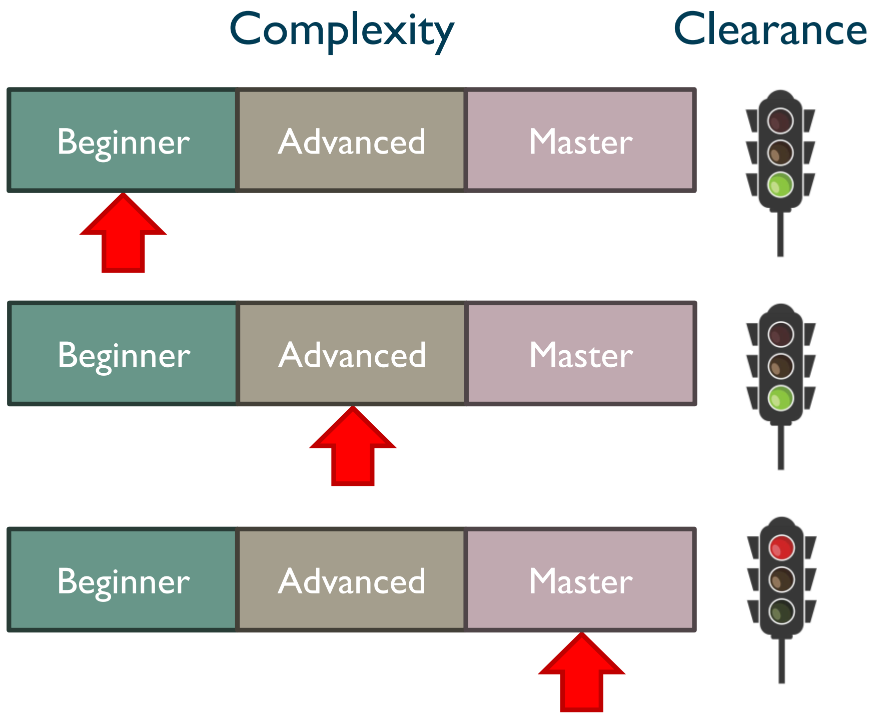


| vsub | vsll | vle | vmerge | vmv | vsxe | vmseq |
|---|---|---|---|---|---|---|
| 16,382.50 | 16,382.50 | 16,382.50 | 16,384 | 16,384 | 16,382.50 | 16,382.50 |

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

# Software Development Vehicles (SDV)

- 3 Steps:
  - **1st step**: Run in a commercial RISC-V platform (scalar CPU)
  - **2nd step**: RISC-V software emulation supporting RVV (RAVE)
  - **3rd step**: Run on VEC mapped into FPGA



## Take home message

⇨While RTL is becoming actual hardware, EPI develops tools for boosting the co-design cycle

- Software and Hardware prototypes (aka Software Development Vehicles)

⇨We can leverage SDVs to:

- Influence hardware design
- Improve compiler autovectorization and system-software support
- Study and improve vectorization of real scientific HPC codes

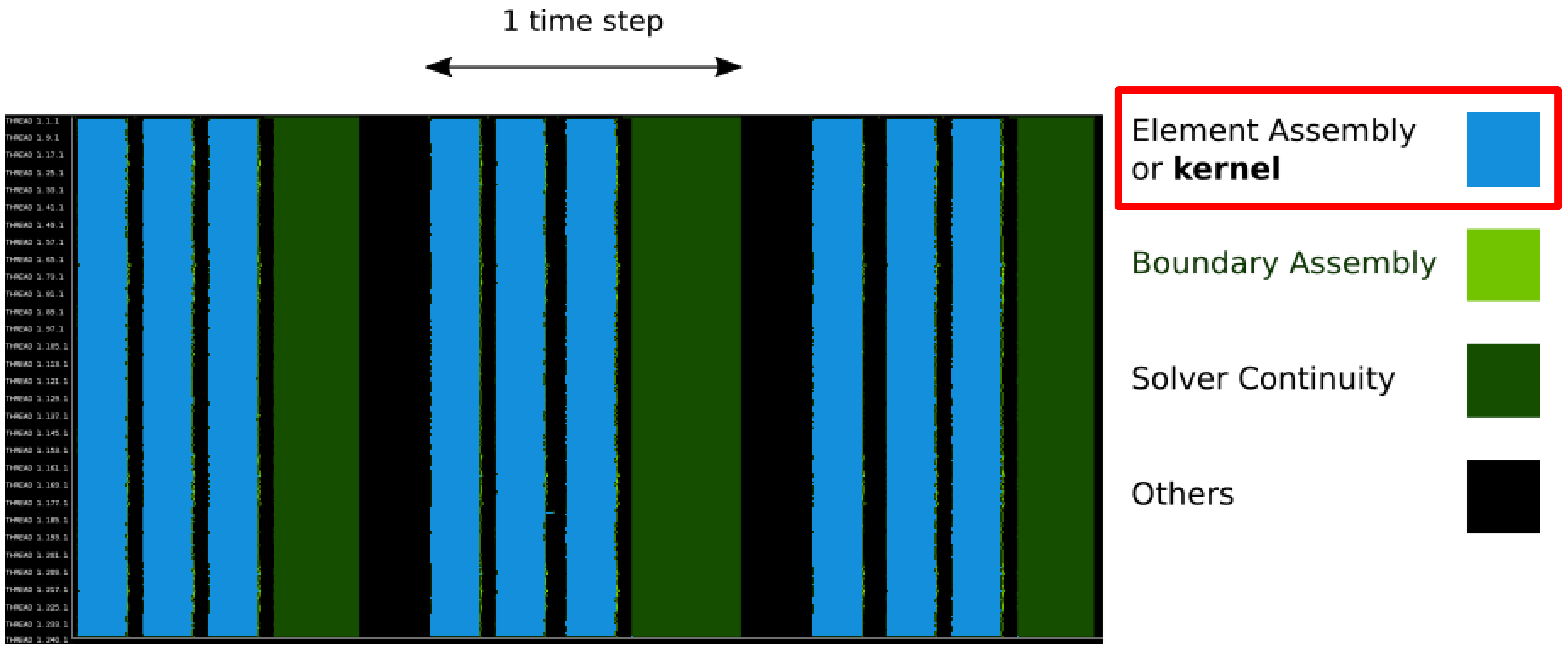
European Processor Initiative

Centre of Excellence in Exascale CFD

## Vectorization of a CFD code

# Vectorization of a real CFD code (Alya)

- Alya is a modular code → We study the module called “Nastin”
- “VECTOR_SIZE”
  - Allocates data structures in a vector-friendly way
  - Values under study → [16, 64, 128, 240, 256, 512]



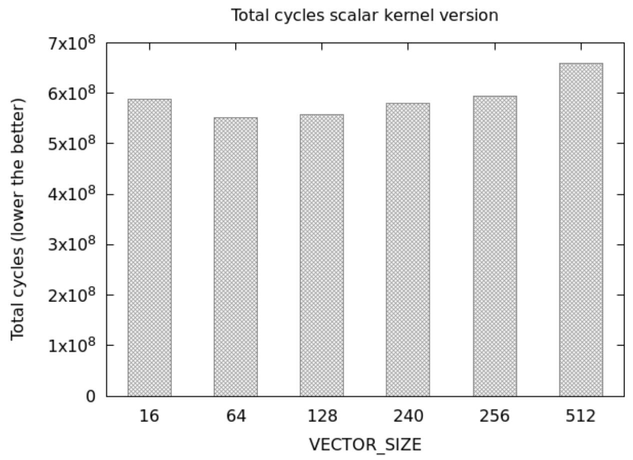
## Alya mini-app

- We worked on a mini-app that mimics the behaviour of the Assembly of Alya
- We divided the mini-app in "phases"
  - Mini-app phases are regions of codes with one or more loops
  - We are interested in loops because is where there is potential for vectorization
  - 8 phases identified: P1+P2+P3+P4+P5+P6+P7+P8 = mini-app
- We based our study and optimization on the autovectorization capabilities
  - No intrinsics → portability is preserved

# 1st step: Run on commercial RISC-V platforms (scalar CPU)

| Phase | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| % of total cycles | 1,29% | 3,33% | 19,80% | 14,45% | 3,49% | 40,99% | 14,68% | 1,96% |

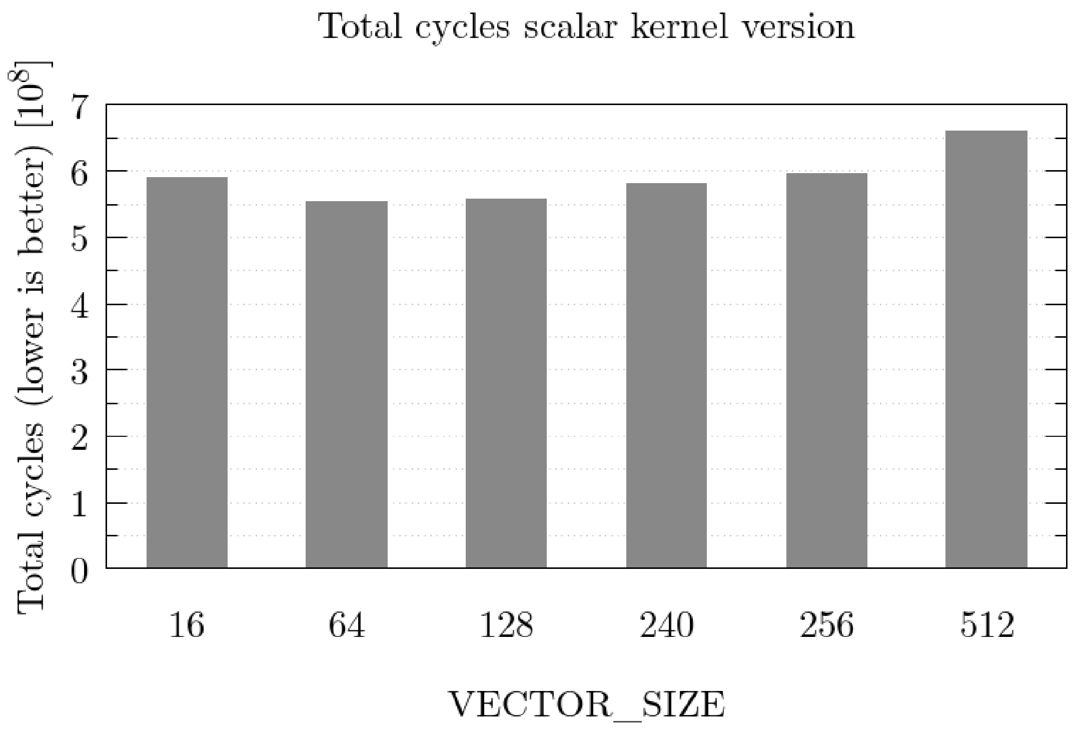
- Phases taking longer (6,3,7,4) correspond to compute intensive regions
- Phases lasting less (5,2,8,1) are memory bound regions
- VECTOR_SIZE parameter has almost no influence on the execution (5% coefficient of variation)



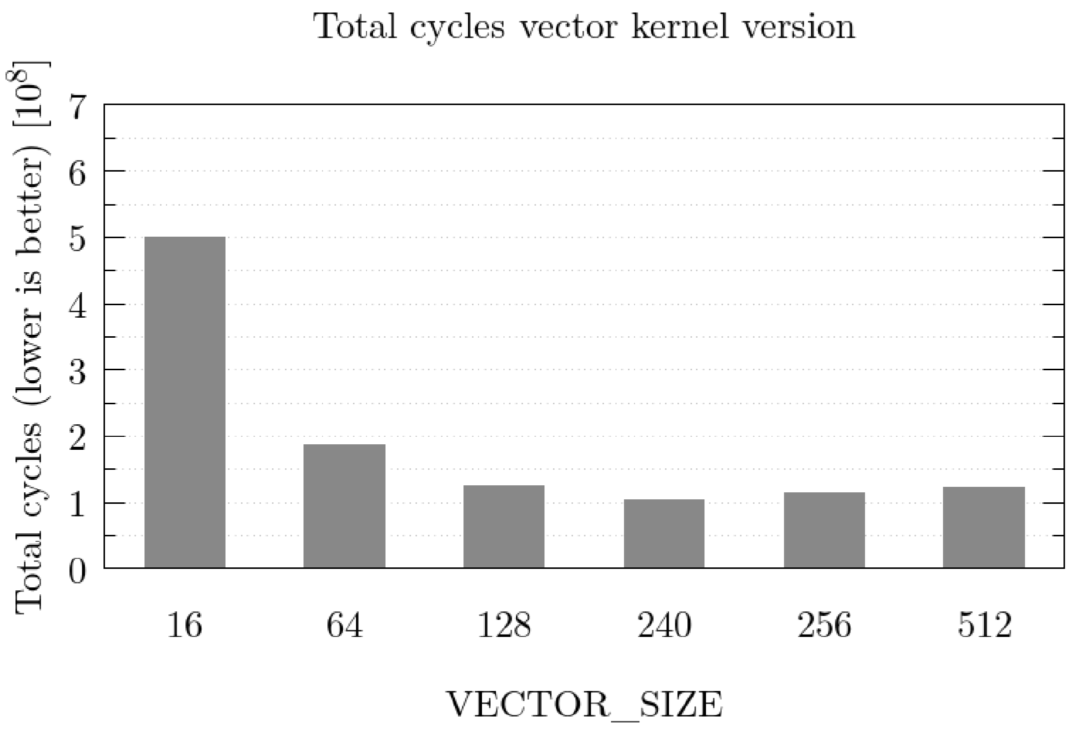
Commercial RISC-V platform (scalar CPU)

## 1st step: Enabling auto-vectorization

- Auto-vectorization results without touching any line of code
- VECTOR_SIZE parameter strongly influences when executing with vectors



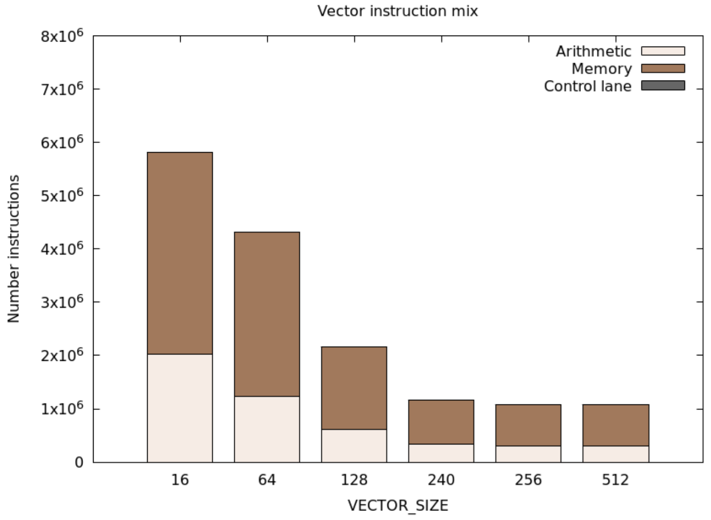
Enabling Compiler Auto-vec



# 2nd step: Emulation supporting RVV (RAVE)

|  | Phase |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| VECTOR_SIZE | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 16 | 0,00% | 0,00% | 1,84% | 0,00% | 0,00% | 0,95% | 24,64% | 0,00% |
| 64 | 0,00% | 0,00% | 12,73% | 17,37% | 17,86% | 21,58% | 25,87% | 0,00% |
| 128 | 0,00% | 0,00% | 16,05% | 16,80% | 17,94% | 20,39% | 25,23% | 0,00% |
| 240 | 0,00% | 0,00% | 15,31% | 16,45% | 16,82% | 19,90% | 23,90% | 0,00% |
| 256 | 0,00% | 0,00% | 15,36% | 16,21% | 15,88% | 19,78% | 24,23% | 0,00% |
| 512 | 0,00% | 0,00% | 16,65% | 18,19% | 18,47% | 21,82% | 26,20% | 0,00% |

|  |
|---|
| 🟩 30,00% |
| 🟨 15,00% |
| 🟥 0,00% |



Analysis of % of vector instructions:

- Higher VECTOR_SIZE helps the compiler to insert more vector instructions
- Higher VECTOR_SIZE reduces the total number of vector instructions
- 70% of vector instructions are memory type

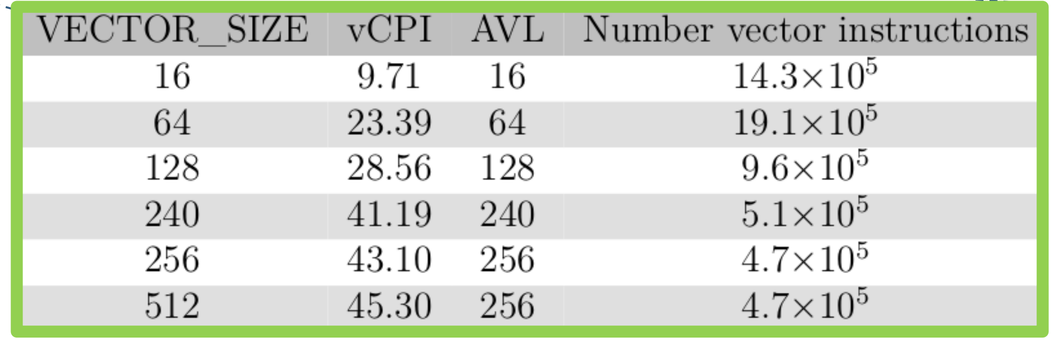
# 3rd step: Run on VEC mapped into FPGA

|  | Phase |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| VECTOR_SIZE | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 16 | 0,00% | 0,00% | 15,72% | 0,00% | 0,00% | 7,66% | 73,30% | 0,00% |
| 64 | 0,00% | 0,00% | 72,59% | 76,62% | 57,73% | 86,85% | 77,70% | 0,00% |
| 128 | 0,00% | 0,00% | 81,94% | 79,36% | 64,01% | 88,96% | 79,59% | 0,00% |
| 240 | 0,00% | 0,00% | 83,69% | 83,08% | 70,75% | 90,61% | 81,94% | 0,00% |
| 256 | 0,00% | 0,00% | 83,76% | 83,03% | 71,29% | 90,26% | 82,83% | 0,00% |
| 512 | 0,00% | 0,00% | 85,74% | 87,59% | 80,61% | 91,14% | 88,50% | 0,00% |

|  |  |
|---|---|
| 🟩 | 100,00% |
| 🟨 | 75,00% |
| 🟧 | 50,00% |
| 🟧 | 25,00% |
| 🟥 | 0,00% |

Analysis of % of vector cycles:

- High vCPI → we are computing several elements per instruction (GOOD)
- AVL == VECTOR_SIZE → the more elements we process per vector instruction, the less vector instructions we execute (GOOD)



| VECTOR_SIZE | vCPI | AVL | Number vector instructions |
|---|---|---|---|
| 16 | 9.71 | 16 | $14.3\times10^5$ |
| 64 | 23.39 | 64 | $19.1\times10^5$ |
| 128 | 28.56 | 128 | $9.6\times10^5$ |
| 240 | 41.19 | 240 | $5.1\times10^5$ |
| 256 | 43.10 | 256 | $4.7\times10^5$ |
| 512 | 45.30 | 256 | $4.7\times10^5$ |

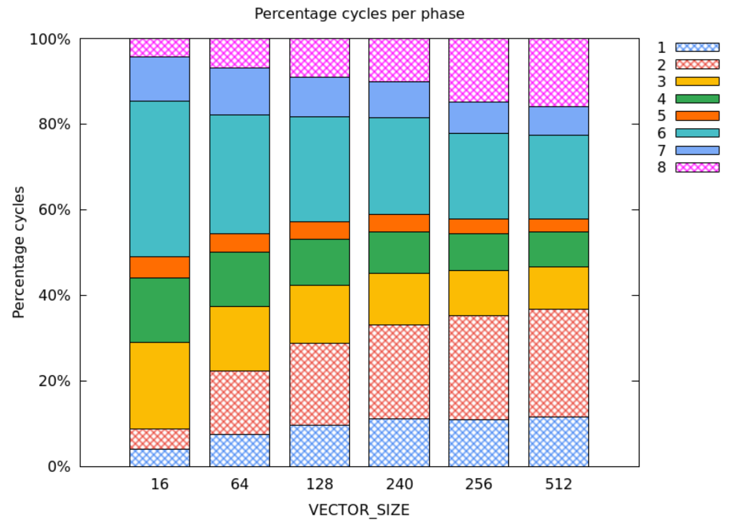
vCPI, AVL and # vector instructions phase 6

# 3rd step: Run on VEC mapped into FPGA

|  | Phase |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| VECTOR_SIZE | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 16 | 0,00% | 0,00% | 15,72% | 0,00% | 0,00% | 7,66% | 73,30% | 0,00% |
| 64 | 0,00% | 0,00% | 72,59% | 76,62% | 57,73% | 86,85% | 77,70% | 0,00% |
| 128 | 0,00% | 0,00% | 81,94% | 79,36% | 64,01% | 88,96% | 79,59% | 0,00% |
| 240 | 0,00% | 0,00% | 83,69% | 83,08% | 70,75% | 90,61% | 81,94% | 0,00% |
| 256 | 0,00% | 0,00% | 83,76% | 83,03% | 71,29% | 90,26% | 82,83% | 0,00% |
| 512 | 0,00% | 0,00% | 85,74% | 87,59% | 80,61% | 91,14% | 88,50% | 0,00% |

| |
|---|
| 100,00% |
| 75,00% |
| 50,00% |
| 25,00% |
| 0,00% |

- Phases 1, 2 and 8 are not vectorized (pattern colored in plot)
- Next step: focus in vectorize phase 2
  - Costing 30% of time ⚠



## Example of optimization: phase 2 aka VEC2

European Processor Initiative

**Problem**

- Compiler unable to vectorize loop, not sure of VECTOR_DIM value

```
subroutine nsi_miniapp(VECTOR_DIM, pnode, pgaus, list_elements)
```

```
loop not vectorized: unsafe dependent memory operations in loop.
```

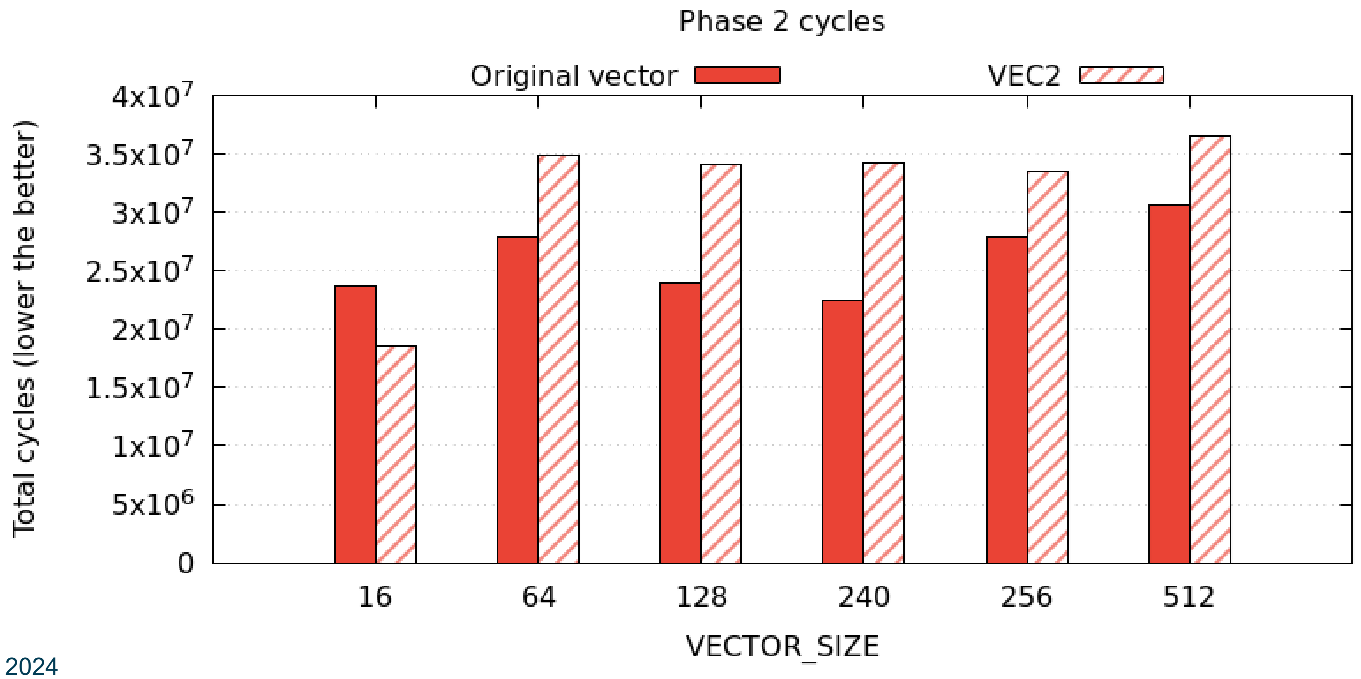
**Solution**

- We know VECTOR_DIM value

```
integer(ip), parameter :: VECTOR_DIM = VECTOR_SIZE
```

# Optimization - VEC2

- Enabled vectorization in phase 2
  - Performance get worst instead of improving
  - AVL of vector instructions is low! ⚠
    We are not taking advantage of the full-VL. Why?



## Optimization - VEC2+VL

**Problem**

- pnode comes from input, we do not know its value
- Experimentally found pnode << VECTOR_DIM

```
do ivect = 1,VECTOR_DIM
  do inode = 1,pnode
    !WORK
  end do
end do
```

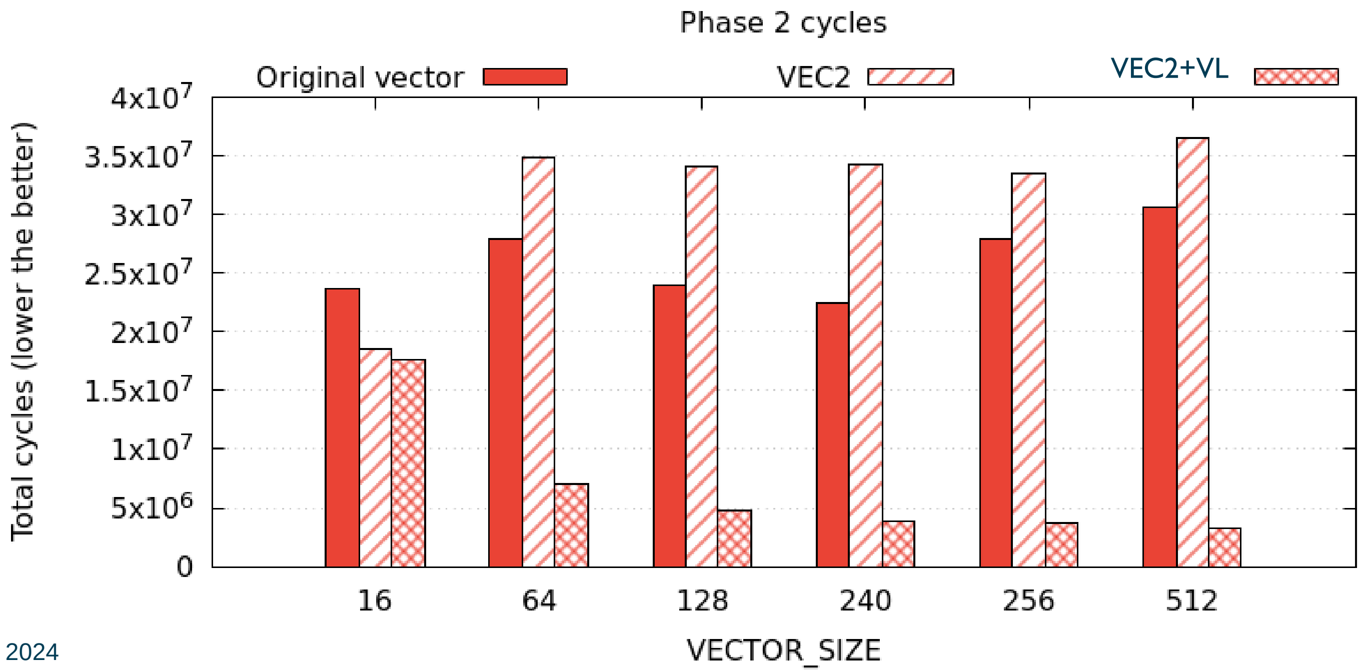
**Solution**

- Swap induction variables

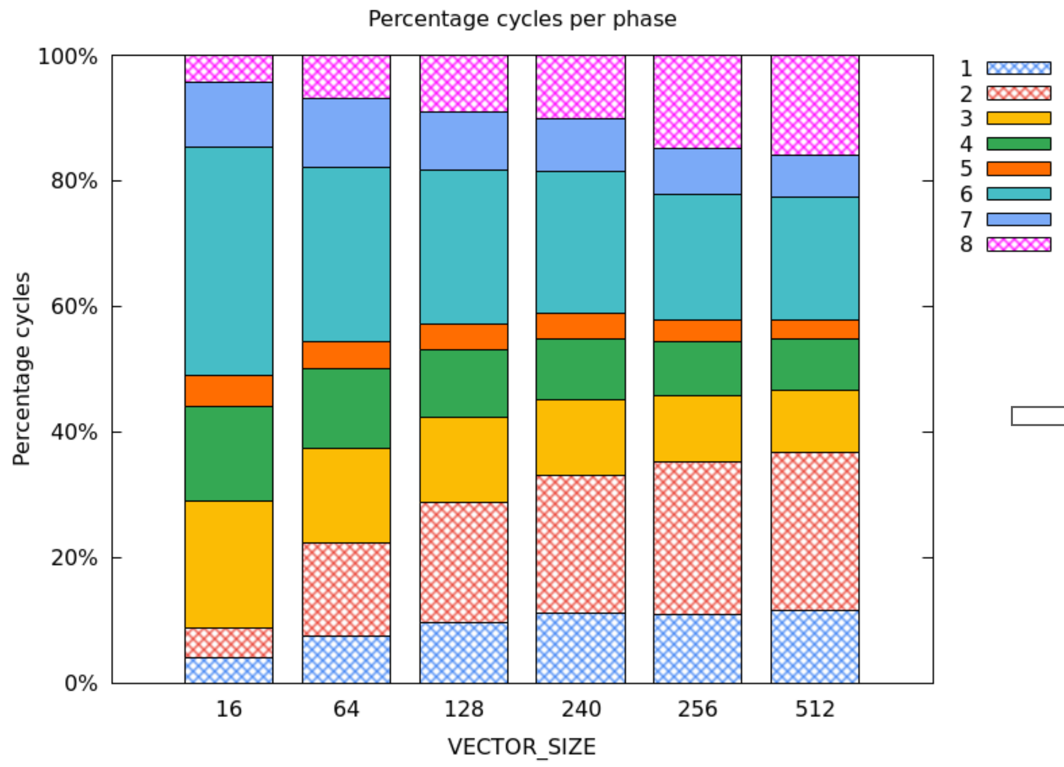
```
do inode = 1,pnode
  do ivect = 1,VECTOR_DIM
    !WORK
  end do
end do
```

## Optimization VEC2+VL: results

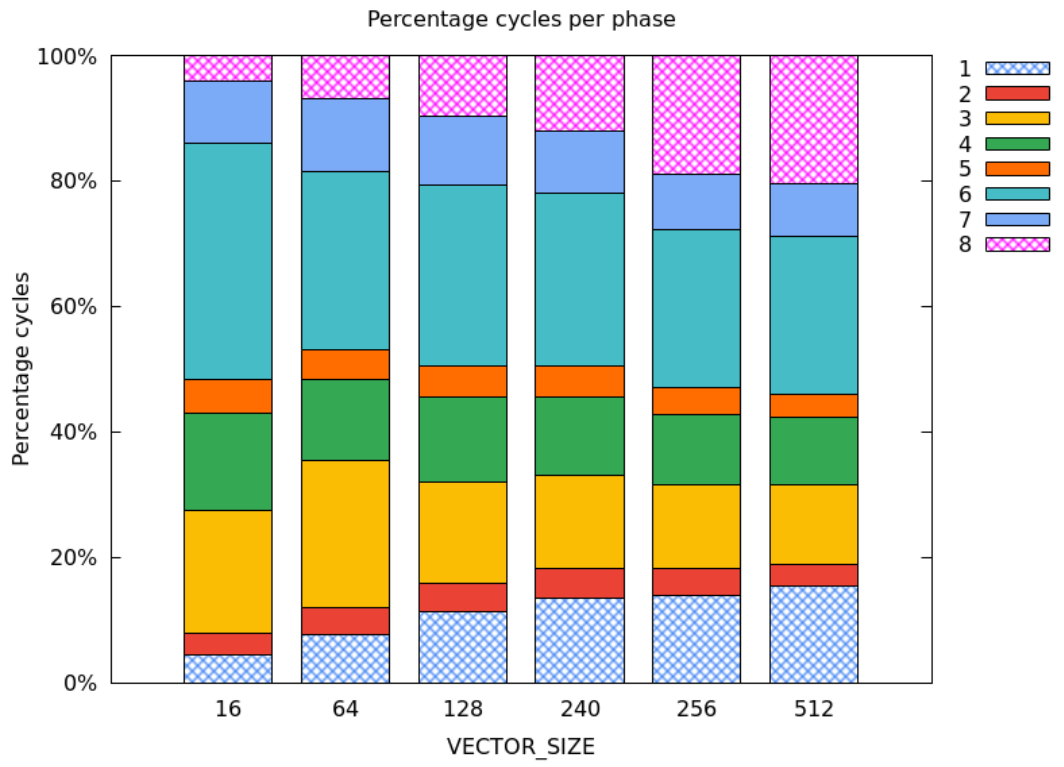
- Improved AVL vectorization in phase 2
  - Vector instructions running with AVL == VECTOR_SIZE



# Alya preliminary results - VEC2+VL



Before optimization

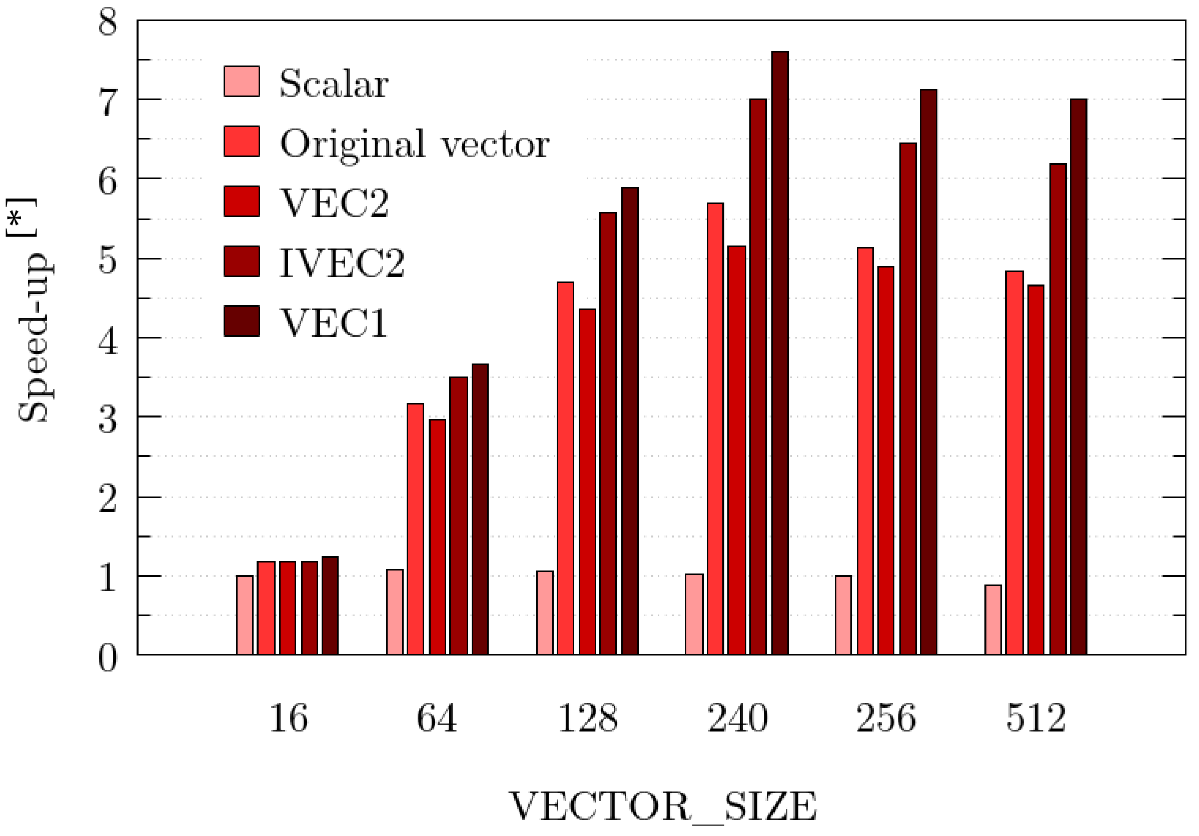


After optimization

# Evaluation: RISC-V vector prototype

- After a detailed study and manual optimizations, we achieve a peak of 7.6x speedup (VEC1)

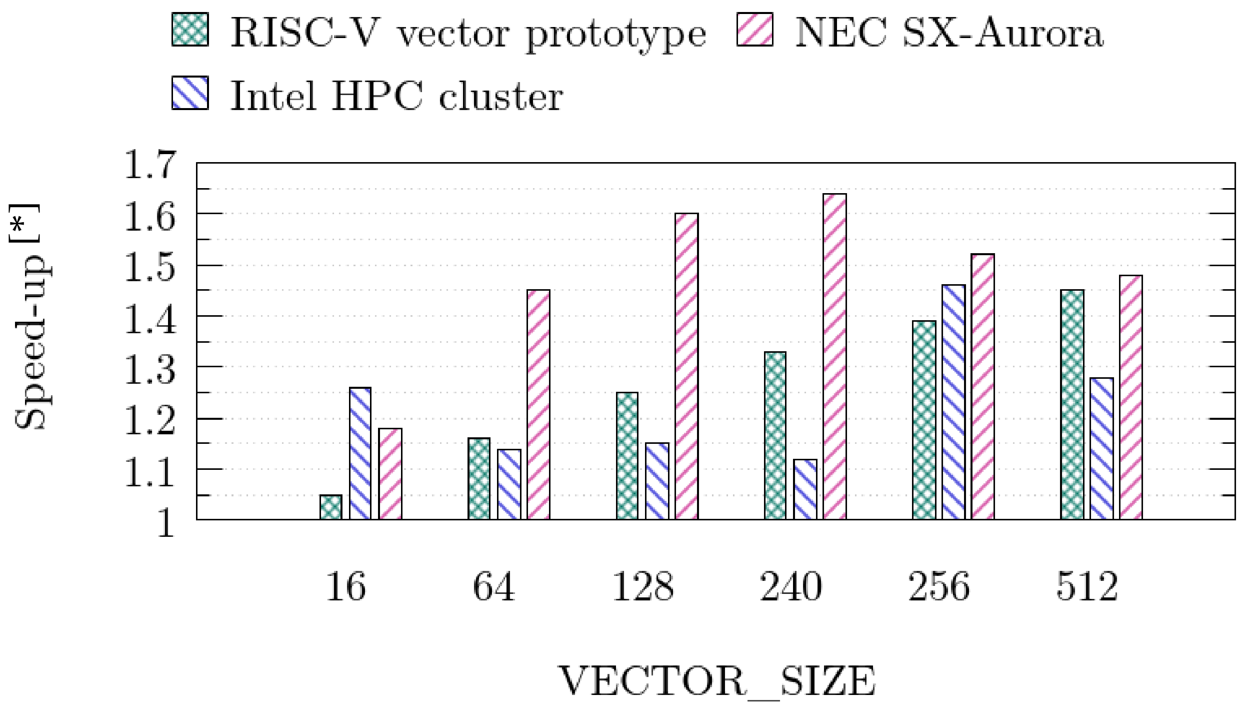
- Code remains portable
  No intrinsics!



[*] Speed-up defined as: scalar VECTOR_SIZE$_{16}$ / optimized vector

## Portability across other HPC platforms

- Optimizations portable to other architectures
  - “Traditional” cluster (Intel x86)
  - Long-vector architecture (NEC SX-Aurora)



[*] Speed-up defined as: vanilla vector / optimized vector

## Take home message

⇨ We leveraged the EPI Software Development Vehicles (SDVs) to study and improve vectorization of a complex CFD code (Alya) written in Fortran

⇨ Vectorization techniques improve performance on EPAC – VEC and are portable

⇨ Similar studies are on going for several scientific codes part of EU CoEs

CEEC
Centre of Excellence in Exascale CFD

PLASMA PEPSC

ChEESE

Blancafort, Marc, et al. "Exploiting long vectors with a CFD code: a co-design show case." 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2024.

## References

- 📝 Mantovani, Filippo, et al. "Software Development Vehicles to enable extended and early co-design: a RISC-V and HPC case of study." International Conference on High Performance Computing. Cham: Springer Nature Switzerland, 2023. https://arxiv.org/abs/2306.01797
- 📝 Vizcaino, Pablo, et al. "Short reasons for long vectors in HPC CPUs: a study based on RISC-V." Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis. 2023. https://arxiv.org/abs/2309.06865
- 📝 Vizcaino, Pablo, et al. "RAVE: RISC-V Analyzer of Vector Executions, a QEMU tracing plugin." arXiv preprint arXiv:2409.13639 (2024). https://arxiv.org/abs/2409.13639
- 📝 Blancafort, Marc, et al. "Exploiting long vectors with a CFD code: a co-design show case." 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2024. https://arxiv.org/abs/2411.00815

- 📰 https://www.eetimes.com/examining-the-top-five-fallacies-about-risc-v/
- 📽 https://www.youtube.com/watch?v=iFlcJFcOJKk

Google Scholar

## EPI FUNDING

This research has received funding from the European High Performance Computing Joint Undertaking (JU) under Framework Partnership Agreement No 800928 (European Processor Initiative) and Specific Grant Agreement No 101036168 (EPI SGA2). The JU receives support from the European Union's Horizon 2020 research and innovation programme and from Croatia, France, Germany, Greece, Italy, Netherlands, Portugal, Spain, Sweden, and Switzerland. The EPI-SGA2 project, PCI2022-132935 is also co-funded by MCIN/AEI /10.13039/501100011033 and by the UE NextGenerationEU/PRTR.