Parallelism and Vector Instructions

CMPT 295 Week 9

Parallelism and Vector Instructions

WARNING: Lab 9, Ass 5 work with fixed-length vector intrinsics. Not RISC-V

- Most concepts carry over, if not programming details
- RISC-V supports variable length vectors.
  Lab 9 and ASS 5 do not

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```

Memory & data
Arrays & structs
Integers & floats
RISC V assembly
Procedures & stacks
Executables
Memory & caches
Parallelism
Processor Pipeline
Performance

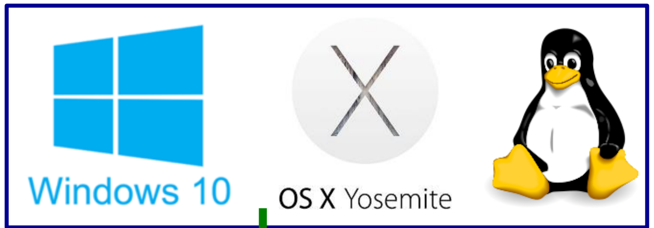
Assembly language:

```
get_mpg(car*):
        lw      a5,0(a0)
        lw      a4,4(a0)
        divw    a5,a5,a4
        fcvt.s.w        fa0,a5
        ret
```

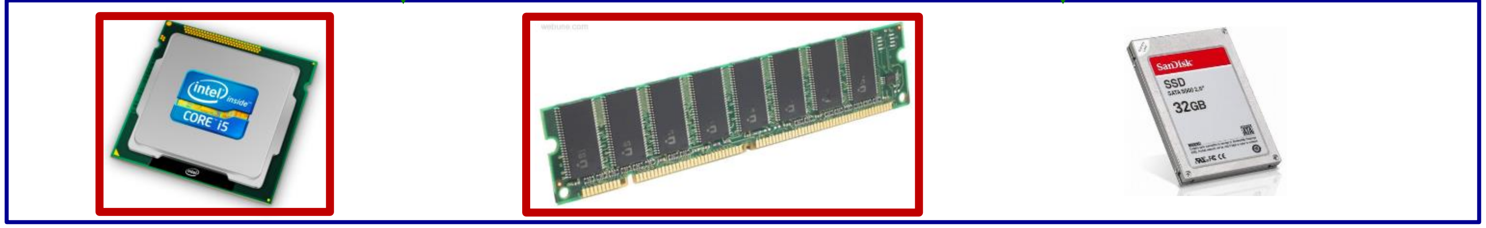
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:



Computer system:



What is a computer program?

```
for (int i = 0; i < N; i++){
  output[i] = x[i] * y[i];
}
```

What is a computer program?

Processor executes instruction referenced by the program counter (PC)

(executing the instruction will modify machine state: contents of registers, memory, CPU state, etc.)

Move to next instruction …

Then execute it…

And so on…

PC →

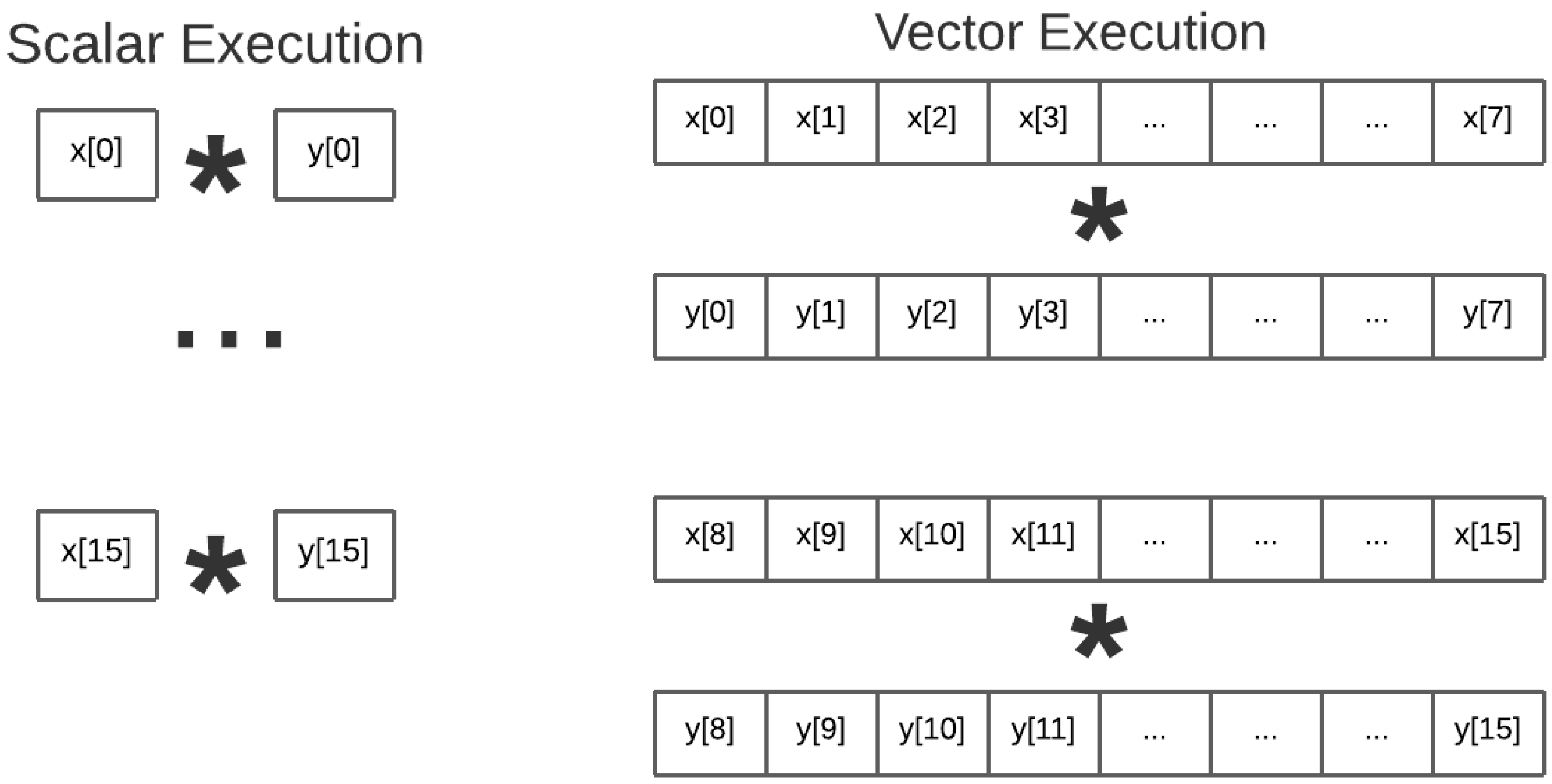
```
# a0: &x[0], a1: &y[0], a2:
&result[0], a5: N
# t1 = 0: loop index i
loop:
# load x[i] and y[i]
lw a4,0(a0)
lw a3,0(a1)
# multiplication
mul a4,a4,a3
# store word
sw a4,0(a2)
# Bump pointers
addi a0,a0,4
addi a1,a1,4
addi a2,a2,4
addi t1, t1, 1
bne t1,a5,loop
```

Scalar Loop

```
for (i = 0; i < N; i++){
 output[i] = x[i] * y[i];
}
```

Vector Loop (data parallelism)

```
for (i = 0; i < N; i=i+VLEN){
 output[i:i+VLEN-1] =
   x[i:i+VLEN-1] * y[i:i+VLEN-1];
}
```


Scalar Execution
x[0]
*
y[0]
...
x[15]
*
y[15]
Vector Execution
x[0] x[1] x[2] x[3] ... ... x[7]
*
y[0] y[1] y[2] y[3] ... ... y[7]
x[8] x[9] x[10] x[11] ... ... x[15]
*
y[8] y[9] y[10] y[11] ... ... y[15]

```
for (i = 0; i < N; i++){
 output[i] = x[i] * y[i];
}
```

```
# a0: &x[0], a1: &y[0], a2:
&result[0], a5: N
# t1 = 0: loop index i
loop:
# load x[i] and y[i]
lw a4,0(a0)
lw a3,0(a1)
# multiplication
mul a4,a4,a3
# store word
sw a4,0(a2)
# Bump pointers
addi a0,a0,4
addi a1,a1,4
addi a2,a2,4
addi t1, t1, 1
bne t1,a5,loop
```

How many total ins?

N * 9

How many useful inst?

4* N (LD,LD,MUL,ST)

How many useless (maintenance) inst?

5*N

```
for (i = 0; i < N; i=i+VLEN){
 output[i:i+VLEN-1] =
   x[i:i+VLEN-1] * y[i:i+VLEN-1];
}
```

How many total ins?

N * 9 / VLEN

How many useful ins ?

4*N/VLEN

How many useless inst?

5*N / VLEN

Why Parallelism?
Why Efficiency?

A parallel computer is a collection of processing elements that cooperate to solve problems quickly

We care about performance

We care about efficiency

We're going to use multiple processing element to get it

Speedup

One major motivation of using parallel processing: Speedup

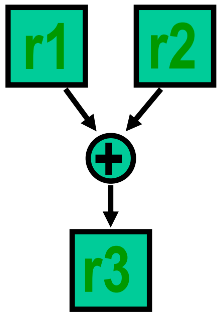
For a given problem:

$$\text{speedup} = \frac{\text{execution time (using 1 elements)}}{\text{execution time (using P elements)}}$$

Parallel Model: Vector Processing

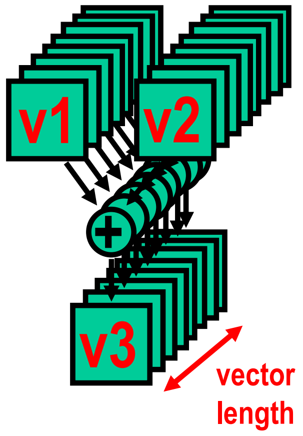
- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

SCALAR
(1 operation)



`add r3, r1, r2`

VECTOR
(N operations)

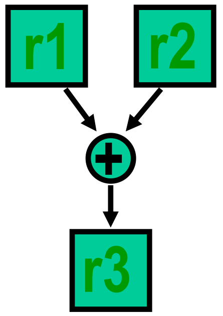


`add.vv v3, v1, v2`

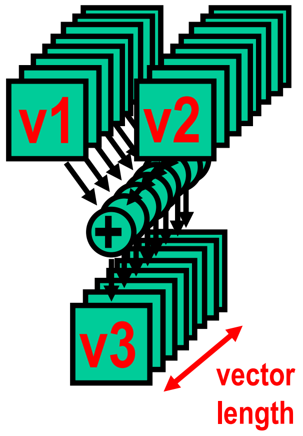
Parallel Model:Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

SCALAR (1 operation)



VECTOR (N operations)



```
add r3, r1, r2
```

```
add.vv v3, v1, v2
```

```
out[0] = x[0]+y[0]
out[1] = x[1]+y[1]
….
```

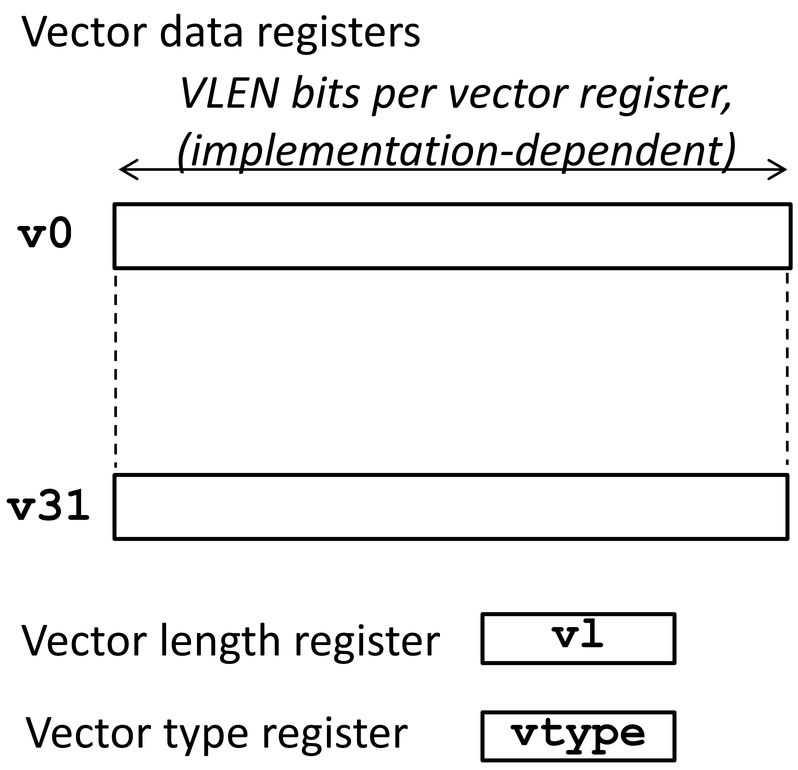
```
out[0:VLEN-1] = x[0:VLEN-1] + Y[0:VLEN-1]
```

```
out[VLEN:2*VLEN-1] = x[VLEN:2*VLEN-1]
           + y[VLEN:2*VLEN-1]
```

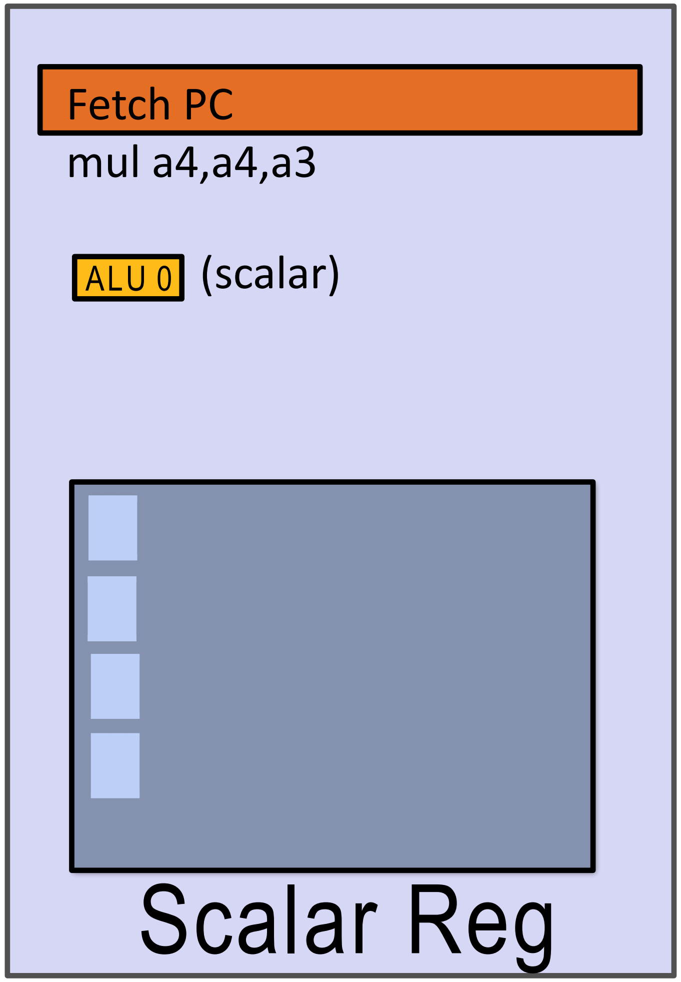
Vector Registers

- **32 vector data registers, `v0-v31`, each VLEN bits long**
- **Vector length register `vl`**
- **Vector type register `vtype`**

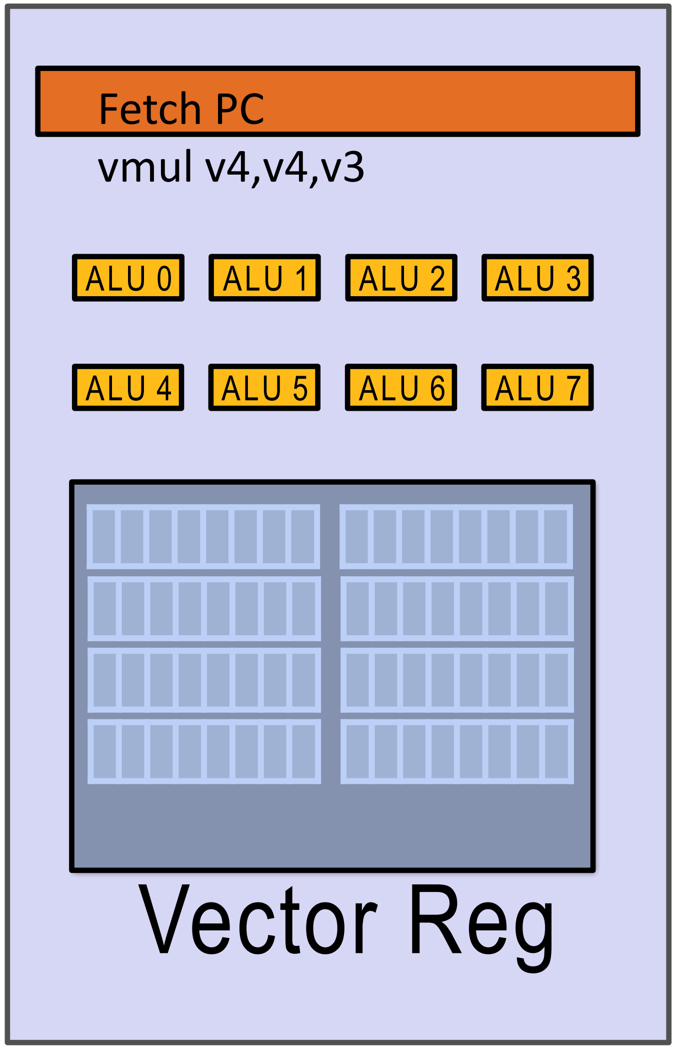
- **Vector register file**
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- **Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes")**



Baseline CPU



Vector CPU: Add arithmetic units to increase compute capability



Fetch Ins

Single instruction, multiple data

- **Parallelism**: Multiple data elements
- **Efficiency**: Fetch single instruction

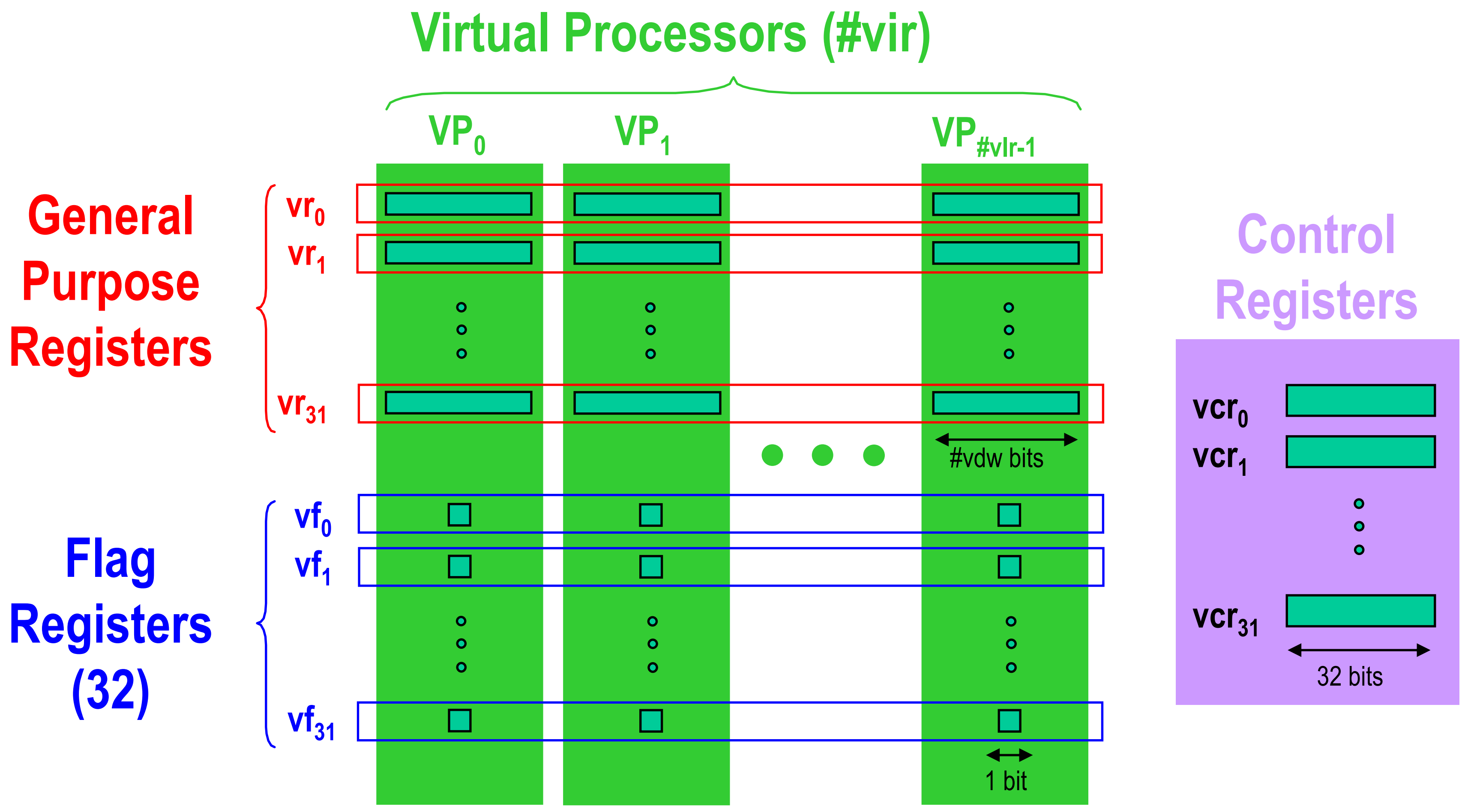
Same instruction broadcast on all ALUs

Each instruction updates/reads multiple elements from vector register

Virtual Processor Vector Model

- Vector operations are SIMD
  (single instruction multiple data) operations
- Each element is computed by a virtual processor (VP)
- Number of VPs given by vector length
  - vector control register

Vector Architectural State



Scalar Code

```
for (i = 0; i < N; i++){
 output[i] = x[i] + y[i];
}
```

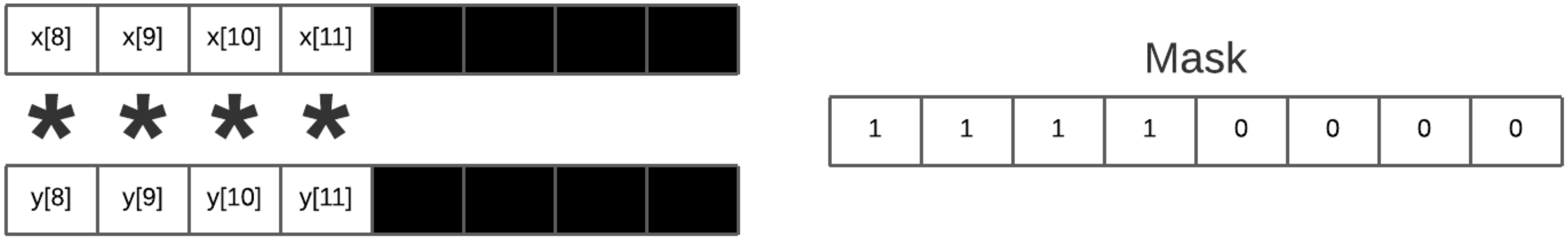
```
loop:
# load x[i] and y[i]
lw a5,0(a2)
lw a6,0(a3)
# addition
add a5,a5,a6
# store word
sw a5,0(a1)
# Bump pointers
addi a1,a0,4
addi a2,a1,4
addi a3,a2,4
addi a3,a2,4
sub  a0,a0,1
bnez a0, loop
```

Vector Code

```
for (i = 0; i < N;i=i+VLEN){
 output[i:i+VLEN] =
   x[i:i+VLEN] + y[i:i+VLEN];
}
```

```
loop: # t0=VLEN
# load x[I,i+VLEN], y[] vle32.v v8, (a2)
vle32.v v16, (a3)
# addition
vadd.vv v24,v8,v16
# store res[i:i+VLEN]
vse32.v v24,(a1)
# Bump pointers
slli t1,t0,2
add  a2, a2,t1
add  a3,a3,t1
add  a1,a1,t1
# Bump loop by vlen
sub  a0,a0,t0
bnez a0, loop
```

Masking and Conditional Ops

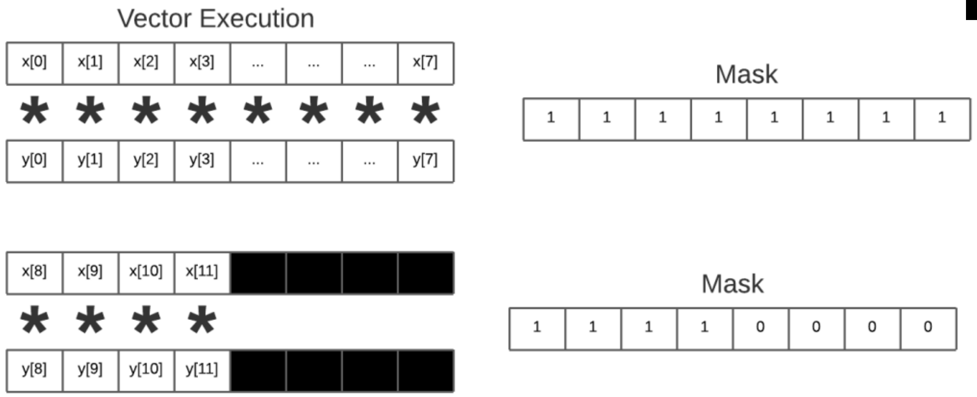


- Disable unwanted vector lanes
- Conditional branches where different operations for different vector elements
- Handling tail/left-over elements when software array length not multiple of vector width.

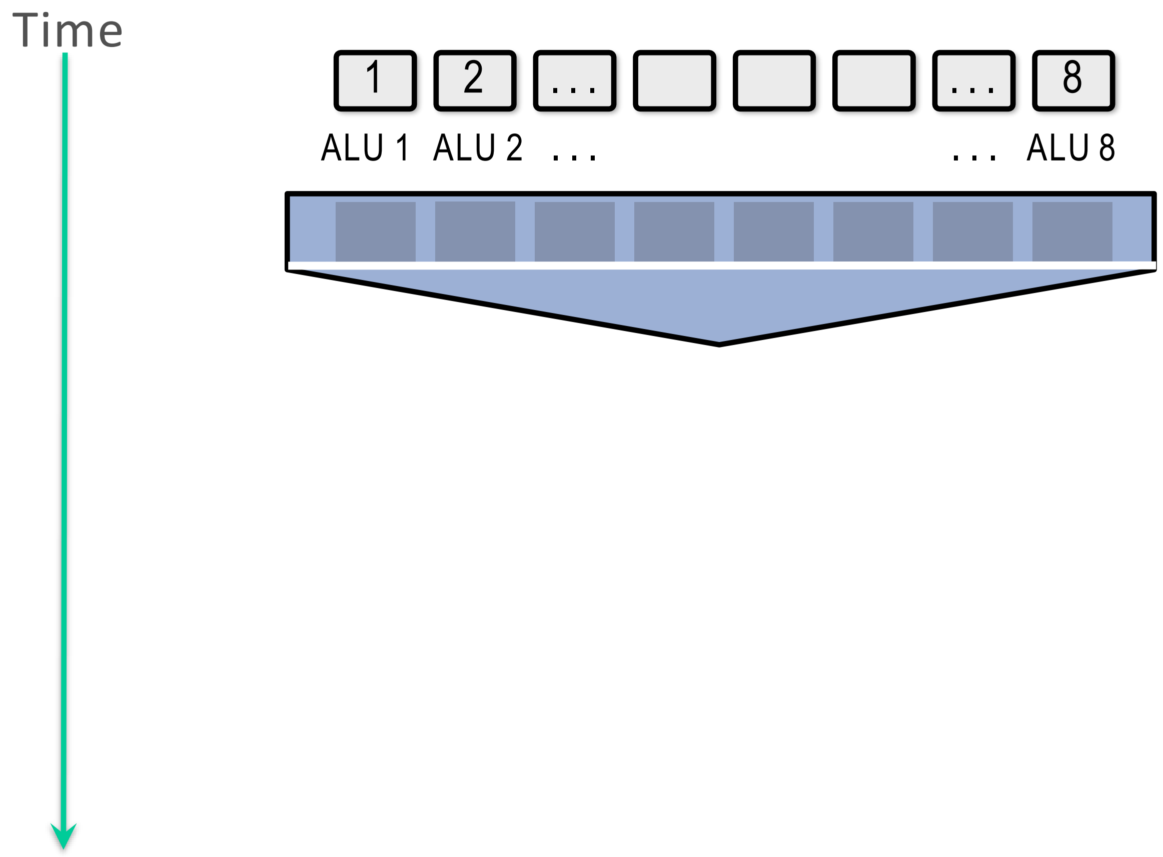
Tail Processing

```
Remaining = N
for (i = 0; i < N;){
int VLEN;
if (N-i > MAX_VLEN)
  VLEN = MAX_VLEN
else
  VLEN = N-i

setvl(VLEN)

res[i:i+VLEN] =
   x[i:i+VLEN] + y[i:i+VLEN];
}
```

```
loop:
vsetvli t0, a0, e32 # Set VLEN
# load x[I,i+VLEN], y[] vle32.v v8, (a2)
vle32.v v16, (a3)
# addition
vadd.vv v24,v8,v16
# store res[i:i+VLEN]
vse32.v v24,(a1)
# Bump pointers
slli t1,t0,2
add  a2, a2,t1
add  a3,a3,t1
add  a1,a1,t1
# Bump loop by vlen
sub  a0,a0,t0
bnez a0, loop
```



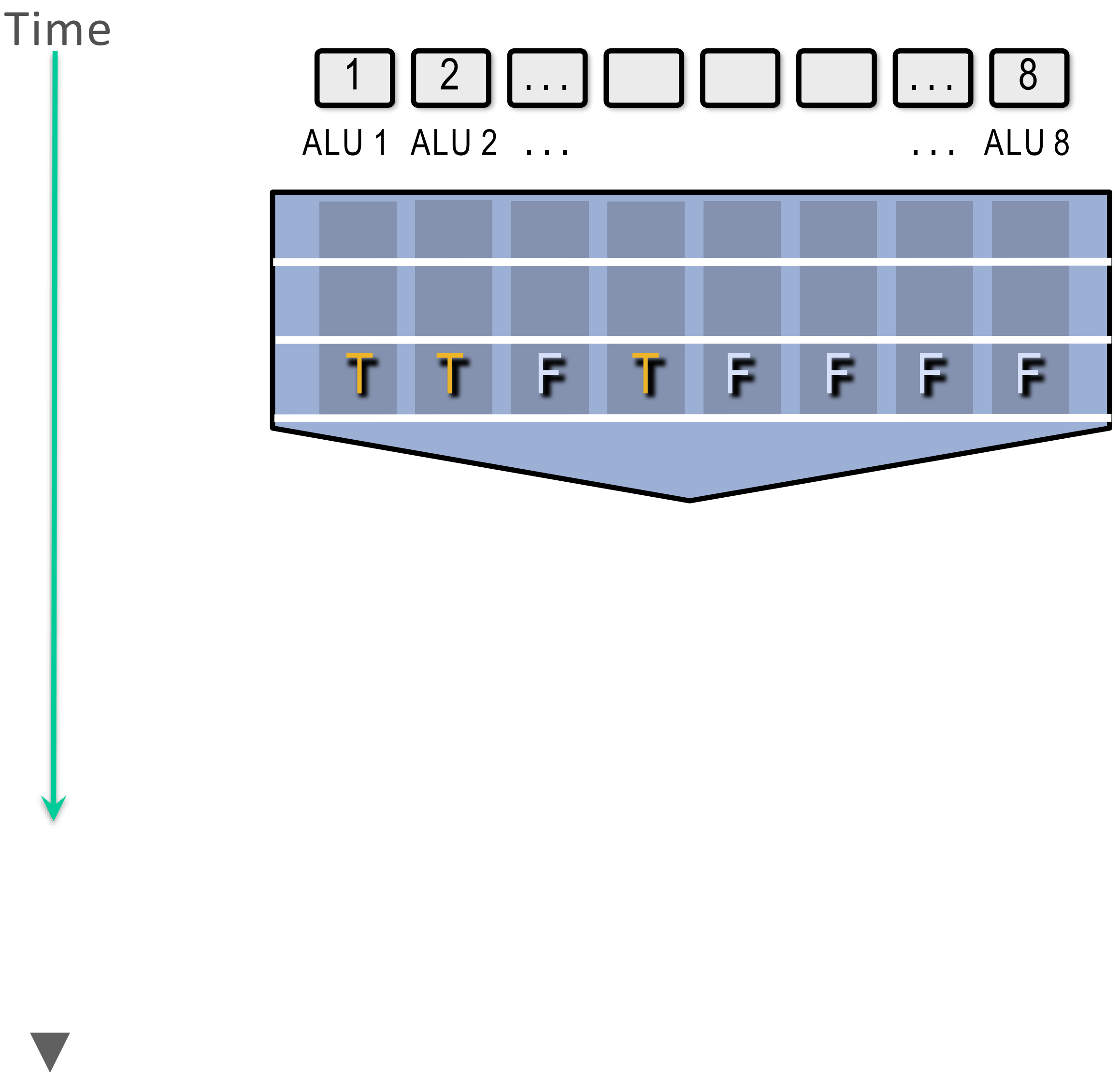
What about conditional branches?



Assume logic below is to be executed for each element in input array ‘A’ producing output into array ‘result’

```
 <unconditional code>

  float x = A[i];

if (x > 0) {
     float tmp = exp(x,5.f);
     tmp *= kMyConst1;
     x = tmp + kMyConst2;
  } else {
           float tmp = kMyConst1;
           x = 2.f * tmp;
  }

   <resume unconditional code>


 result[i] = x;
```

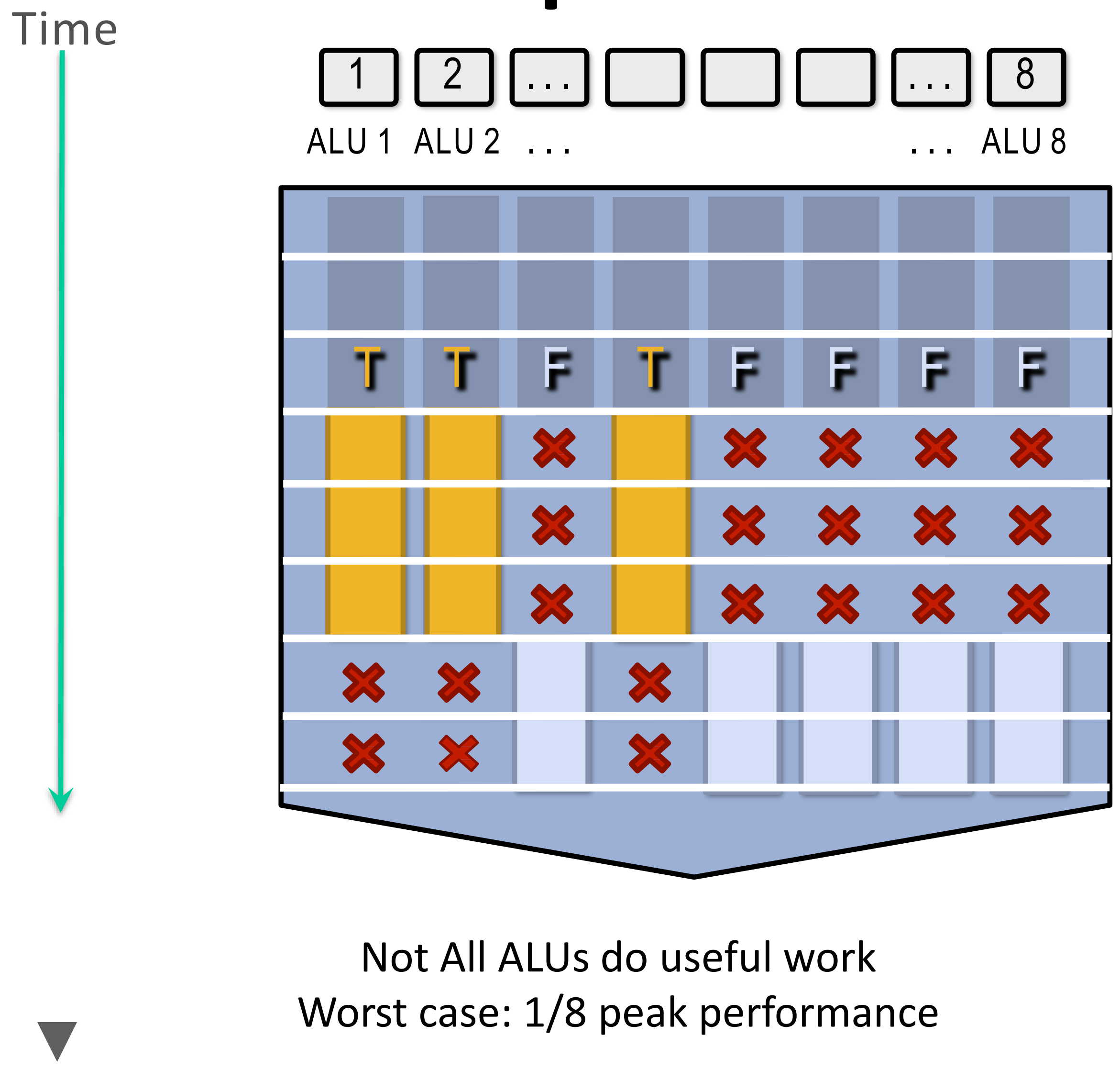
What about conditional branches?



Assume logic below is to be executed for each element in input array 'A' producing output into array 'result'

```
 <unconditional code>

 float x = A[i];

if (x > 0) {

    float tmp = exp(x,5.f);

    tmp *= kMyConst1;

    x = tmp + kMyConst2;
 } else {
       float tmp = kMyConst1;

       x = 2.f * tmp;
 }

   <resume unconditional code>


 result[i] = x;
```

Mask discard output of ALUs

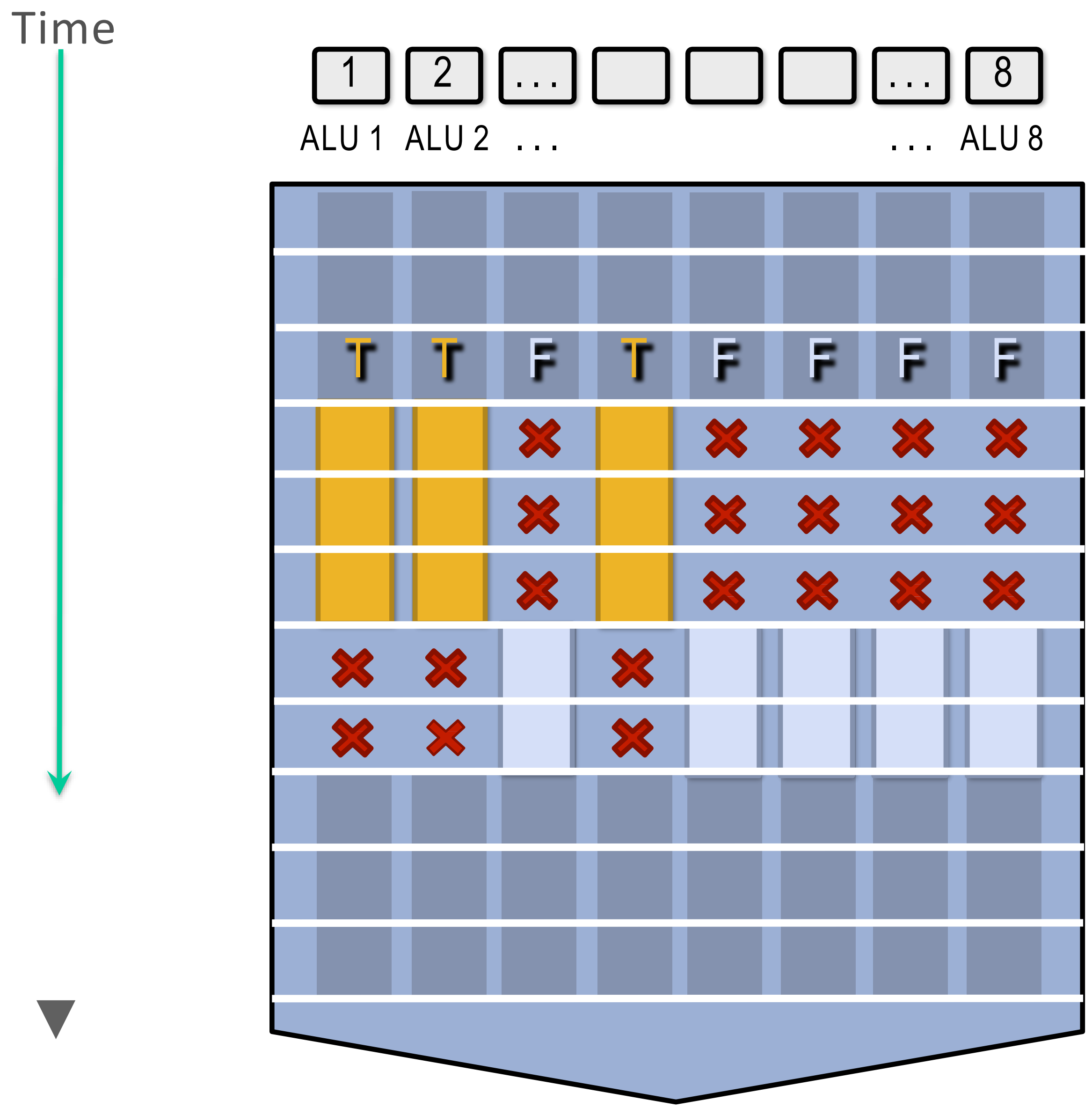


Assume logic below is to be executed for each element in input array 'A' producing output into array 'result'

```
<unconditional code>

float x = A[i];

if (x > 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Not All ALUs do useful work
Worst case: 1/8 peak performance

After branch continue normal execution



Assume logic below is to be executed for each element in input array 'A' producing output into array 'result'

```
 <unconditional code>

 float x = A[i];

if (x > 0) {
      float tmp = exp(x,5.f);
      tmp *= kMyConst1;
      x = tmp + kMyConst2;
 } else {
      float tmp = kMyConst1;
      x = 2.f * tmp;
 }

<resume unconditional code>

 result[i] = x;
```

Terminology

- Instruction stream coherence (“coherent execution”)
  - Same instruction sequence applies to all elements operated upon simultaneously
  - Coherent execution is necessary for efficient use of SIMD processing resources
  - Coherent execution IS NOT necessary for efficient parallelization across cores, since each core has the capability to fetch/decode a different instruction stream
- “Divergent” execution
  - A lack of instruction stream coherence

New RISC-V “V” Vector Extension

- Standard extension to the RISC-V ISA
  - An updated form of Cray-style vectors for modern microprocessors
  - Appearing in commercial implementations from Alibaba, Andes, Semidynamics, SiFive, …
  - Basis of European supercomputer initiative (EPI)
- Following slides present short tutorial on current standard
  - **`https://github.com/riscv/riscv-v-spec`**

RISC-V Scalar State

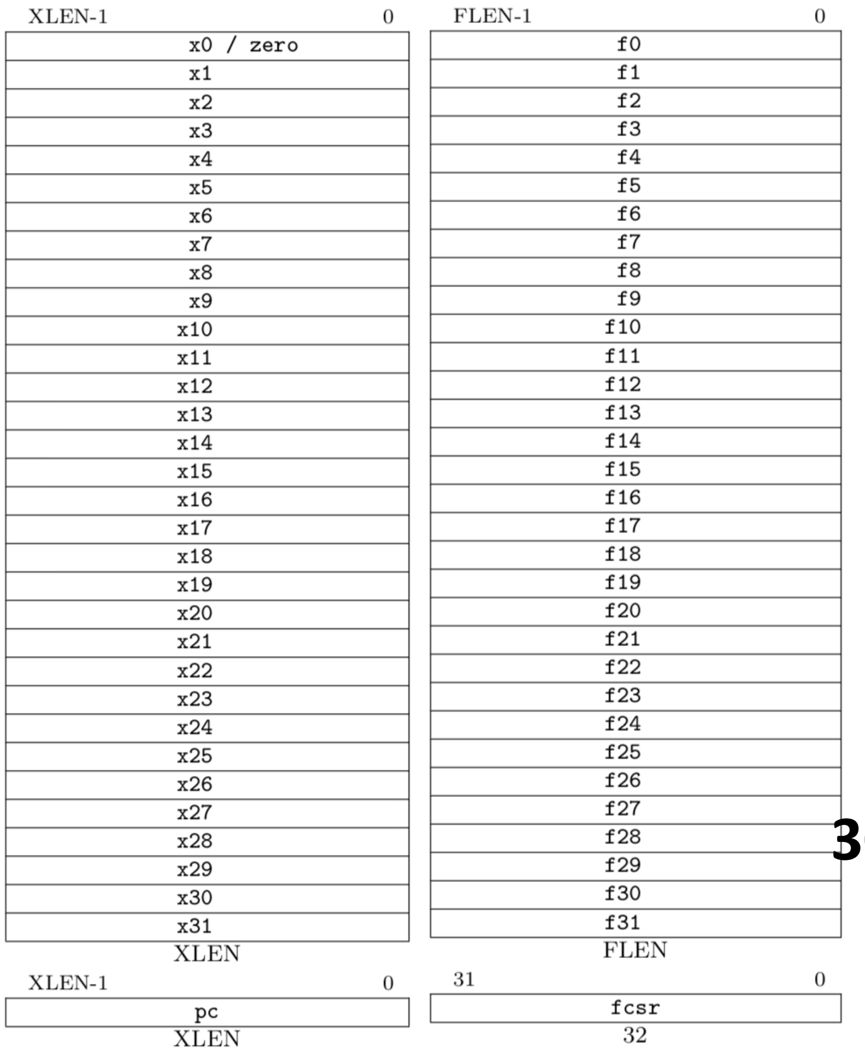
Program counter (**pc**)

32x32/64-bit integer registers (**x0-x31**)
• **x0** always contains a 0

Floating-point (FP), adds 32 registers (**f0-f31**)
• each can contain a single- or double-precision FP value (32-bit or 64-bit IEEE FP)

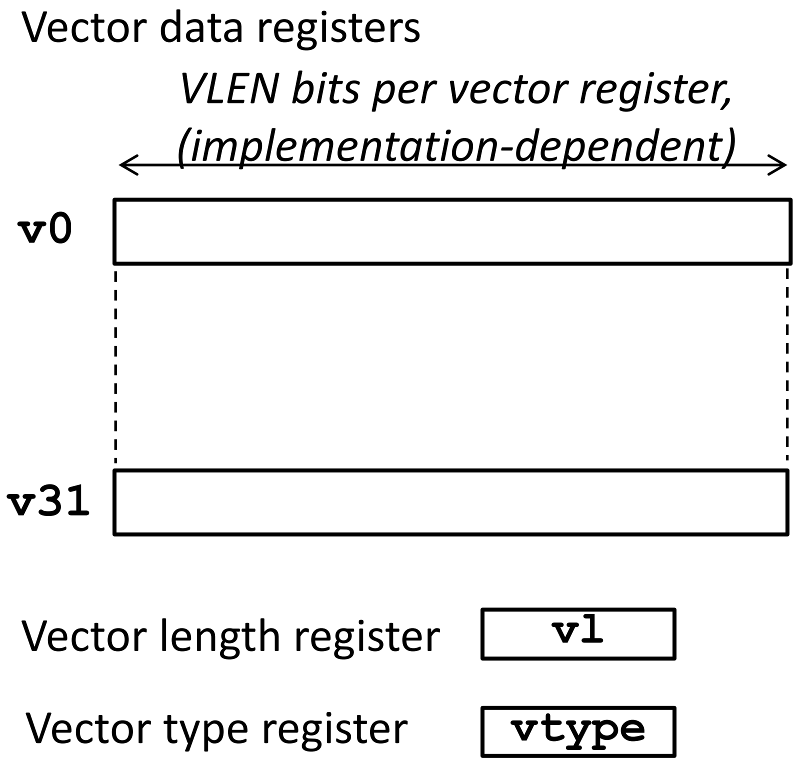
FP status register (**fcsr**), used for FP rounding mode & exception reporting

ISA string options:
- RV32I (XLEN=32, no FP)
- RV32IF (XLEN=32, FLEN=32)
- RV32ID (XLEN=32, FLEN=64)
- RV64I (XLEN=64, no FP)
- RV64IF (XLEN=64, FLEN=32)
- RV64ID (XLEN=64, FLEN=64)



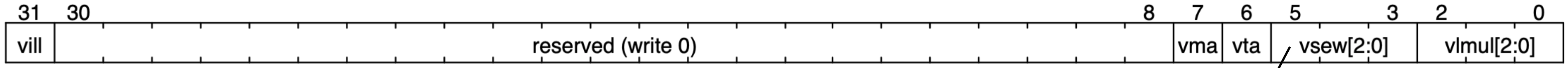
Vector Extension Additional State

- **32 vector data registers, `v0-v31`, each VLEN bits long**
- **Vector length register `vl`**
- **Vector type register `vtype`**
- **Other control registers:**
  - `vstart`
    - For trap handling
  - `vrm/vxsat`
    - Fixed-point rounding mode/saturation
    - Also appear in separate `vcsr`
  - `vlenb`
    - Gives vector length in bytes (read-only)



Vector Type Register (`vtype`)

Ideally, info would be in instruction encoding, but no space in 32-bit instructions. Planned 64-bit encoding extension would add these as instruction bits.



`vsew[2:0]` field encodes standard element width (SEW) in bits of elements in vector register (SEW = 8*$2^{\texttt{vsew}}$)

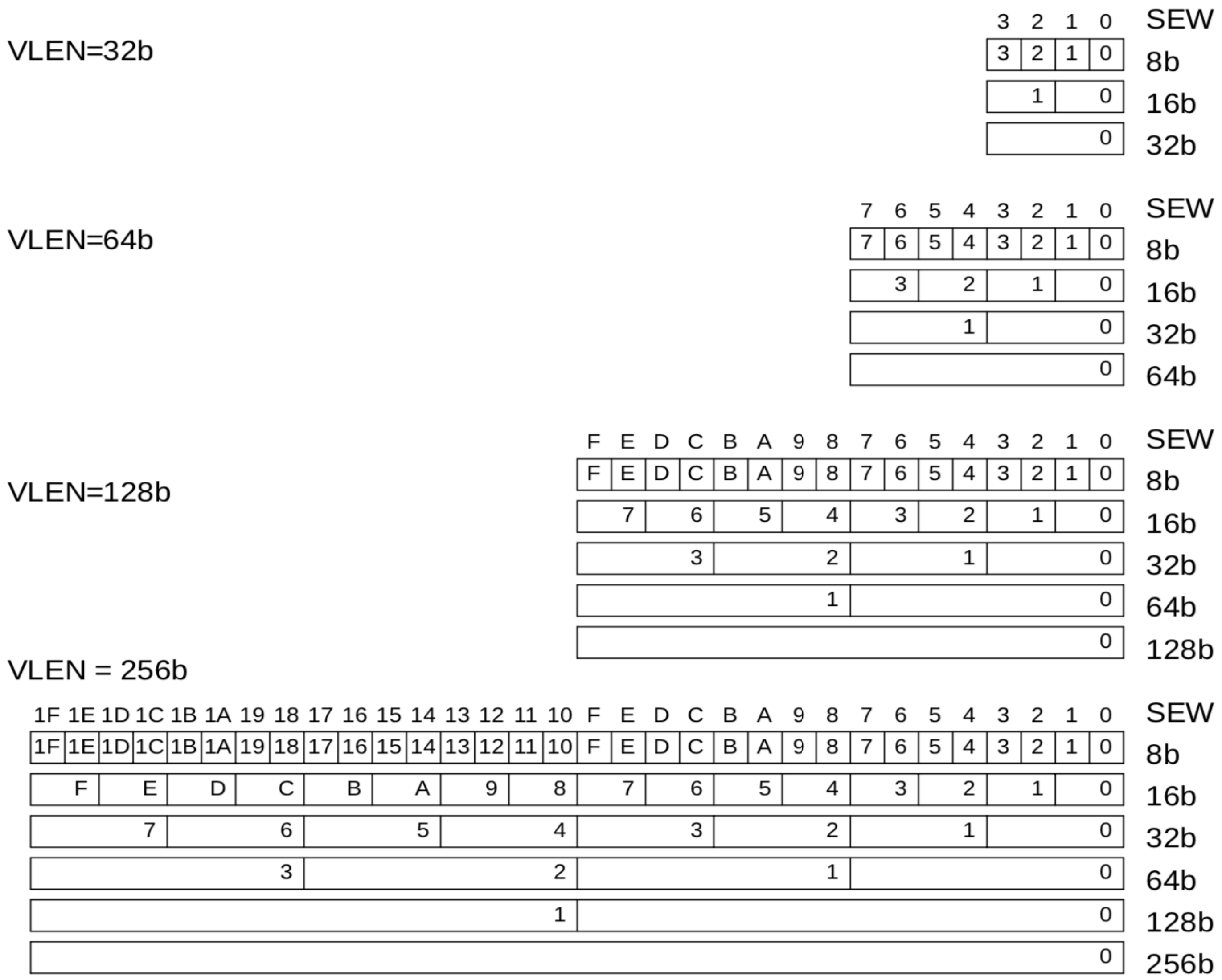
`vlmul[2:0]` encodes vector register length multiplier (LMUL = $2^{\texttt{vlmul}}$ = 1/8 - 8)

`vta` specifies *tail-agnostic*

`vma` specifies *mask-agnostic*

vsew[2:0]			SEW
0	0	0	8
0	0	1	16
0	1	0	32
0	1	1	64
1	0	0	128
1	0	1	256
1	1	0	512
1	1	1	1024

Example Vector Register Data Layouts (LMUL=1)


VLEN=32b
VLEN=64b
VLEN=128b
VLEN = 256b
SEW
8b
16b
32b
64b
128b
256b
1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 F E D C B A 9 8 7 6 5 4 3 2 1 0

Setting vector configuration, `vsetvli/vsetivli/vsetvl`

The **`vset{i}vl{i}`** configuration instructions set the **`vtype`** register, and also set the **`vl`** register, returning the **`vl`** value in a scalar register

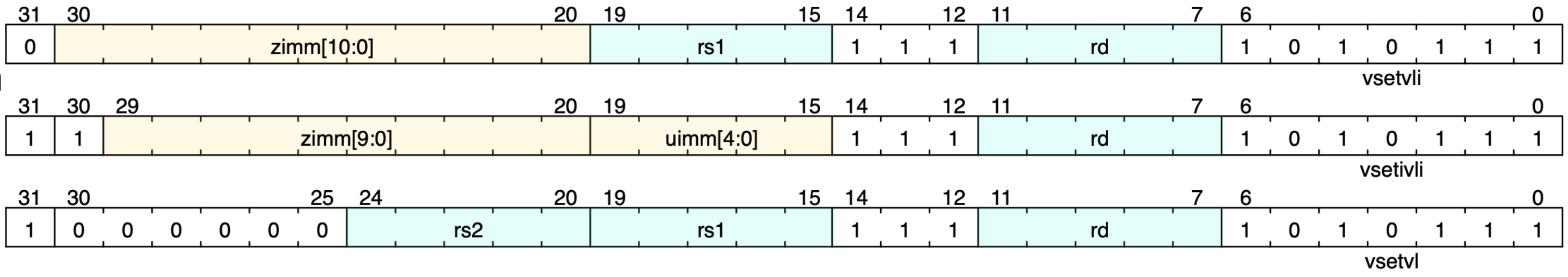
```
vsetvli rd, rs1, e8 # Set SEW=8, vl=min(VLEN/SEW,rs1), rd=vl
```

Resulting machine vector length setting

Requested application vector length

`vtype` parameters (SEW,LMUL,TA,MA) encoded as immediate in instruction

Instruction encoding

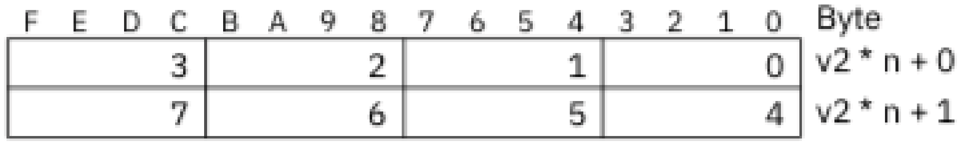

31 30 20 19 15 14 12 11 7 6 0
0 zimm[10:0] rs1 1 1 1 rd 1 0 1 0 1 1 1
vsetvli
31 30 29 20 19 15 14 12 11 7 6 0
1 1 zimm[9:0] uimm[4:0] 1 1 1 rd 1 0 1 0 1 1 1
vsetivli
31 30 25 24 20 19 15 14 12 11 7 6 0
1 0 0 0 0 0 0 rs2 rs1 1 1 1 rd 1 0 1 0 1 1 1
vsetvl

Usually use register-immediate form, **`vsetvli`**, to set **`vtype`** parameters.
Immediate-immediate form, **`vsetivli`**, used when vector length known statically
The register-register version **`vsetvl`** is usually used only for context save/restore

Vector Length Multiplier, LMUL

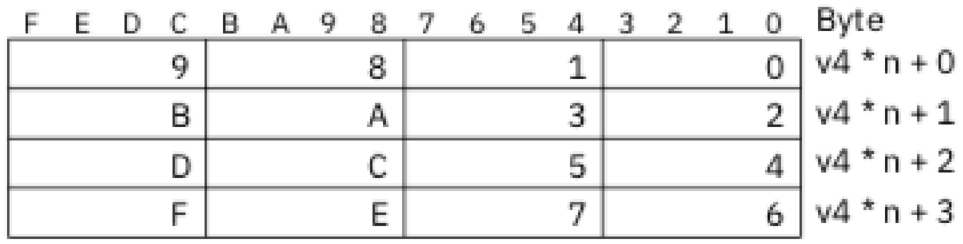
- Gives fewer but longer vector registers
  - **Called “vector register groups**” – operate as single vectors
  - **Must use even register names** only for LMUL=2 (v0,v2,..), and every fourth register for LMUL=4 (v0,v4, …), etc.
- Used for
  - 1) to increase efficiency by using longer vectors
  - 2) accommodate mixed-width operations (e.g., masks)

LMUL=2



F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
			3				2				1				0	v2 * n + 0
			7				6				5				4	v2 * n + 1

LMUL=4



F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
			9				8				1				0	v4 * n + 0
			B				A				3				2	v4 * n + 1
			D				C				5				4	v4 * n + 2
			F				E				7				6	v4 * n + 3

Simple stripmined vector `memcpy` example

```
  # void *memcpy(void* dest, const void* src, size_t n)
  # a0=dest, a1=src, a2=n
  #
memcpy:
    mv a3, a0 # Copy destination
loop:
  vsetvli t0, a2, e8,m8,ta,ma   # Vectors of 8b
  vle8.v v0, (a1)               # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
  vse8.v v0, (a3)               # Store bytes
    add a3, a3, t0              # Bump pointer
    bnez a2, loop               # Any more?
    ret                         # Return
```

Set configuration, calculate vector strip length

Unit-stride vector load elements (bytes)

Unit-stride vector store elements (bytes)

Same binary machine code can run on machines with any VLEN!

Vector Unit-Stride Loads/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
```

```
# vs3 store data, rs1 base address, vm is mask encoding (v0.t or <missing>)
vse8.v    vs3, (rs1), vm  #    8-bit unit-stride store
vse16.v   vs3, (rs1), vm  #   16-bit unit-stride store
vse32.v   vs3, (rs1), vm  #   32-bit unit-stride store
vse64.v   vs3, (rs1), vm  #   64-bit unit-stride store
```

```
for i = 0 to VLEN - 1
    vd[i] = load(rs1 + i)
```

Vector Strided Load/Store Instructions

```
# vd destination, rs1 base address, rs2 byte stride
vlse8.v    vd, (rs1), rs2, vm  #   8-bit strided load
vlse16.v   vd, (rs1), rs2, vm  #  16-bit strided load
vlse32.v   vd, (rs1), rs2, vm  #  32-bit strided load
vlse64.v   vd, (rs1), rs2, vm  #  64-bit strided load
```

```
# vs3 store data, rs1 base address, rs2 byte stride
vsse8.v    vs3, (rs1), rs2, vm  #   8-bit strided store
vsse16.v   vs3, (rs1), rs2, vm  #  16-bit strided store
vsse32.v   vs3, (rs1), rs2, vm  #  32-bit strided store
vsse64.v   vs3, (rs1), rs2, vm  #  64-bit strided store
```

```
for i = 0 to VLEN - 1
    vd[i] = load(rs1 + i*rs2)
```

Vector Indexed Loads/Stores

```
# Vector unordered indexed load instructions
# vd destination, rs1 base address, vs2 indices
vluxei8.v    vd, (rs1), vs2, vm  # unordered  8-bit indexed load of SEW data
vluxei16.v   vd, (rs1), vs2, vm  # unordered 16-bit indexed load of SEW data
vluxei32.v   vd, (rs1), vs2, vm  # unordered 32-bit indexed load of SEW data
vluxei64.v   vd, (rs1), vs2, vm  # unordered 64-bit indexed load of SEW data

# Vector ordered indexed load instructions
# vd destination, rs1 base address, vs2 indices
vloxei8.v    vd, (rs1), vs2, vm  # ordered  8-bit indexed load of SEW data
vloxei16.v   vd, (rs1), vs2, vm  # ordered 16-bit indexed load of SEW data
vloxei32.v   vd, (rs1), vs2, vm  # ordered 32-bit indexed load of SEW data
vloxei64.v   vd, (rs1), vs2, vm  # ordered 64-bit indexed load of SEW data

# Vector unordered-indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsuxei8.v   vs3, (rs1), vs2, vm # unordered  8-bit indexed store of SEW data
vsuxei16.v  vs3, (rs1), vs2, vm # unordered 16-bit indexed store of SEW data
vsuxei32.v  vs3, (rs1), vs2, vm # unordered 32-bit indexed store of SEW data
vsuxei64.v  vs3, (rs1), vs2, vm # unordered 64-bit indexed store of SEW data

# Vector ordered indexed store instructions
# vs3 store data, rs1 base address, vs2 indices
vsoxei8.v    vs3, (rs1), vs2, vm  # ordered  8-bit indexed store of SEW data
vsoxei16.v   vs3, (rs1), vs2, vm  # ordered 16-bit indexed store of SEW data
vsoxei32.v   vs3, (rs1), vs2, vm  # ordered 32-bit indexed store of SEW data
vsoxei64.v   vs3, (rs1), vs2, vm  # ordered 64-bit indexed store of SEW data
```

```
for i = 0 to VLEN - 1
    vd[i] = load(rs1 + vs2[i])
```

Index data width encoded in instruction, while data size encoded in vtype.vsew field

VLEN=256b, SLEN=128b

SEW=8b, LMUL=1, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	v1 * n + 0

SEW=16b, LMUL=2, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
	17		16		15		14		13		12		11		10		7		6		5		4		3		2		1		0	v2 * n + 0
	1F		1E		1D		1C		1B		1A		19		18		F		E		D		C		B		A		9		8	v2 * n + 1

SEW=32b, LMUL=4, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
			13				12				11				10				3				2				1				0	v4 * n + 0
			17				16				15				14				7				6				5				4	v4 * n + 1
			1B				1A				19				18				B				A				9				8	v4 * n + 2
			1F				1E				1D				1C				F				E				D				C	v4 * n + 3

SEW=64b, LMUL=8, VLMAX=32

1F	1E	1D	1C	1B	1A	19	18	17	16	15	14	13	12	11	10	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
							11								10								1								0	v8 * n + 0
							13								12								3								2	v8 * n + 1
							15								14								5								4	v8 * n + 2
							17								16								7								6	v8 * n + 3
							19								18								9								8	v8 * n + 4
							1B								1A								B								A	v8 * n + 5
							1D								1C								D								C	v8 * n + 6
							1F								1E								F								E	v8 * n + 7

LMUL=8 stripmined vector `memcpy` example

```
  # void *memcpy(void* dest, const void* src, size_t n)
  # a0=dest, a1=src, a2=n
  #
memcpy:
    mv a3, a0 # Copy destination
loop:
  vsetvli t0, a2, e8,m8,ta,ma   # Vectors of 8b
  vle8.v v0, (a1)               # Load bytes
    add a1, a1, t0              # Bump pointer
    sub a2, a2, t0              # Decrement count
  vse8.v v0, (a3)               # Store bytes
    add a3, a3, t0              # Bump pointer
    bnez a2, loop               # Any more?
    ret                         # Return
```

Set configuration, calculate vector strip length

Unit-stride vector load bytes

Unit-stride vector store bytes

Combine eight vector registers into group (v0,v1,…,v7)

Binary machine code can run on machines with any VLEN!

Masking

- Nearly all operations can be optionally under a mask (or predicate) held in vector register **`v0`**
- A single *vm* bit in instruction encoding selects whether unmasked or under control of **`v0`**
- Integer and FP compare instructions provided to set masks into any vector register
- Can perform mask logical operations between any vector registers

Vector Integer Add Instructions

```
# Integer adds.
vadd.vv vd, vs2, vs1, vm   # Vector-vector
vadd.vx vd, vs2, rs1, vm   # vector-scalar
vadd.vi vd, vs2, imm, vm   # vector-immediate

# Integer subtract
vsub.vv vd, vs2, vs1, vm   # Vector-vector
vsub.vx vd, vs2, rs1, vm   # vector-scalar

# Integer reverse subtract
vrsub.vx vd, vs2, rs1, vm   # vd[i] = rs1 - vs2[i]
vrsub.vi vd, vs2, imm, vm   # vd[i] = imm - vs2[i]
```

Integer Compare Instructions

```
Comparison      Assembler Mapping             Assembler Pseudoinstruction

va < vb         vmslt{u}.vv vd, va, vb, vm
va <= vb        vmsle{u}.vv vd, va, vb, vm
va > vb         vmslt{u}.vv vd, vb, va, vm    vmsgt{u}.vv vd, va, vb, vm
va >= vb        vmsle{u}.vv vd, vb, va, vm    vmsge{u}.vv vd, va, vb, vm

va < x          vmslt{u}.vx vd, va, x, vm
va <= x         vmsle{u}.vx vd, va, x, vm
va > x          vmsgt{u}.vx vd, va, x, vm
va >= x         see below

va < i          vmsle{u}.vi vd, va, i-1, vm    vmslt{u}.vi vd, va, i, vm
va <= i         vmsle{u}.vi vd, va, i, vm
va > i          vmsgt{u}.vi vd, va, i, vm
va >= i         vmsgt{u}.vi vd, va, i-1, vm    vmsge{u}.vi vd, va, i, vm

va, vb vector register groups
x      scalar integer register
i      immediate
```

Mask Logical Operations

```
    vmand.mm vd, vs2, vs1     # vd[i] =   vs2[i].LSB &&  vs1[i].LSB
    vmnand.mm vd, vs2, vs1    # vd[i] = !(vs2[i].LSB &&  vs1[i].LSB)
    vmandnot.mm vd, vs2, vs1  # vd[i] =   vs2[i].LSB && !vs1[i].LSB
    vmxor.mm  vd, vs2, vs1    # vd[i] =   vs2[i].LSB ^^  vs1[i].LSB
    vmor.mm  vd, vs2, vs1     # vd[i] =   vs2[i].LSB ||  vs1[i].LSB
    vmnor.mm  vd, vs2, vs1    # vd[i] = !(vs2[i[.LSB ||  vs1[i].LSB)
    vmornot.mm  vd, vs2, vs1  # vd[i] =   vs2[i].LSB || !vs1[i].LSB
    vmxnor.mm vd, vs2, vs1    # vd[i] = !(vs2[i].LSB ^^  vs1[i].LSB)
```

Several assembler pseudoinstructions are defined as shorthand for common uses of mask logical operations:

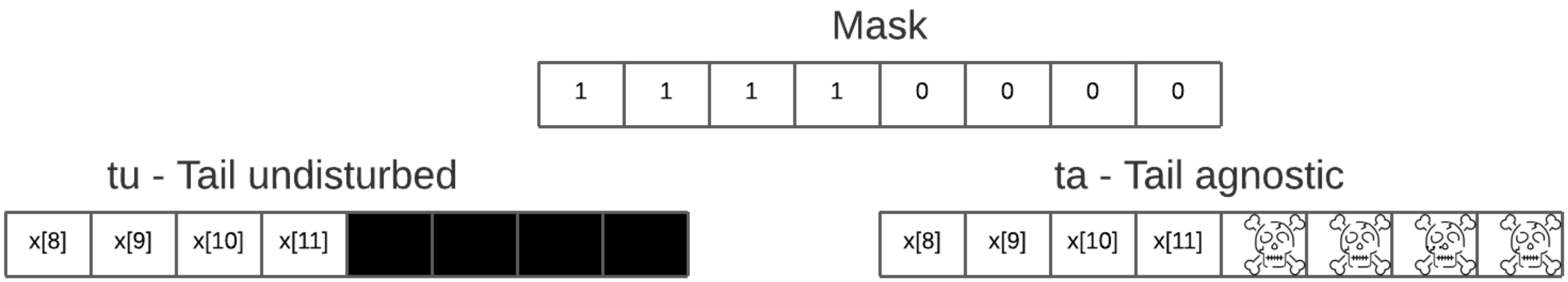
```
    vmcpy.m vd, vs  => vmand.mm vd, vs, vs  # Copy mask register
    vmclr.m vd     => vmxor.mm vd, vd, vd   # Clear mask register
    vmset.m vd     => vmxnor.mm vd, vd, vd  # Set mask register
    vmnot.m vd, vs => vmnand.mm vd, vs, vs  # Invert bits
```

Agnostic vs Undisturbed

vsetvli t0, a0, e32, m1, ta, ma

ta – Tail agnostic
tu – Tail undisturbed

ma – Mask agnostic
mu – Mask undisturbed


Mask
1
1
1
1
0
0
0
0
tu - Tail undisturbed
x[8]
x[9]
x[10]
x[11]
ta - Tail agnostic
x[8]
x[9]
x[10]
x[11]