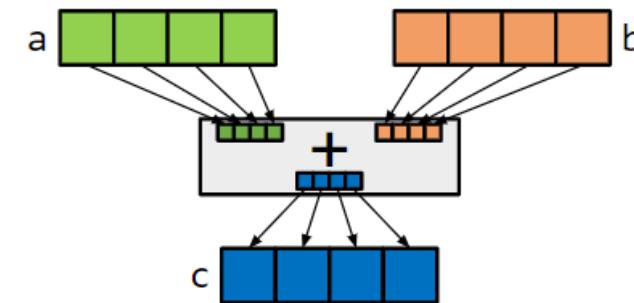


Arquitectura vectorial

Casos para procesamiento de datos

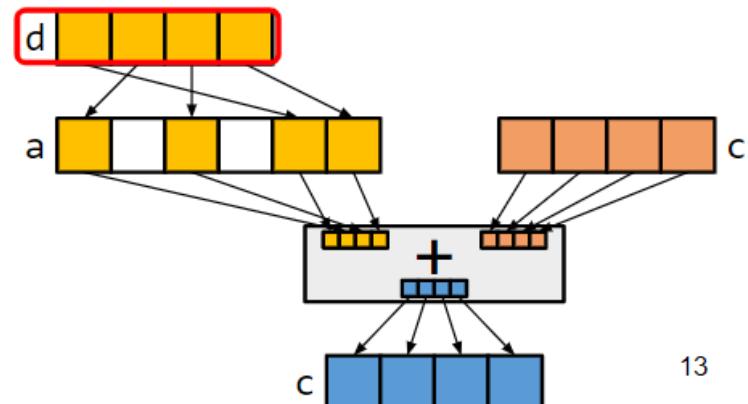
- Datos contiguos (en memoria).

```
38 void* add(void* arg)
39 {
40     for (int i = 0; i <= MAX_SIZE; i++) {
41         c[i] = a[i] + b[i];
42     }
43 }
```

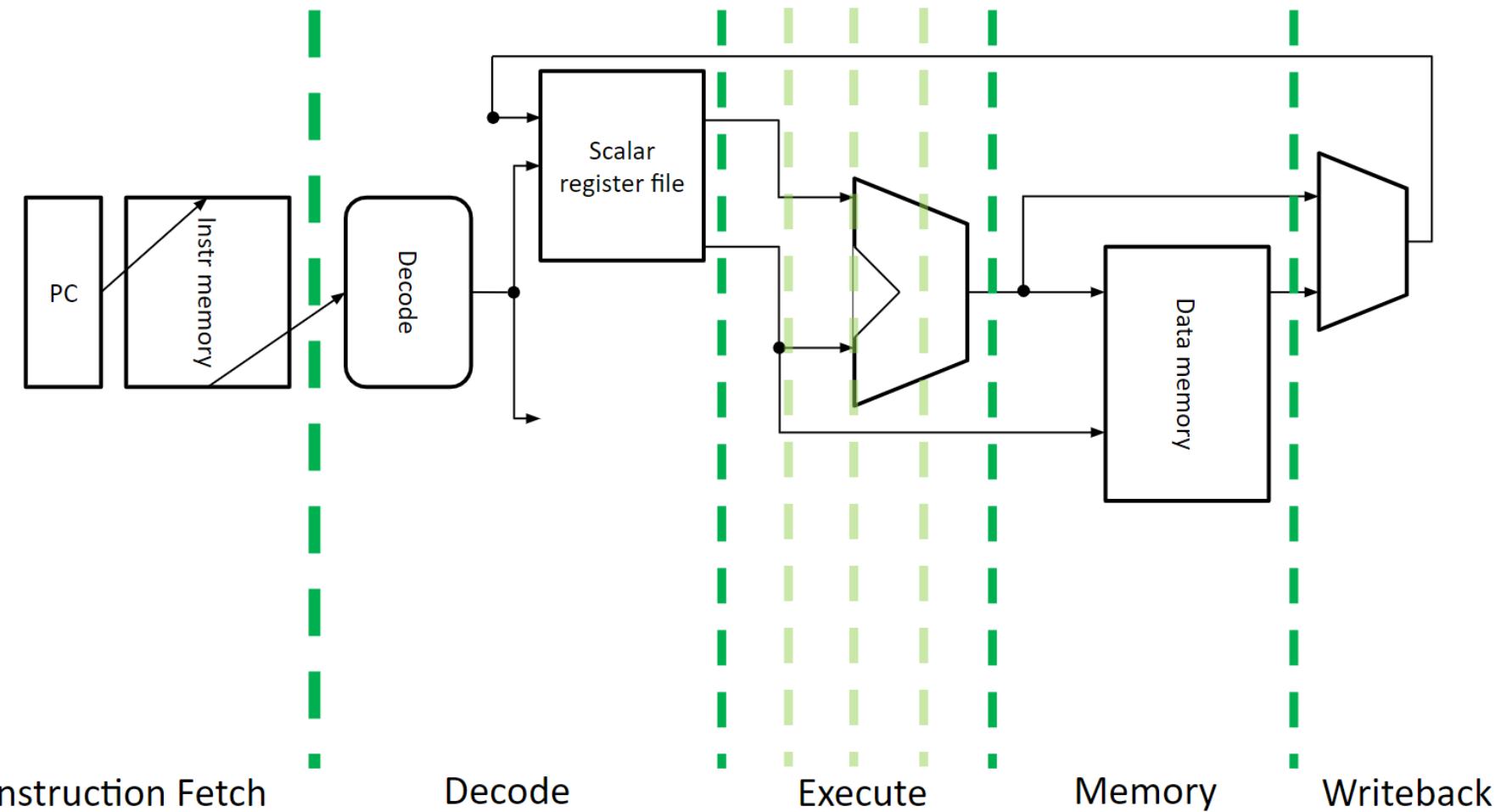


- Datos dispersos (en memoria).

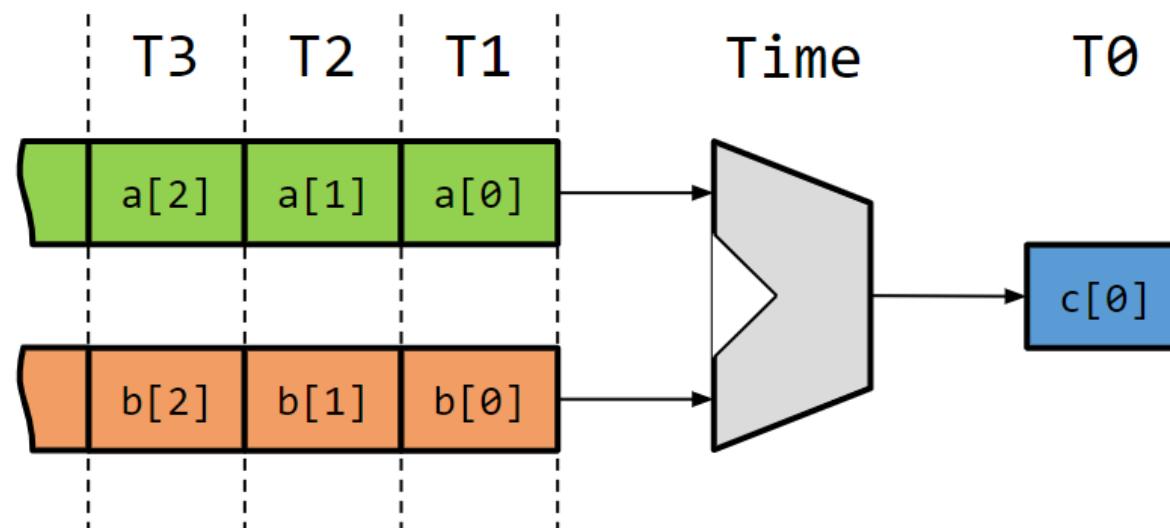
```
38 void* add(void* arg)
39 {
40     for (int i = 0; i <= MAX_SIZE; i++) {
41         c[i] = a[d[i]] + b[i];
42     }
43 }
```



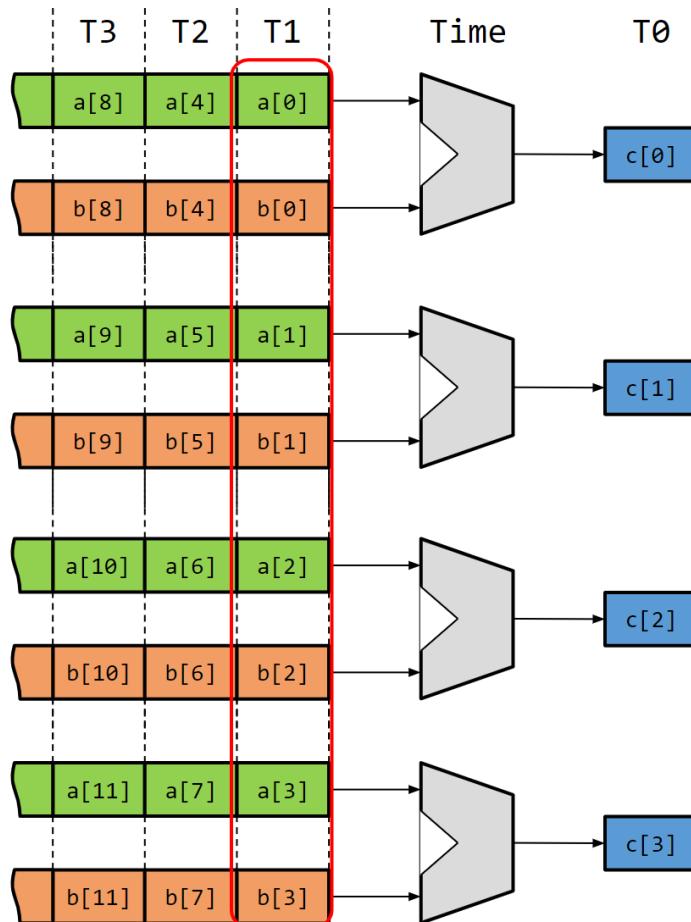
Arquitectura vectorial



Arquitectura vectorial



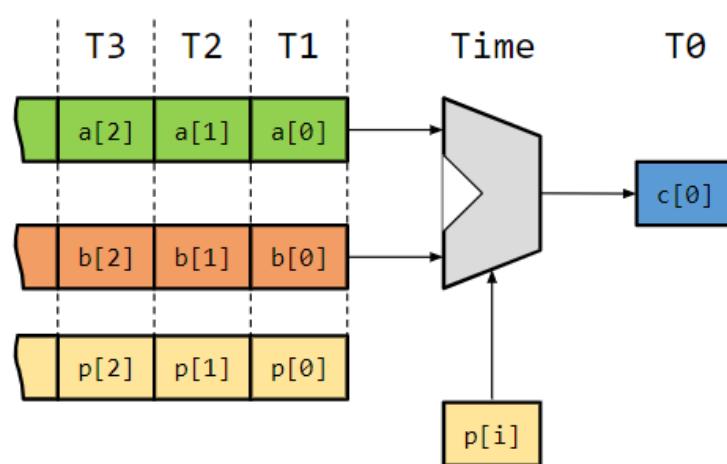
Arquitectura vectorial



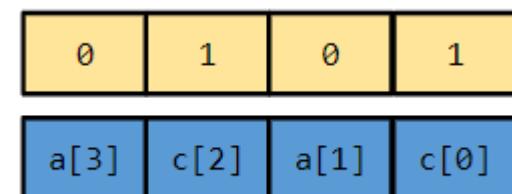
Arquitectura vectorial

Uso de predicado / Vector máscara

- Arreglo de un bit. Permite control sobre las operaciones de vector.

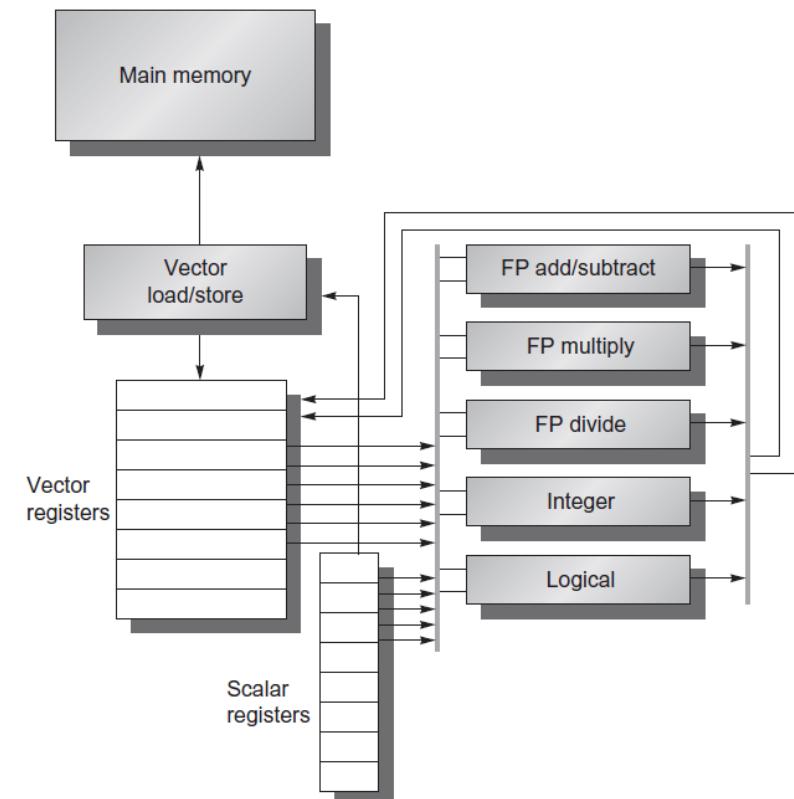


```
1  if (p[i] == 1) {  
2      c[i] = a[i] + b[i];  
3  }  
4  else {  
5      c[i] = a[i];  
6  }
```

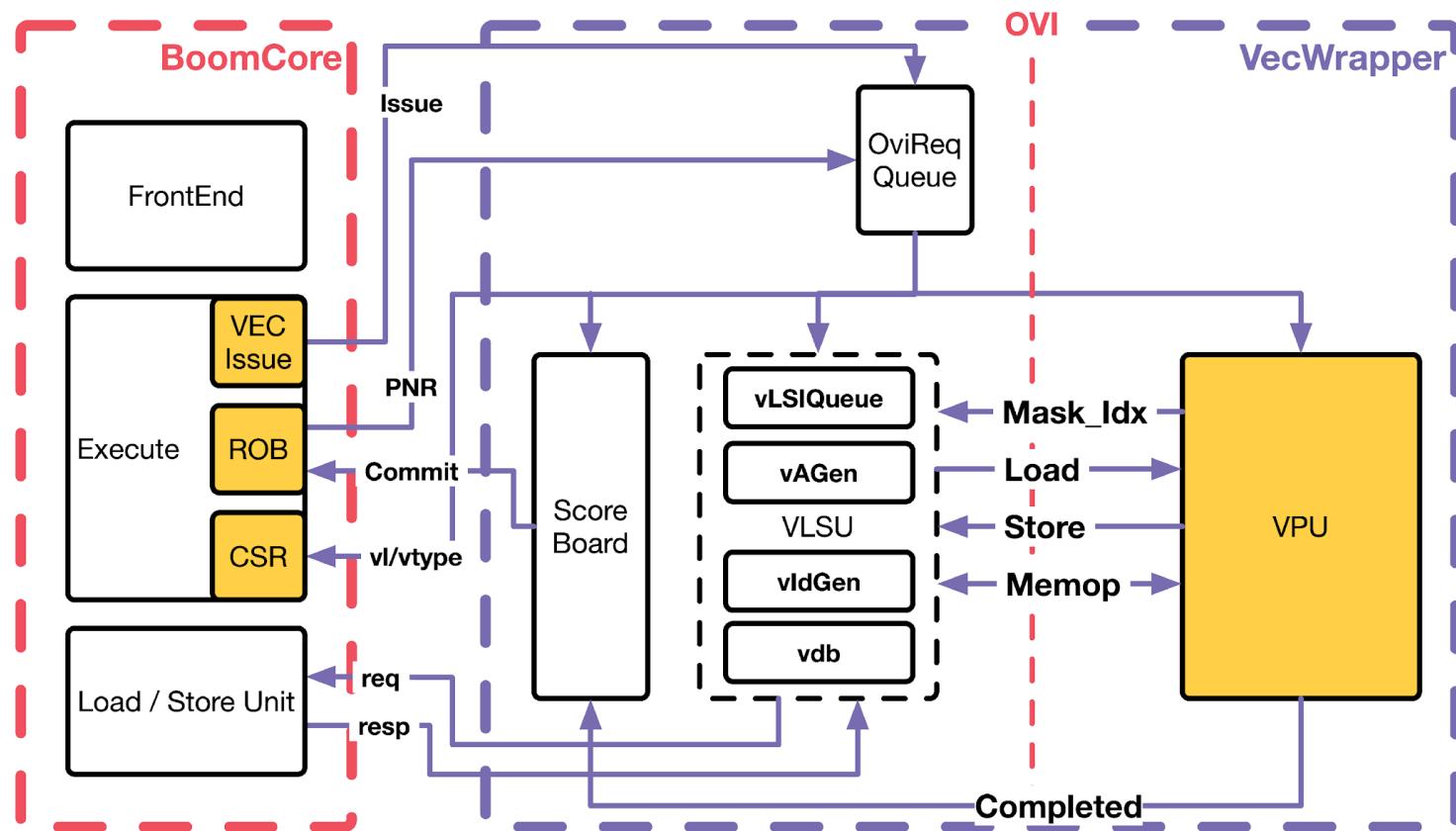


Arquitectura vectorial – RV64V

- Registros vectoriales. Cada registro almacena un vector. RV64V tiene 32 de 64 bits.
- FU vectoriales. Puede estar con pipeline.
- Unidad load/store.
- Grupo de registros escalares.



Arquitectura vectorial – Ocelot



Arquitectura vectorial – Ocelot

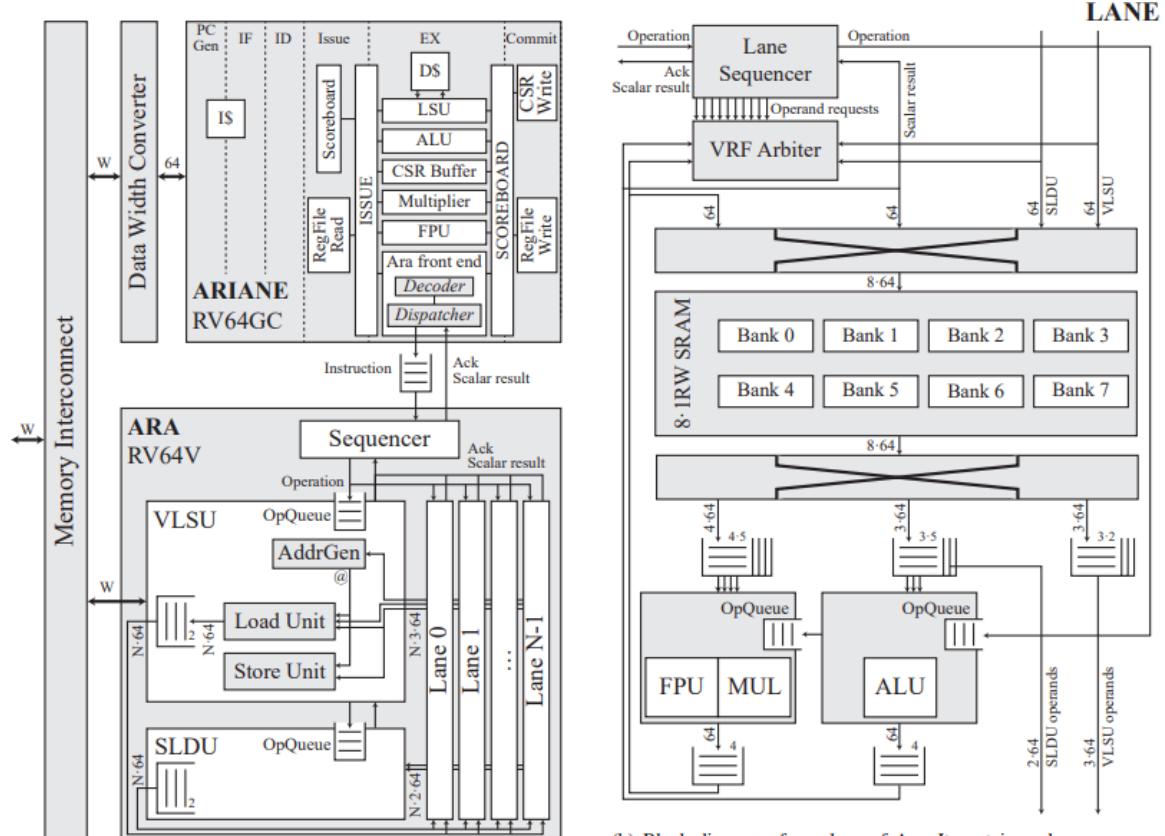
In-order Vector Issue Queue: Functioning similarly to other issue queues, it monitors write-back signals from INT/FP/MEM units, activating vector instructions with dependencies on INT/FP registers.

Request Queue: This queue manages vector instructions that have resolved their INT/FP register dependencies. It employs a Point of No Return (PNR) mechanism to halt the process if the oldest vector instruction is speculative.

Detached Vector Load/Store Unit: In alignment with the OVI specifications, the core must generate vector load/store requests. Leveraging this requirement, we have created a fully detached unit to enhance performance, enabling unit-stride and strided loads to be executed out-of-order.

Augmented Load/Store Unit: Modifications to the existing Load/Store Unit (LSU) have been made to handle dependencies between scalar and vector Load/Store requests effectively.

Arquitectura vectorial – RV64



(a) Block diagram of an Ara instance with N parallel lanes. Ara receives its commands from Ariane, a RV64GC scalar core. The vector unit has a main sequencer; N parallel lanes; a Slide Unit (SLDU); and a Vector Load/Store Unit (VLSU). The memory interface is W bit wide.

(b) Block diagram of one lane of Ara. It contains a lane sequencer (handling up to 8 vector instructions); a 16 KiB vector register file; ten operand queues; an integer Arithmetic Logic Unit (ALU); an integer multiplier (MUL); and a Floating Point Unit (FPU).

Ara: A 1 GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating Point Support in 22 nm FD-SOI

Matheus Cavalcante,* Fabian Schuiki,* Florian Zaruba,* Michael Schaffner,* Luca Benini,^{*†} Fellow, IEEE

RV64V <https://arxiv.org/pdf/1906.00478.pdf>

Arquitectura vectorial

Código fuente para DAXPY. X y Y tienen 32 elementos. x5 y x6 son las direcciones iniciales de X y Y.

Here is the RISC-V code:

```
fld    f0,a          # Load scalar a
addi   x28,x5,#256   # Last address to load
Loop: fld    f1,0(x5)  # Load X[i]
      fmul.d f1,f1,f0  # a × X[i]
      fld    f2,0(x6)  # Load Y[i]
      fadd.d f2,f2,f1  # a × X[i] + Y[i]
      fsd    f2,0(x6)  # Store into Y[i]
      addi   x5,x5,#8   # Increment index to X
      addi   x6,x6,#8   # Increment index to Y
      bne    x28,x5,Loop # Check if done
```

{S,D}APY operations

Given X and Y as vectors and A as a scalar => $Y \leftarrow AX + Y$

S => Single Precision

D => Double Precision

Here is the RV64V code for DAXPY:

```
vsetdcfg 4*FP64      # Enable 4 DP FP vregs
fld    f0,a          # Load scalar a
vld    v0,x5          # Load vector X
vmul  v1,v0,f0        # Vector-scalar mult
vld    v2,x6          # Load vector Y
vadd  v3,v1,v2        # Vector-vector add
vst    v3,x6          # Store the sum
vdisable                    # Disable vector regs
```

Arquitectura vectorial

<https://godbolt.org/z/xPYh7Ws84>

```
1 double X[32] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.
2 double Y[32] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.
3 double a      = 1.0;
4 void add(){
5
6     for (int i = 0; i < 32; ++i){
7         Y[i] += a*X[i] ;
8     }
9 }
```

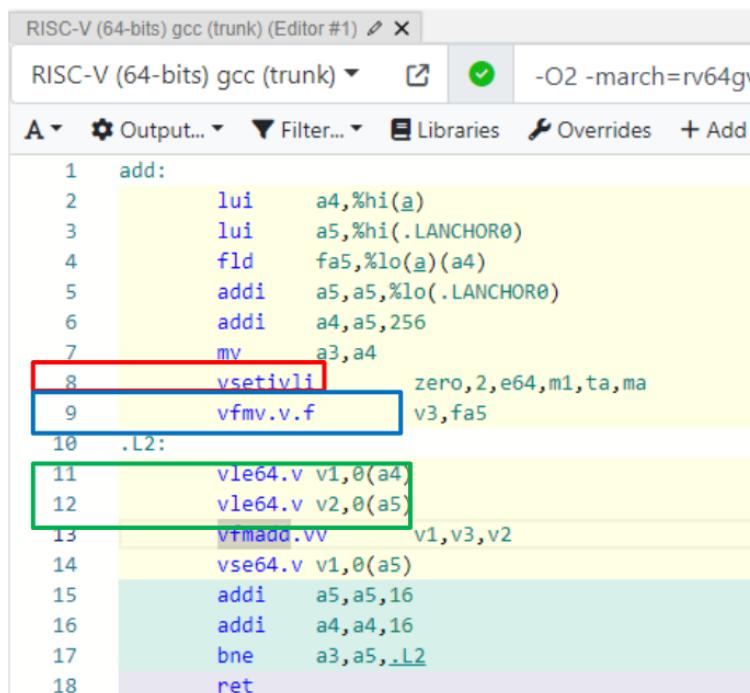
RISC-V (64-bits) gcc (trunk) (Editor #1)   -O2 -march=rv64gv

A  Output...  Filter... Libraries Overrides + Add r

```
1 add:
2     lui    a4,%hi(a)
3     lui    a5,%hi(.LANCHOR0)
4     fld    fa5,%lo(a)(a4)
5     addi   a5,a5,%lo(.LANCHOR0)
6     addi   a4,a5,256
7     mv    a3,a4
8     vsetivli zero,2,e64,m1,ta,ma
9     vfmv.v.f v3,fa5
10 .L2:
11    vle64.v v1,0(a4)
12    vle64.v v2,0(a5)
13    vfmadd.vv v1,v3,v2
14    vse64.v v1,0(a5)
15    addi   a5,a5,16
16    addi   a4,a4,16
17    bne   a3,a5,_L2
18    ret
```

Arquitectura vectorial

<https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc#sec-vector-config>



The screenshot shows an assembly editor interface for RISC-V (64-bits) with gcc (trunk). The code window displays the following assembly code:

```
1 add:
2     lui    a4,%hi(a)
3     lui    a5,%hi(.LANCHOR0)
4     fld    fa5,%lo(a4)(a4)
5     addi   a5,a5,%lo(.LANCHOR0)
6     addi   a4,a5,256
7     mv    a3,a4
8     vsetivli zero,2,e64,m1,ta,ma
9     vfmv.v.f v3,fa5
10 .L2:
11    vle64.v v1,0(a4)
12    vle64.v v2,0(a5)
13    vtmadd.vv v1,v3,v2
14    vse64.v v1,0(a5)
15    addi   a5,a5,16
16    addi   a4,a4,16
17    bne   a3,a5,.L2
18    ret
```

A red box highlights the `vsetivli` instruction at line 8. A callout box provides documentation for this instruction:

A set of instructions is provided to allow rapid configuration of the values in `vl` and `vtype` to match application needs. The `vset{i}vl{i}` instructions set the `vtype` and `vl` CSRs based on their arguments, and write the new value of `vl` into `rd`.

Below the code window, two callout boxes provide detailed descriptions of specific instructions:

13.16. Vector Floating-Point Move Instruction

The vector floating-point move instruction *splats* a floating-point scalar operand to a vector register group. The instruction copies a scalar `f` register value to all active elements of a vector register group. This instruction is encoded as an unmasked instruction (`vm=1`). The instruction must have the `vs2` field set to `v0`, with all other values for `vs2` reserved.

`vfmv.v.f vd, rs1 # vd[i] = f[rs1]`

7.4. Vector Unit-Stride Instructions

Vector unit-stride loads and stores

vd destination, rs1 base address, vm is mask encoding (v0.t or <missing>)

`vle8.v vd, (rs1), vm # 8-bit unit-stride load`
`vle16.v vd, (rs1), vm # 16-bit unit-stride load`
`vle32.v vd, (rs1), vm # 32-bit unit-stride load`
`vle64.v vd, (rs1), vm # 64-bit unit-stride load`

Investigar

- ¿Qué afecta el tiempo de ejecución vectorial?
- ¿Qué diferencia las extensiones SIMD a un procesamiento vectorial genérico?
- ¿Qué son x86 AVX/AMX, y ARM SVE/SME?
- ¿Qué son intrinsics?

Referencias

- Stallings, W. (2003). Computer organization and architecture: designing for performance. Pearson Education India.
- Hennessy, J., & Patterson, D. (2012). Computer Architecture: A Quantitative Approach (5th ed.). Elsevier Science.

CE4302 – Arquitectura de Computadores II

Introducción al parallelismo a nivel de datos

PROFESOR: ING. LUIS BARBOZA ARTAVIA