An Open-Source RISC-V Vector Math Library

Ping Tak Peter Tang
Rivos Inc.
June 11, 2024
ARITH 2024, Malaga, Spain

An Open-Source RISC-V VecLibm

- Background and Motivation
- Snapshots of RISC-V Vector ISA
- RISC-V Vector Math Library: Strategies and Illustrations
- Rivos FP64 Vector Libm current status – at a glance

Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible

Background and Motivation

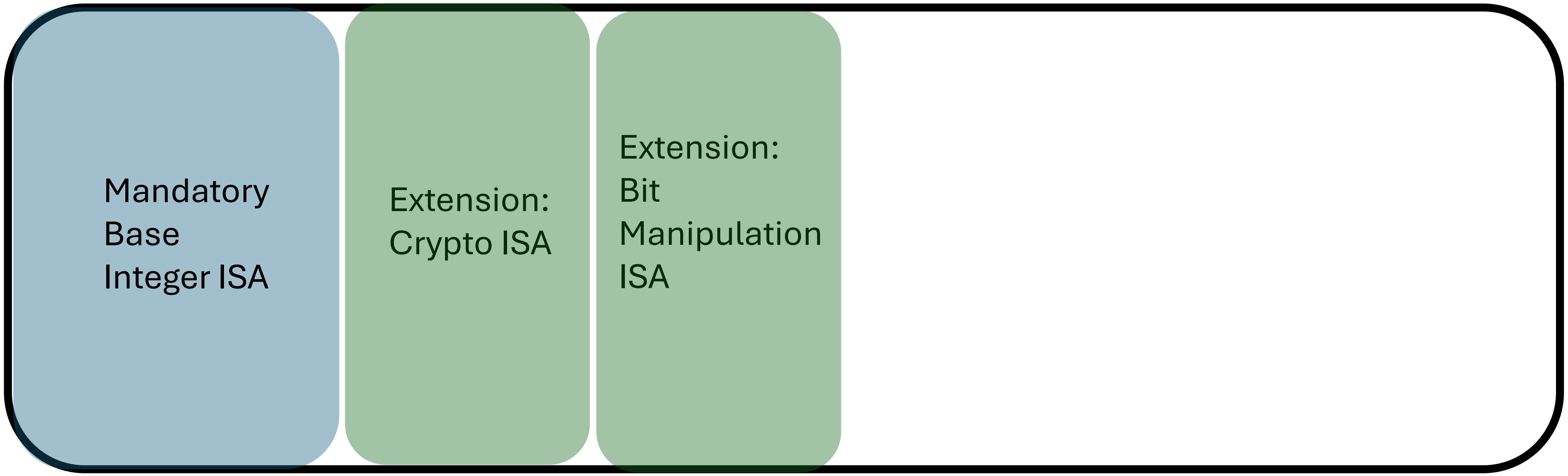
- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible

Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible

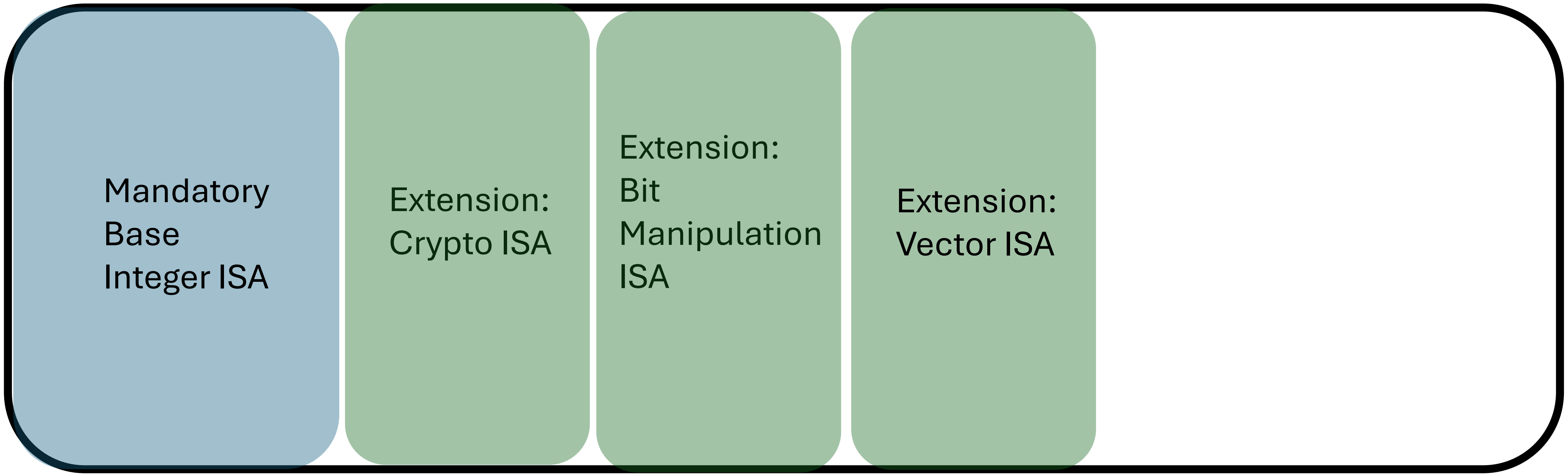
Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible



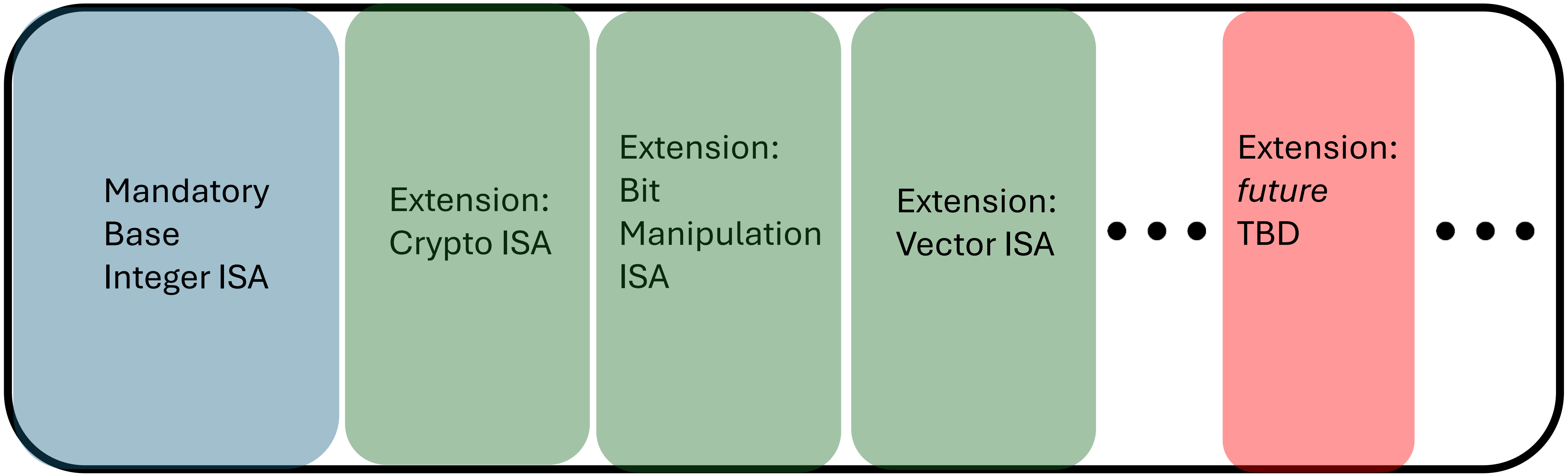
Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible



Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible



Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible
- Open ISA: accelerates innovations via robust ecosystems

Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible
- Open ISA: accelerates innovations via robust ecosystems
- Open RISC-V Vector Libms are worthy additions
  - FP64 vector libm fits the need to traditional computational science and HPC

Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible
- Open ISA: accelerates innovations via robust ecosystems
- Open RISC-V Vector Libms are worthy additions
  - FP64 vector libm fits the need to traditional computational science and HPC

In `math.h` `exp(x)`, `sin(x)`, `atan(x)`, etc.

Background and Motivation

- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible
- Open ISA: accelerates innovations via robust ecosystems
- Open RISC-V Vector Libms are worthy additions
  - FP64 ***vector*** libm fits the need to traditional computational science and HPC

In `math.h` `exp(x)`, `sin(x)`, `atan(x)`, etc.

Scalar:
```
double exp(double x);
```

VS.

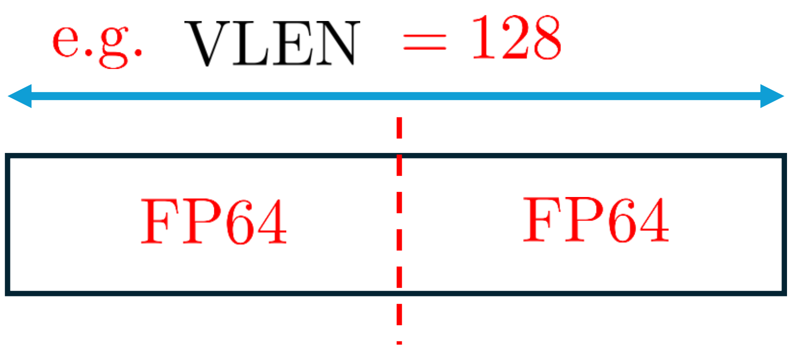
Vector:
```
void vec_exp(int N, const double* x, double* y);
```

Background and Motivation

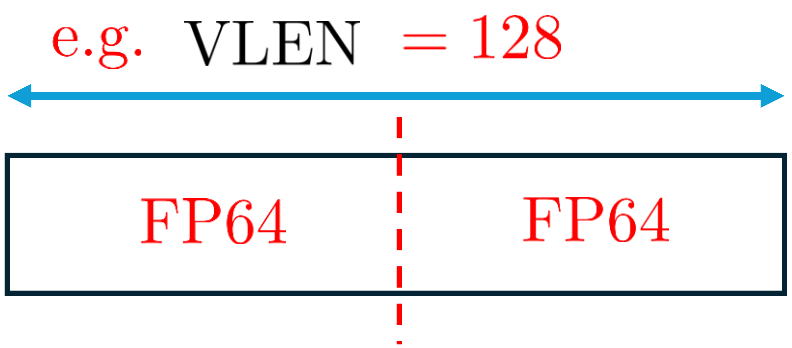
- RISC-V: An open ISA first developed in 2011
- Two distinguished features: modular and extensible
- Open ISA: accelerates innovations via robust ecosystems
- Open RISC-V Vector Libms are worthy additions
  - FP64 vector libm fits the need to traditional computational science and HPC
  - Requires experience to construct a numerically reliable library
  - Scope is modest that a start up can undertake as a good-citizen project

Snapshots of RISC-V Vector ISA: *General*

32 architectural registers
VLEN implementation defined



32 architectural registers
VLEN implementation defined

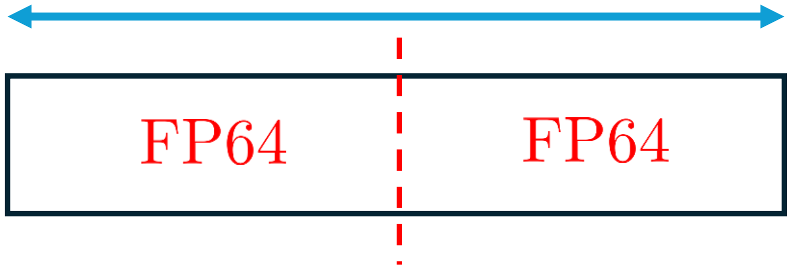


Registers are type agnostic:
Just a number of bits,
to be interpreted in the context
of the instructions

32 architectural registers
VLEN implementation defined

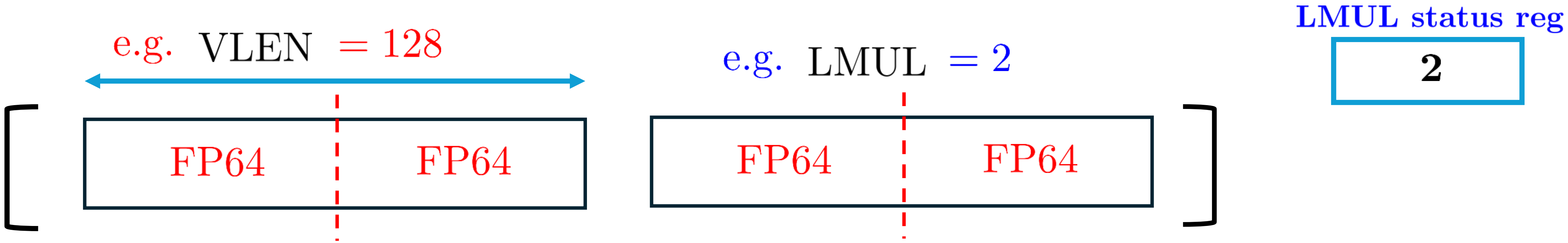
Can configure; group LMUL physical regs to form one logical vector reg.

e.g. VLEN $= 128$



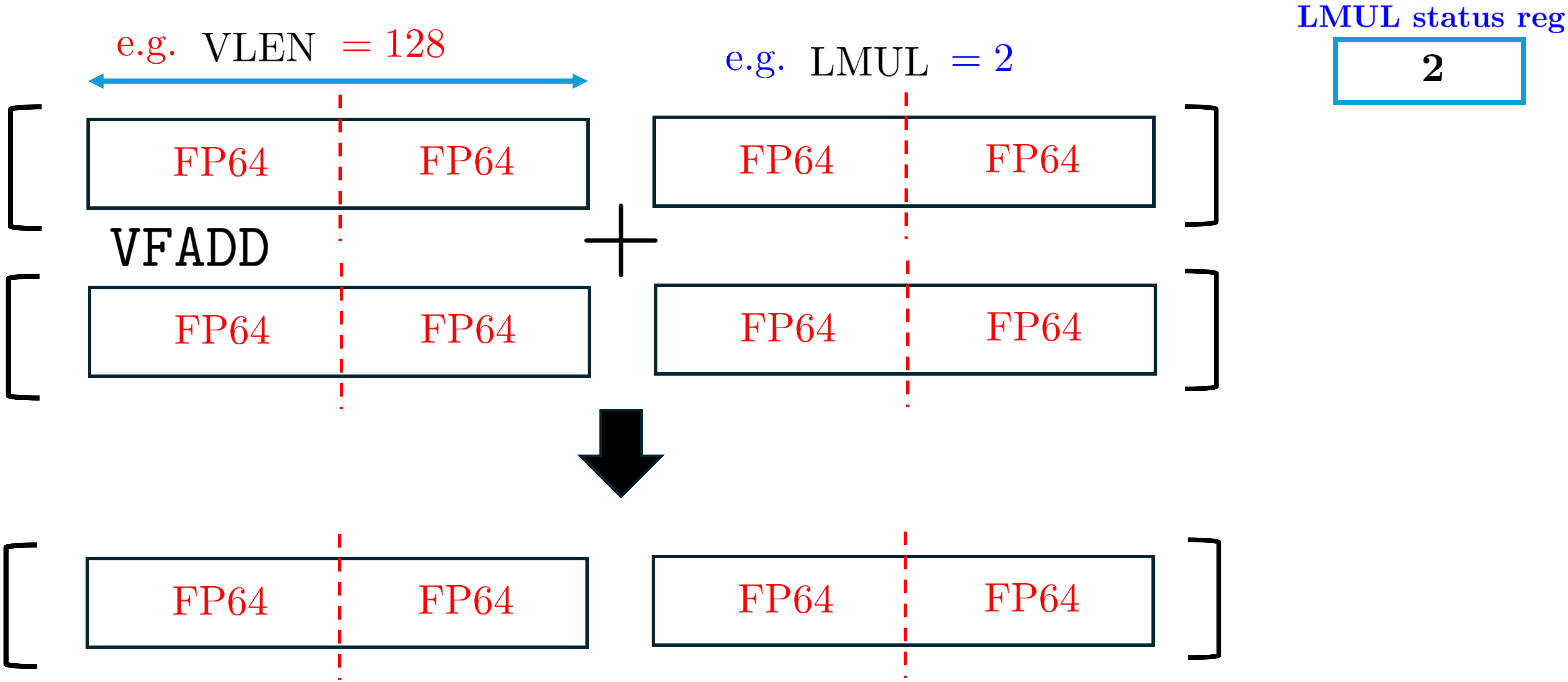
32 architectural registers
VLEN implementation defined

Can configure; group LMUL physical regs
to form one logical vector reg.



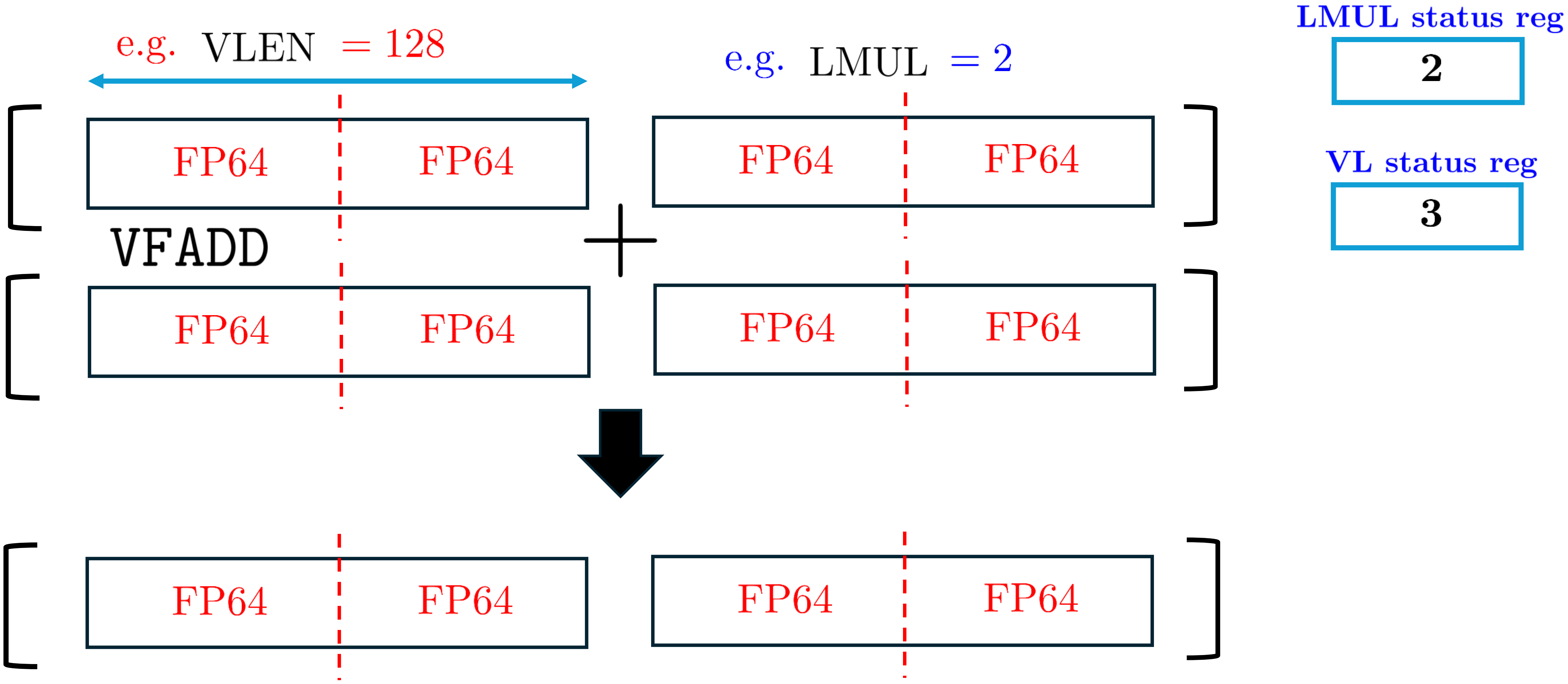
32 architectural registers
VLEN implementation defined

Can configure; group LMUL physical regs to form one logical vector reg.



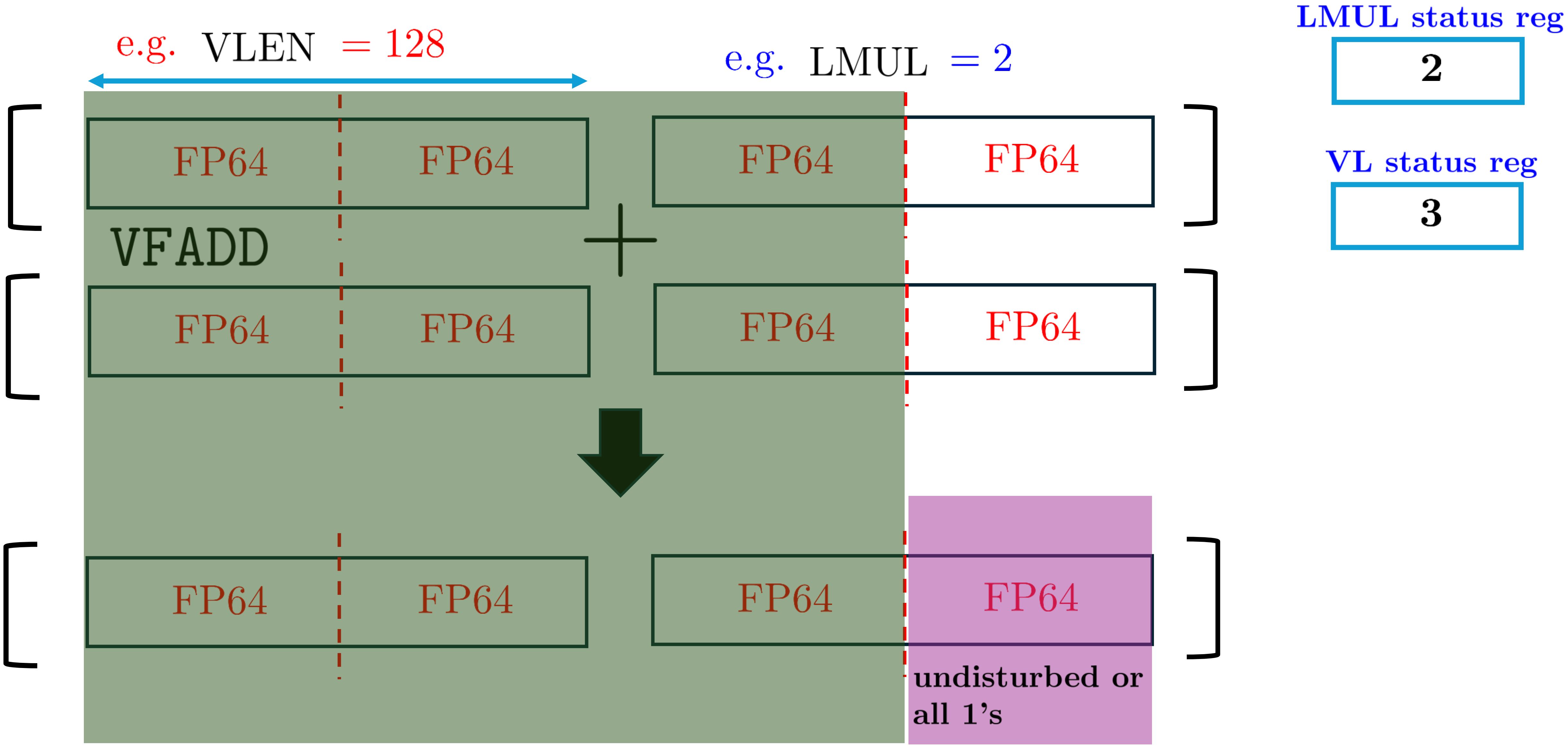
32 architectural registers
VLEN implementation defined

Can configure; group LMUL physical regs to form one logical vector reg.



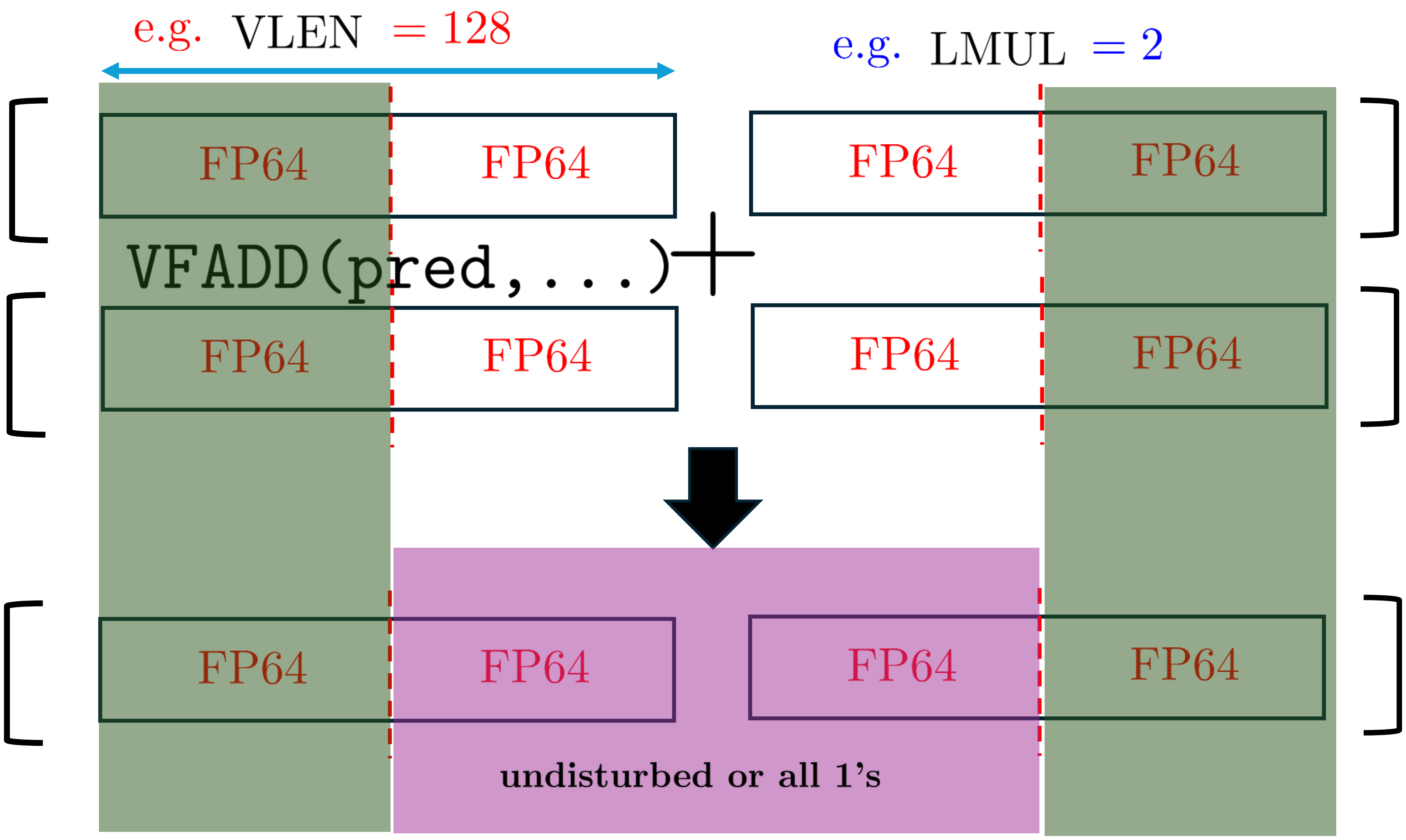
32 architectural registers
VLEN implementation defined

Can configure; group LMUL physical regs
to form one logical vector reg.



32 architectural registers
VLEN implementation defined

Can configure; group LMUL physical regs
to form one logical vector reg.



LMUL status reg

2

VL status reg

4

predicated execution

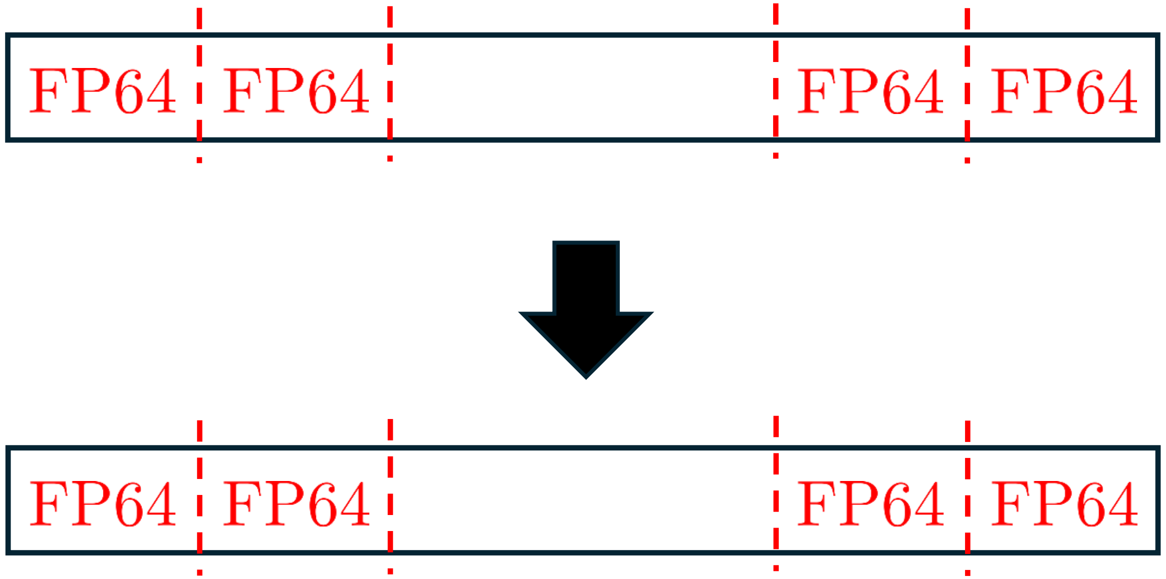
pred. reg

1 0 0 1 • • •

Snapshots of RISC-V Vector ISA: *Floating-Pt.*

Some Useful Vector Instructions for Veclibm

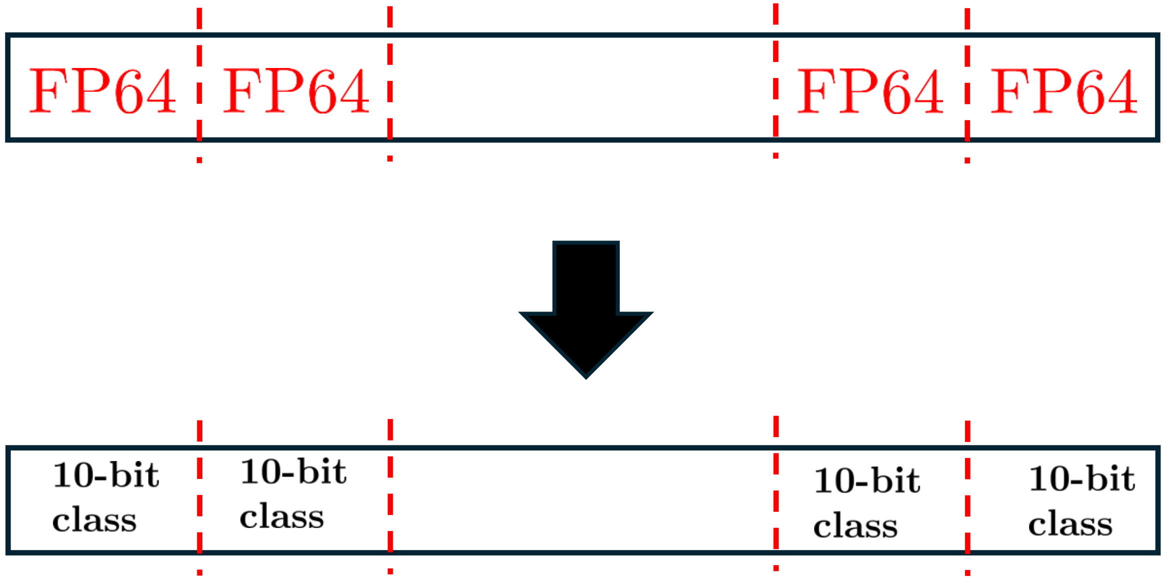
`VFREC7, VFRSQRT, VFSQRT, etc`



Unary FP instructions:
`vfrec7, vfrsqrt7, vfsqrt`

Some Useful Vector Instructions for Veclibm

`VFCLASS`

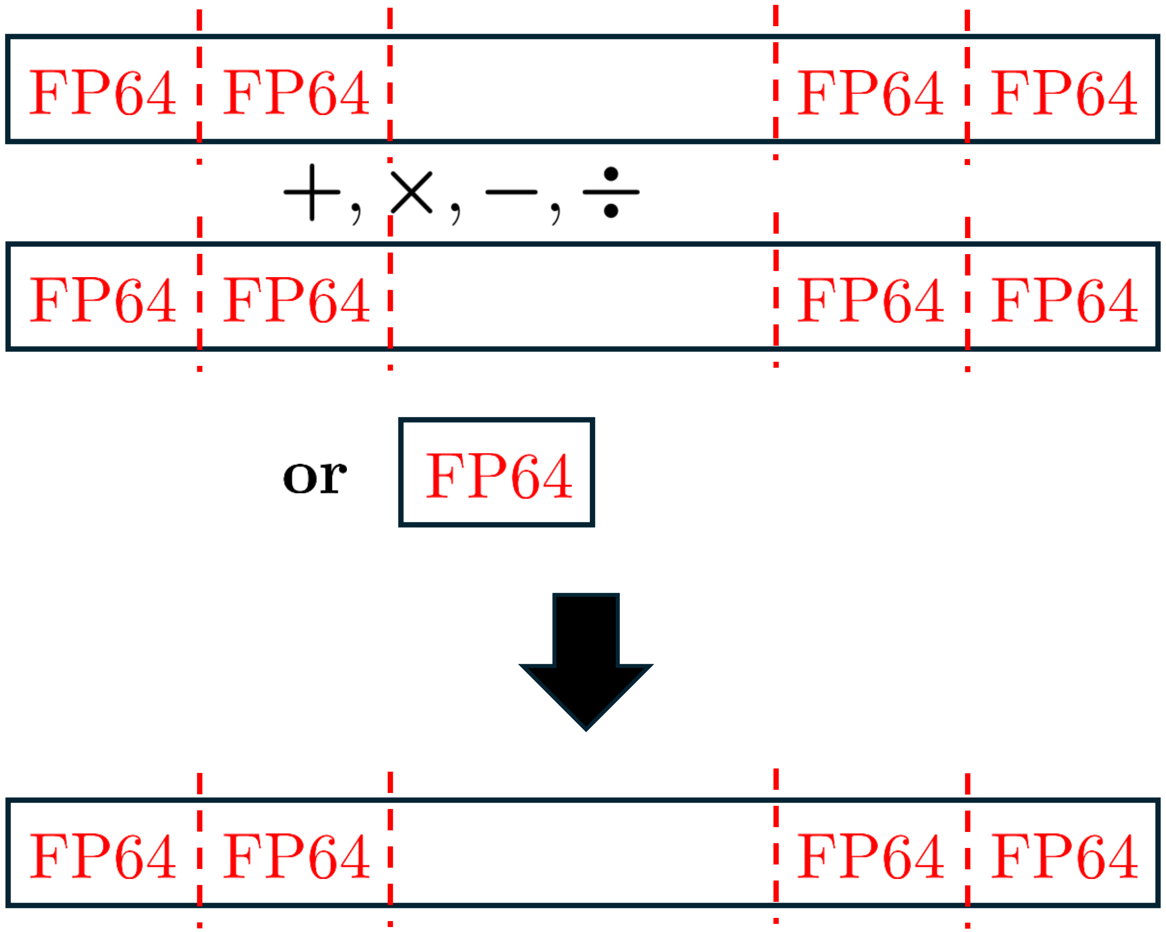


Unary FP instructions:

`vfrec7, vfrsqrt7, vfsqrt`
`vfclass`

rd bit	Meaning
0	*rs1* is $-\infty$.
1	*rs1* is a negative normal number.
2	*rs1* is a negative subnormal number.
3	*rs1* is -0.
4	*rs1* is $+0$.
5	*rs1* is a positive subnormal number.
6	*rs1* is a positive normal number.
7	*rs1* is $+\infty$.
8	*rs1* is a signaling NaN.
9	*rs1* is a quiet NaN.

Some Useful Vector Instructions for Veclibm



Unary FP instructions:

`vfrec7`, `vfrsqrt7`, `vfsqrt`

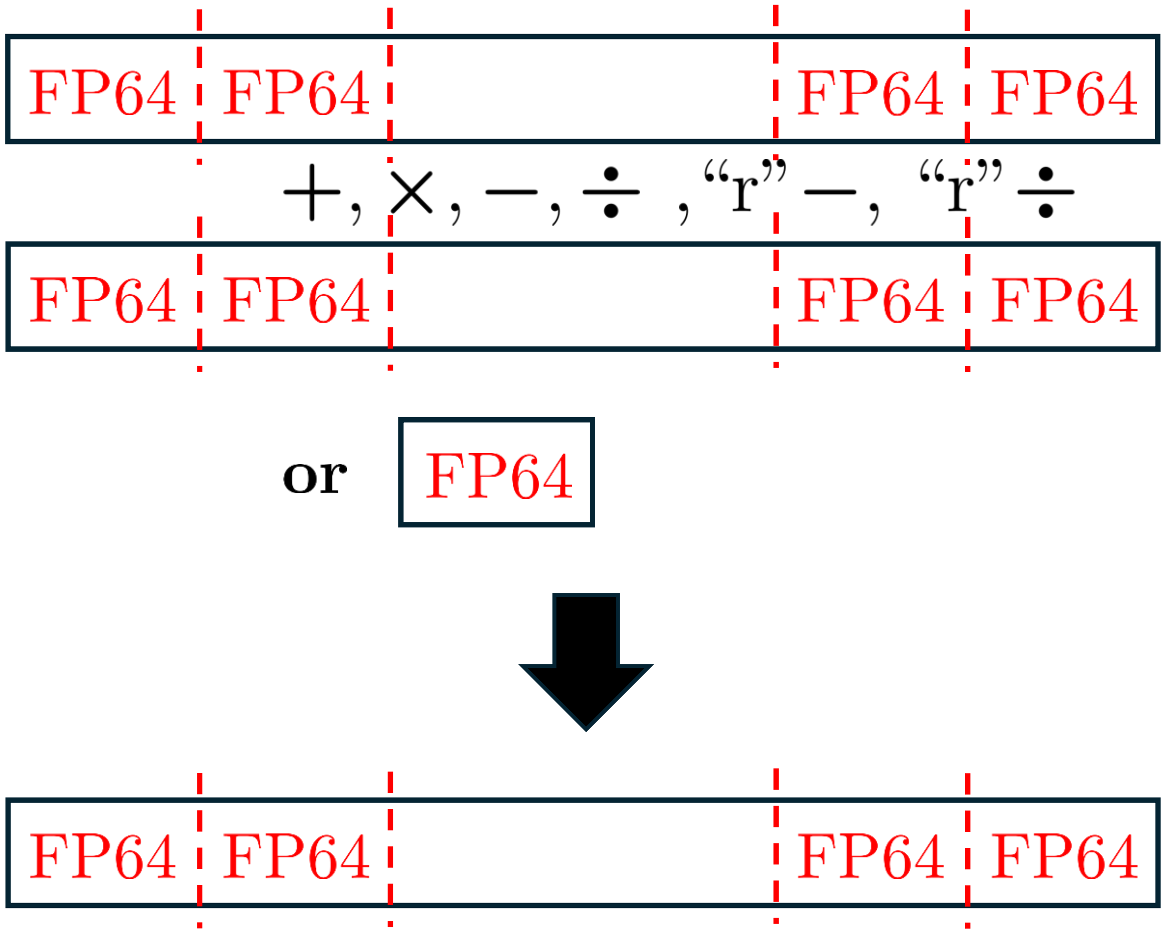
`vfclass`

Binary FP instructions:

`vfadd`, `vfmul`

`vfsub`, `vfdiv`

Some Useful Vector Instructions for Veclibm



Unary FP instructions:

`vfrec7`, `vfrsqrt7`, `vfsqrt`

`vfclass`

Binary FP instructions:

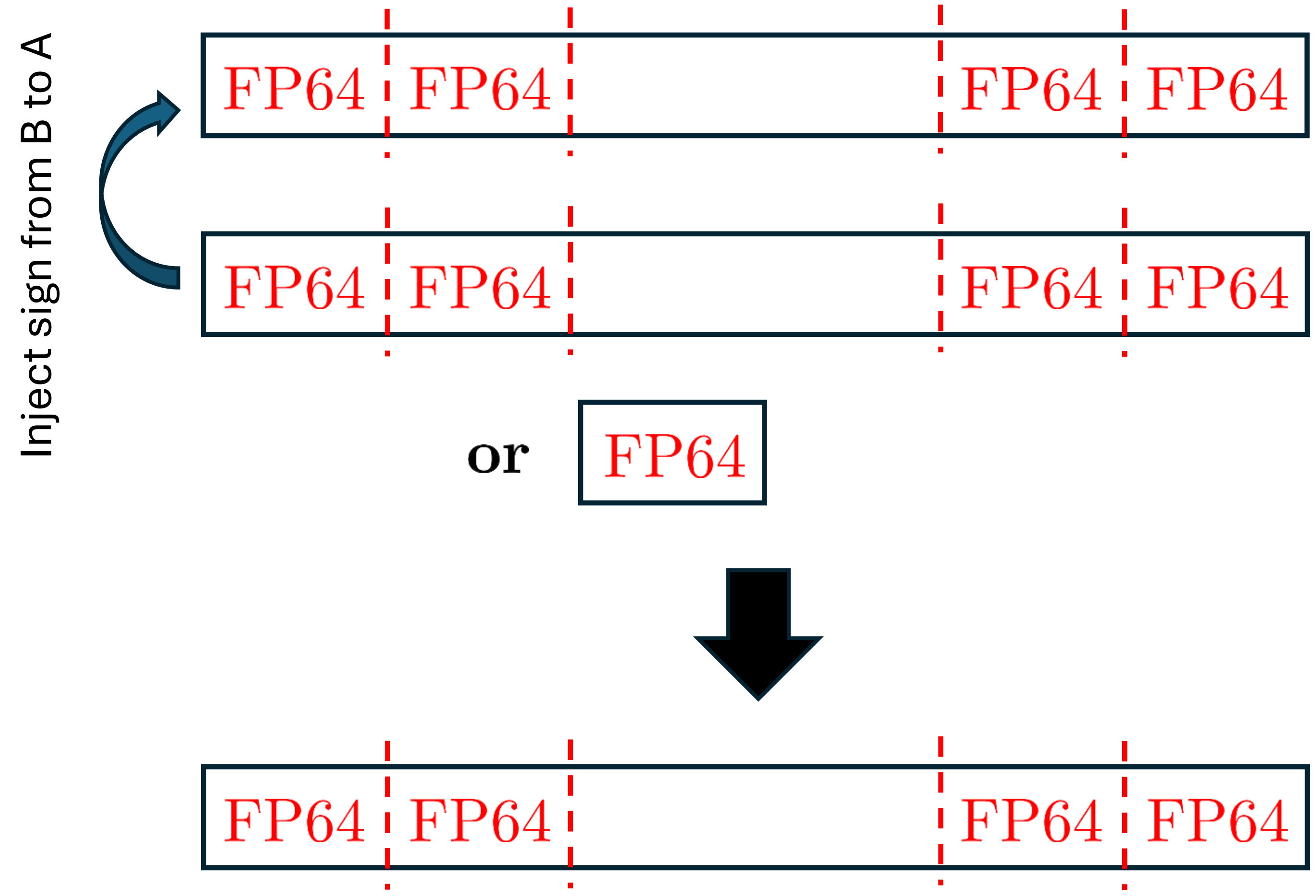
`vfadd`, `vfmul`

`vfsub`, `vfdiv`

`vfrsub`, `vfrdiv`

Some Useful Vector Instructions for Veclibm

`vfsgnj, vfsgnjn, vfsgnjx`



Unary FP instructions:

`vfrec7, vfrsqrt7, vfsqrt`

`vfclass`

Binary FP instructions:

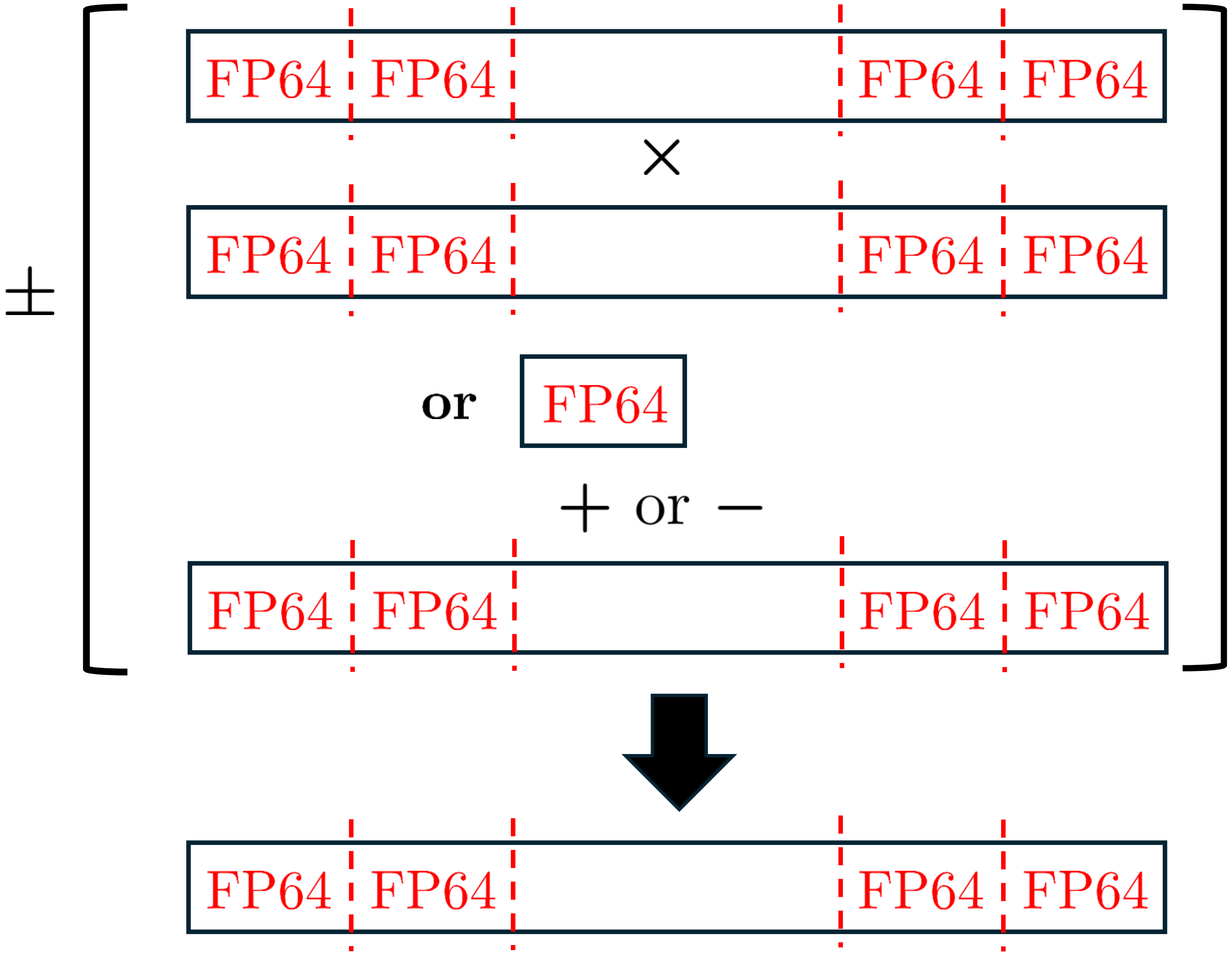
`vfadd, vfmul`

`vfsub, vfdiv`

`vfrsub, vfrdiv`

`vfsgnj, vfsgnjn, vfsgnjx`

Some Useful Vector Instructions for Veclibm

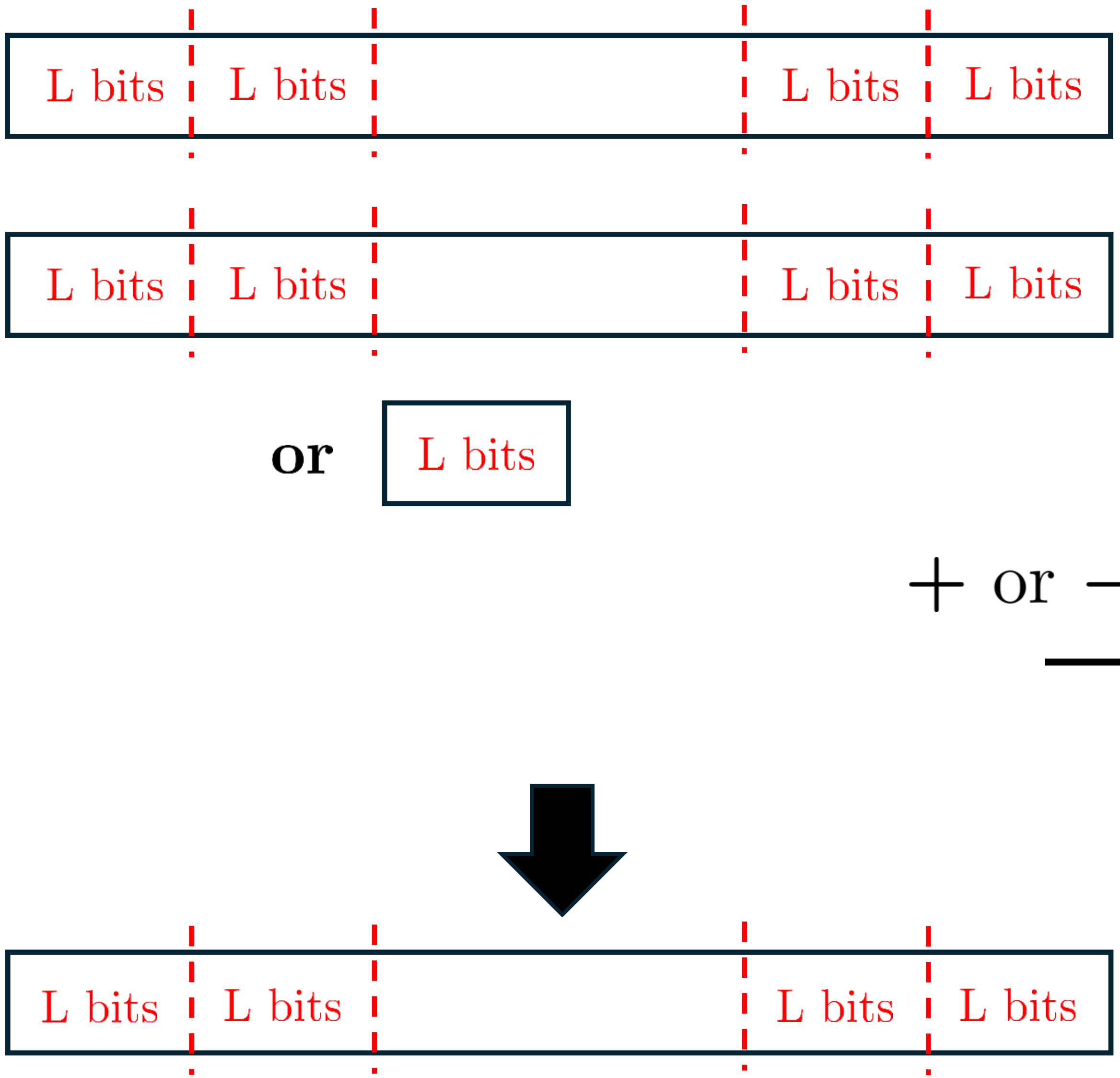


Unary FP instructions:
`vfrec7`, `vfrsqrt7`, `vfsqrt`
`vfclass`

Binary FP instructions:
`vfadd`, `vfmul`
`vfsub`, `vfdiv`
`vfrsub`, `vfrdiv`
`vfsgnj`, `vfsgnjn`, `vfsgnjx`

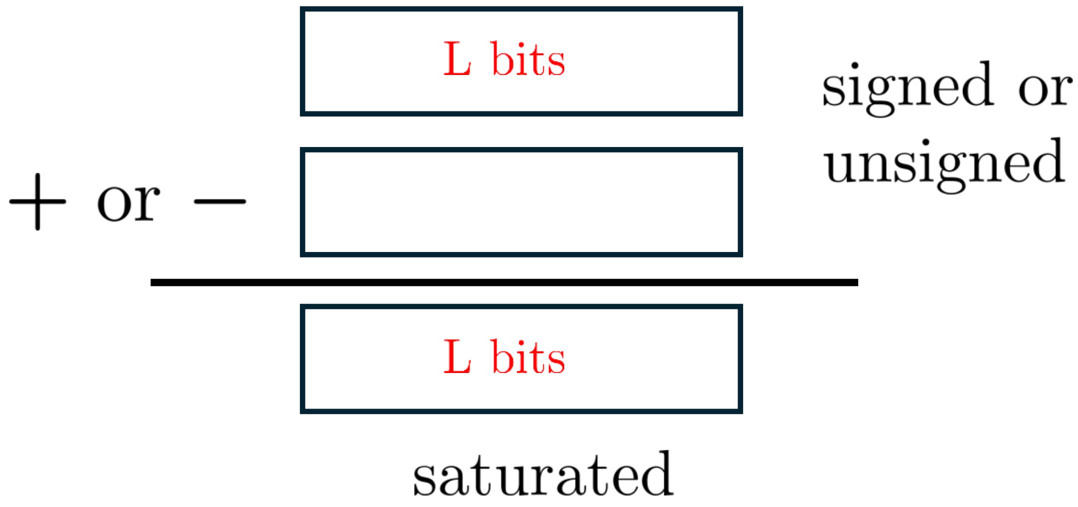
Ternary FP instructions:
`vf[n]madd`, `vf[n]msub`
`vf[n]macc`, `vf[n]msac`

Some Useful Vector Instructions for Veclibm

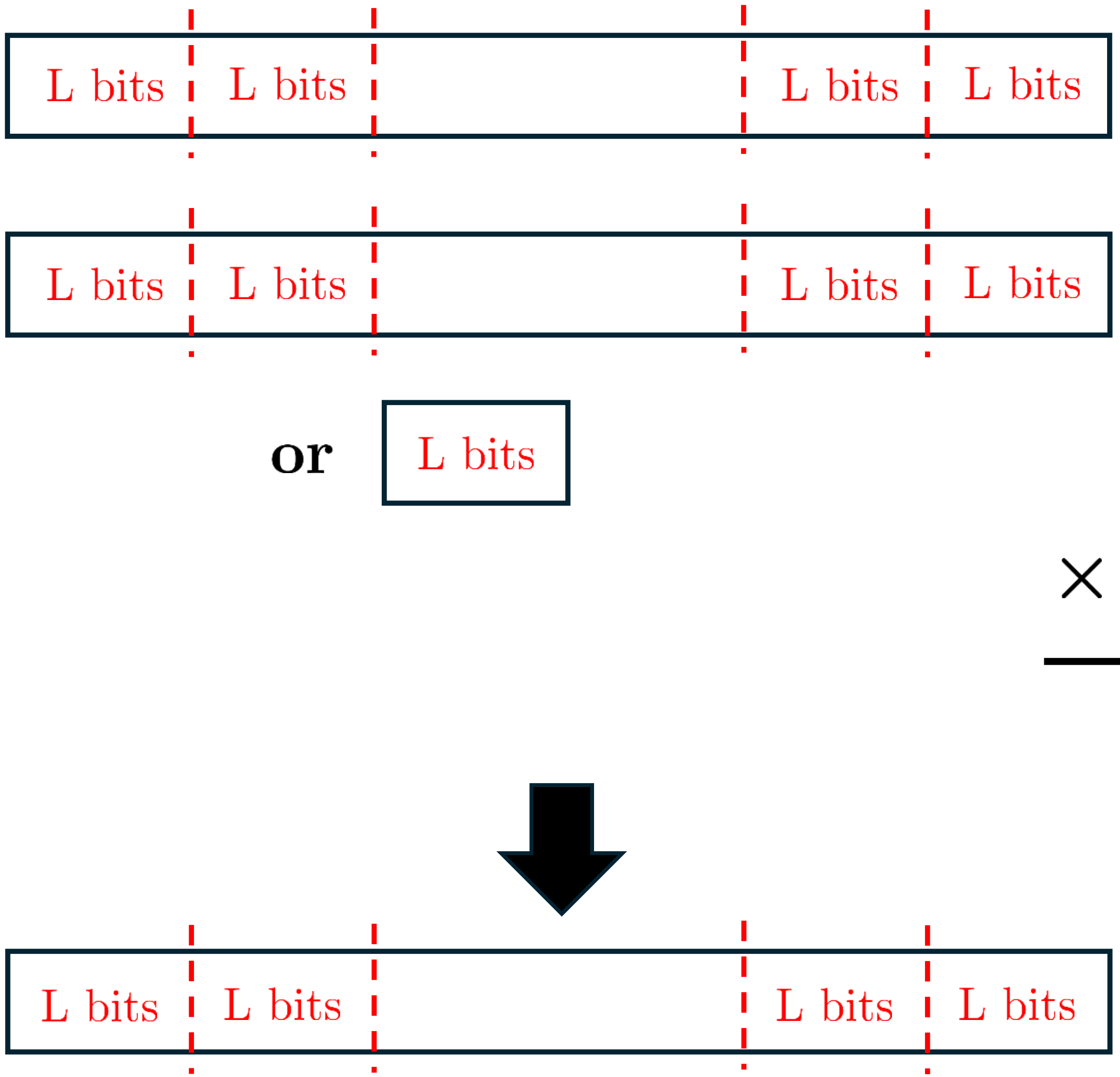


Fixed-point arithmetic:

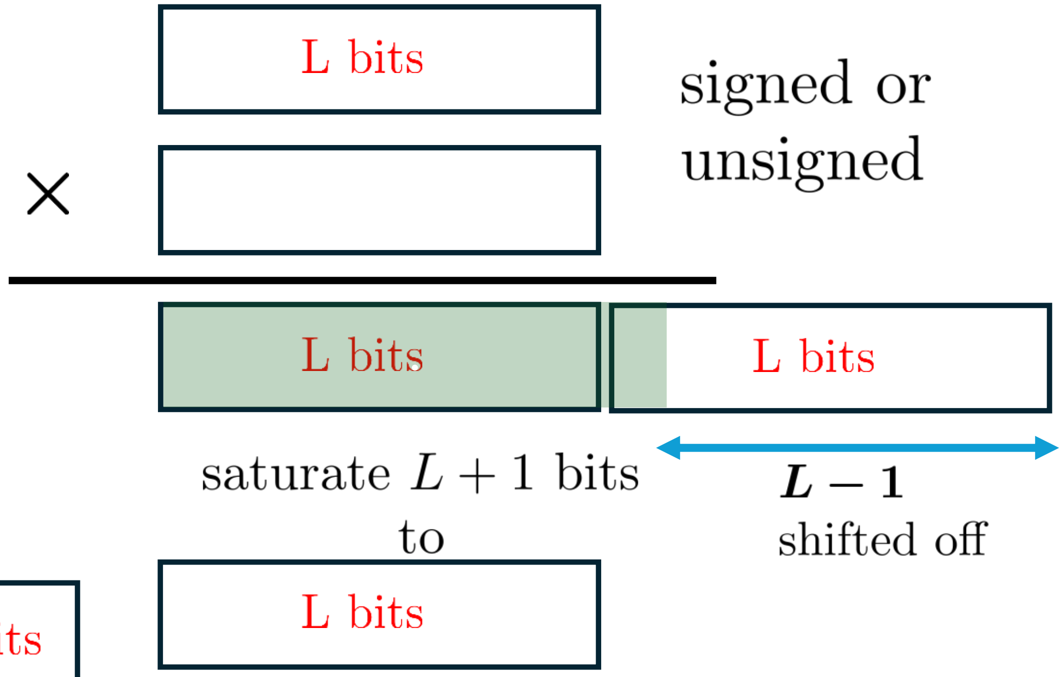
`vsadd`, `vssub`, `vaadd`, `vasub`
`vsmul`



Some Useful Vector Instructions for Veclibm



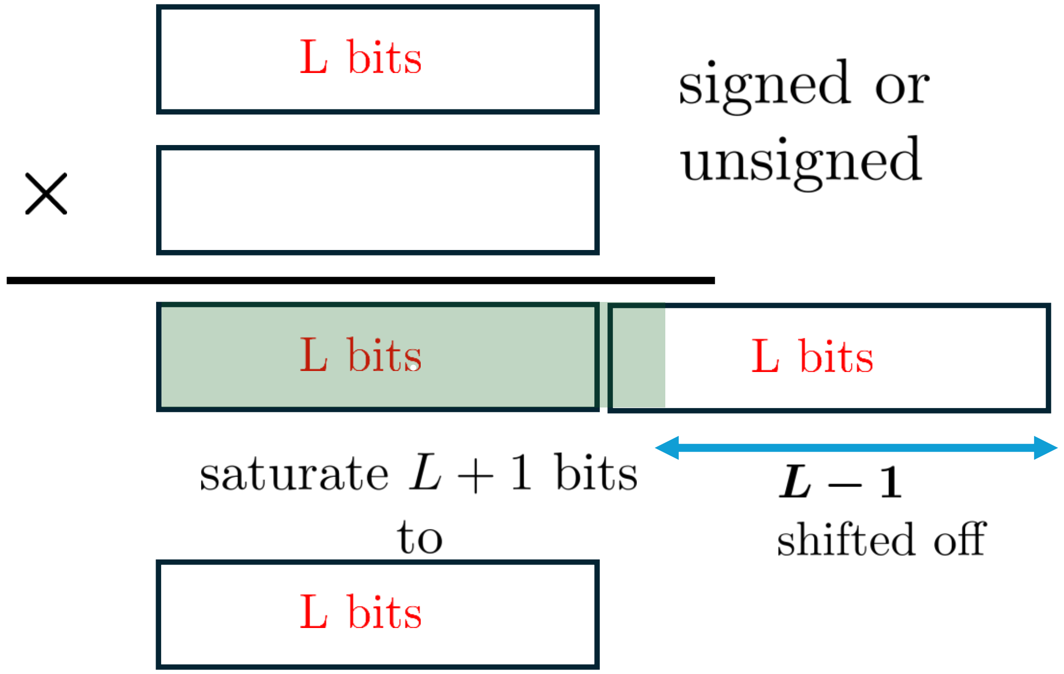
Fixed-point arithmetic:
`vsadd`, `vssub`, `vaadd`, `vasub`
`vsmul`



Some Useful Vector Instructions for Veclibm

$A = \text{int}(a \times 2^{\sigma_a})$ (A is a in Q-σ_a format)
$B = \text{int}(b \times 2^{\sigma_b})$ (B is b in Q-σ_b format)
$C := \texttt{vsmul}(A, B)$ C is ab in σ_c format
$\sigma_c = \sigma_a + \sigma_b - 63$

Fixed-point arithmetic:
`vsadd`, `vssub`, `vaadd`, `vasub`
`vsmul`

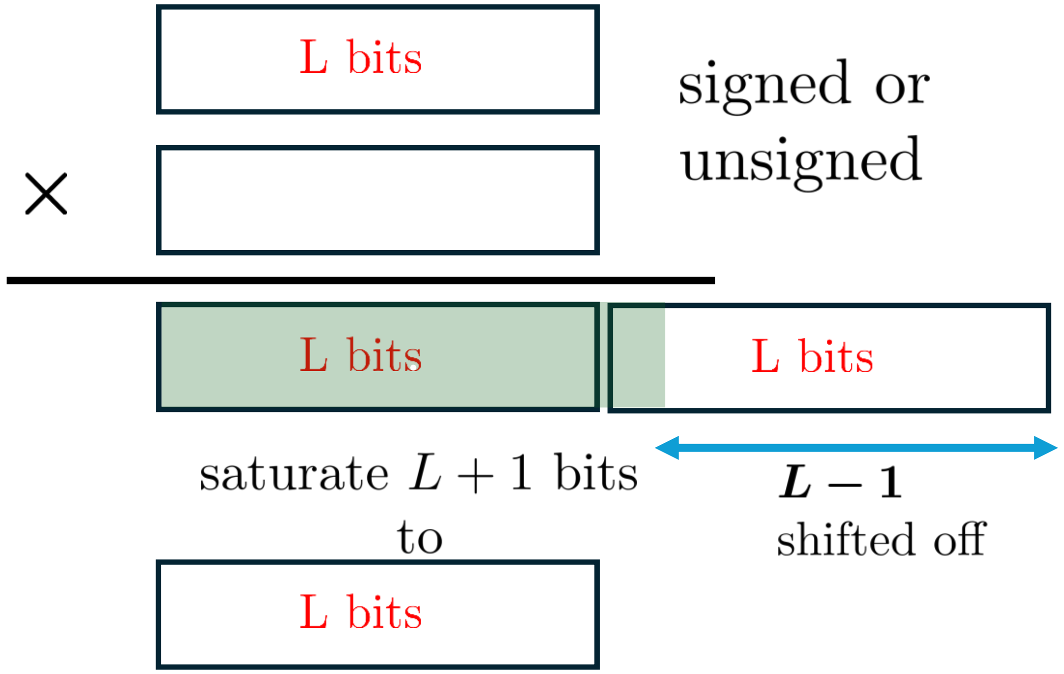


Some Useful Vector Instructions for Veclibm

$A = \text{int}(a \times 2^{\sigma_a})$ (A is a in Q-σ_a format)
$B = \text{int}(b \times 2^{\sigma_b})$ (B is b in Q-σ_b format)
$C := \texttt{vsmul}(A, B)$ C is ab in σ_c format
$\sigma_c = \sigma_a + \sigma_b - 63$

Fixed-point arithmetic:
`vsadd`, `vssub`, `vaadd`, `vasub`
`vsmul`

Fixed-point arithmetic can potentially carry 63 bits of precision



Some Useful Vector Instructions for Veclibm

$A = \mathrm{int}(a \times 2^{\sigma_a})$ (A is a in Q-σ_a format)
$B = \mathrm{int}(b \times 2^{\sigma_b})$ (B is b in Q-σ_b format)
$C := \texttt{vsmul}(A, B)$ C is ab in σ_c format
$\sigma_c = \sigma_a + \sigma_b - 63$

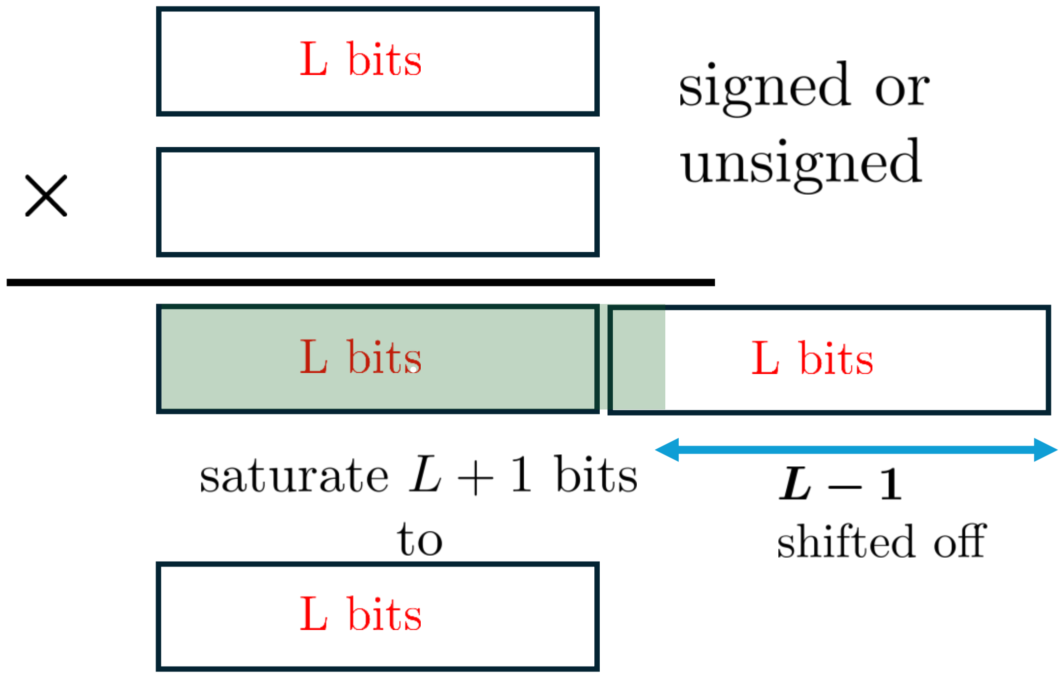
Choice for scales are flexible: but 63 is nice

For example, Horner's recurrence:
$P_0 + R * (P_1 + R * (P_2 + \cdots))$

All P_j having the same scale and R being scaled at 63 eliminates the need for manual shifting

Fixed-point arithmetic:
`vsadd`, `vssub`, `vaadd`, `vasub`
`vsmul`

signed or unsigned



VecLibm: *Strategies and Illustrations*

- Exception Handling
- Precision Preservation

Exception Handling

- Challenge: Need to minimize branching while returning correct value AND signal.

Exception Handling

- Challenge: Need to minimize branching while returning correct value AND signal.
- General strategy

If there is some $x \in$ exceptional arguments

set flag `x_special`
set result `y_special` for special inputs
substitute special inputs with safe values (e.g. 0.0 or 1.0)

Exception Handling

- Challenge: Need to minimize branching while returning correct value AND signal.
- General strategy

If there is some $x \in$ exceptional arguments

set flag `x_special`
set result `y_special` for special inputs
substitute special inputs with safe values (e.g. 0.0 or 1.0)

Compute result `y_normal` for all (modified) inputs
`vmerge(y_normal, y_special, x_special)`

Exception Handling: example

`log(x)`

Value	Result	Signal
$+\infty$	$+\infty$	None
qNaN	qNaN	None
sNaN, -ve	qNaN	invalid
± 0	$-\infty$	divide by 0

Exception Handling: example

`log(x)`

Value	Result	Signal
$+\infty$	$+\infty$	None
qNaN	qNaN	None
sNaN, -ve	qNaN	invalid
± 0	$-\infty$	divide by 0

```
class = vfclass(vx)
x_special = and(class,0x3BF) > 0
if (vcpop(x_special) > 0){
// handle exceptions
  ..substitute -ve with sNaN
  ..substitute +0 with -0
  y_special = vfadd(vx, vfrec7(vx))
  ..substitute special x with 1.0
}
```

Exception Handling: example

`log(x)`

Value	Result	Signal
$+\infty$	$+\infty$	None
qNaN	qNaN	None
sNaN, -ve	qNaN	invalid
± 0	$-\infty$	divide by 0

```
class = vfclass(vx)
x_special = and(class,0x3BF) > 0
if (vcpop(x_special) > 0){
// handle exceptions
  ..substitute -ve with sNaN
  ..substitute +0 with -0
 y_special = vfadd(vx, vfrec7(vx))
  ..substitute special x with 1.0
}

...compute with vx as input
...yielding y_normal as result
y_result = vmerge(y_normal,
                  y_special, x_special)
```

Exception Handling: example

`asinpi(x)`

Value	Result	Signal
qNaN	qNaN	None
sNaN, $\lvert x\rvert > 1$	qNaN	invalid
± 1	$\pm\frac{1}{2}$	None

Exception Handling: example

`asinpi(x)`

Value	Result	Signal
qNaN	qNaN	None
sNaN, $\lvert x\rvert > 1$	qNaN	invalid
± 1	$\pm\frac{1}{2}$	None

```
expo = ((vx >> 52) & 0x7FF)
x_special = expo >= 0x3FF
if (vcpop(x_special) > 0){
//handle exceptions
  ..substitute |x|>1 with sNaN
  ..substitute +-1 with +-1/4
  y_special = vfadd(vx, vx)
  ..substitute special x with 0.0
}
```

Exception Handling: example

```
expo = ((vx >> 52) & 0x7FF)
x_special = expo >= 0x3FF
if (vcpop(x_special) > 0){
//handle exceptions
  ..substitute |x|>1 with sNaN
  ..substitute +-1 with +-1/4
 y_special = vfadd(vx, vx)
  ..substitute special x with 0.0
}
```

`asinpi(x)`

Value	Result	Signal
qNaN	qNaN	None
sNaN, $\|x\| > 1$	qNaN	invalid
± 1	$\pm\frac{1}{2}$	None

```
...compute with vx as input
...yielding y_normal as result
y_result = vmerge(y_normal,
              y_special, x_special)
```

Precision Preservation

Precision Preservation

- General algorithmic approach well understood

Three steps to compute $\exp(x)$

$$
\begin{aligned}
r &\approx x - n\log(2); && \text{reduction} \\
p &\approx \exp(r); && \text{approximation} \\
e^x &\approx 2^n p && \text{reconstruction}
\end{aligned}
$$

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation

Three steps to compute $\exp(x)$

$$
\begin{aligned}
r &\approx x - n\log(2); && \text{reduction} \\
p &\approx \exp(r); && \text{approximation} \\
e^x &\approx 2^n p && \text{reconstruction}
\end{aligned}
$$

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)

Operations	# ops, an op is $+, -, \times$ or $\texttt{fmadd}$
$d(dd) + d(dd) \rightarrow dd$	6, 7, 8
$d(dd) \times d(dd) \rightarrow dd$	2, 3, 4
$d(dd)/d(dd) \rightarrow dd$	3, 4, 5 plus 1 div
$\sqrt{d(dd)} \rightarrow dd$	3, 4 plus 1 sqrt and 1 div

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)
- Our general strategy:
  - Situation-specific extra-precision computation

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)
- Our general strategy:
  - Situation-specific extra-precision computation

Fast2Sum: (3 ops, not 6)
$S := \texttt{vfadd}(A, B)$
$s := \texttt{vfadd}(\texttt{vfsub}(A, S), B)$

Works if $|A| \geq |B|$
also works if $\mathrm{lsb}(A) \geq \mathrm{lsb}(B)$

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)
- Our general strategy:
  - Situation-specific extra-precision computation

Fast2FMA: $AB + C$ (3 ops)
$S := \texttt{vfmadd}(A, B, C)$
$s := \texttt{vfmadd}(A, B, \texttt{vfsub}(C, S))$

Works if $C - S$ is exact

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)
- Our general strategy:
  - Situation-specific extra-precision computation
  - Leverage mixed fixed-and-floating-point arithmetic

Precision Preservation

- General algorithmic approach well understood
- Main challenge is precision preservation
  - General double-double simulation is costly (though is a reliable Plan-B)
- Our general strategy:
  - Situation-specific extra-precision computation
  - Leverage mixed fixed-and-floating-point arithmetic

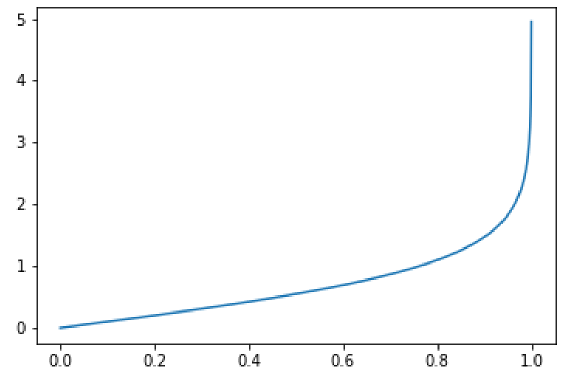
For example

$$
\begin{aligned}
y &:= p_k + r \times (p_{k+1} + r \times (p_{k+2} + \cdots)) \text{ in floating-point} \\
Y &:= \texttt{vfcvt\_x}(\texttt{vfmul}(2^q, y)) \text{ (convert to fixed point)} \\
Y &:= P_0 + R \times (P_1 + R \times (P_2 + ... + R \times (P_{k-1} + R \times Y))) \\
y &:= \texttt{vfmul}(\texttt{vfcvt\_f}(Y), 2^{-q})
\end{aligned}
$$

Precision Preservation: An example

Compute for $0 < x < 1$

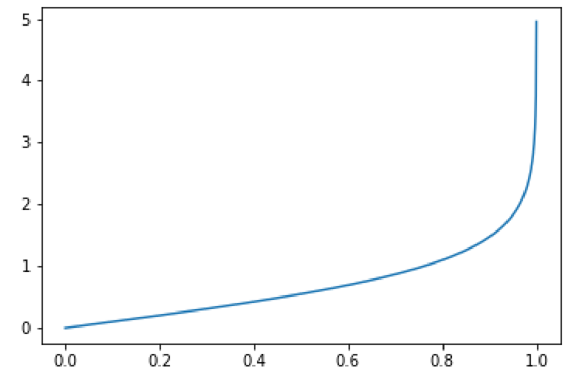
$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$



Precision Preservation: An example

Compute for $0 < x < 1$

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$



Some implementations:

get $1 \pm x$ and their quotient in dd

feed to special `log` that takes dd inputs

which in turn scales and transform in dd

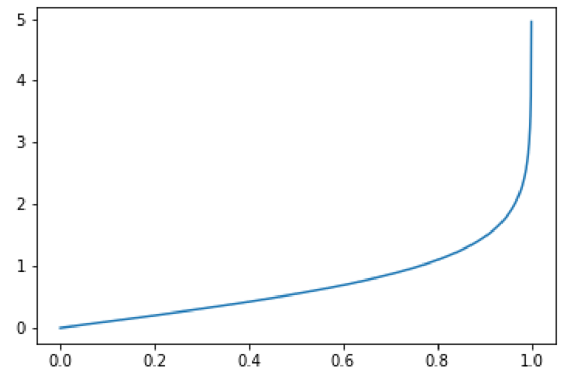
computes poly. approx. of atanh

Precision Preservation: An example

Compute for $0 < x < 1$

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$

$$= (n/2)\log(2) + \frac{1}{2}\log\left(s\frac{1+x}{1-x}\right),\ s = 2^{-n}\ (n \text{ cheap to get})$$



Some implementations:

get $1 \pm x$ and their quotient in dd
feed to special `log` that takes dd inputs
which in turn scales and transform in dd
computes poly. approx. of atanh

VecLibm implementation:

use `vfrec7` to get n

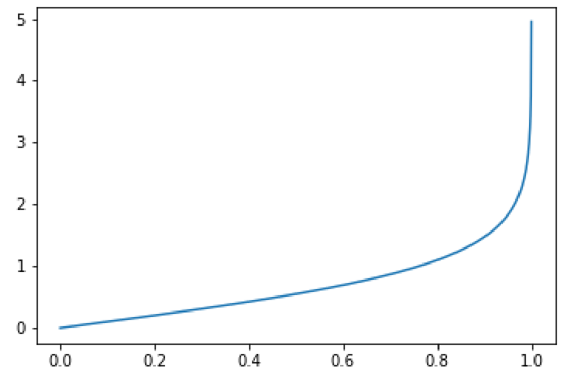
Precision Preservation: An example

Compute for $0 < x < 1$

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$

$$= (n/2)\log(2) + \frac{1}{2}\,\log\left(s\frac{1+x}{1-x}\right),\ s = 2^{-n}\ (n \text{ cheap to get})$$

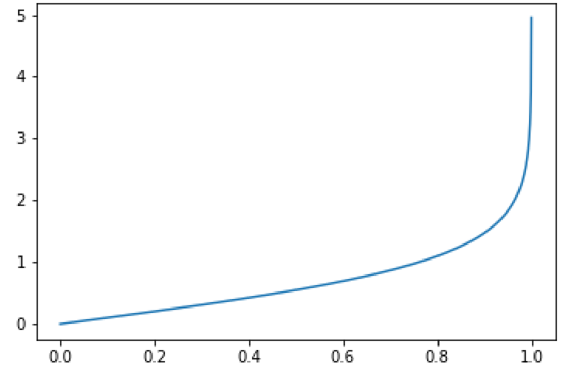
$$= (n/2)\log(2) + \operatorname{atanh}\left(\frac{(1+x)-(1-x)/s}{(1+x)+(1-x)/s}\right)$$



Some implementations:
- get $1 \pm x$ and their quotient in dd
- feed to special `log` that takes dd inputs
- which in turn scales and transform in dd
- computes poly. approx. of atanh

VecLibm implementation:
- use `vfrec7` to get n

Precision Preservation: An example



Compute for $0 < x < 1$

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$

$$= (n/2)\log(2) + \frac{1}{2}\ \log\left(s\frac{1+x}{1-x}\right),\ s = 2^{-n}\ \ (n \text{ cheap to get})$$

$$= (n/2)\log(2) + \operatorname{atanh}\left(\frac{(1+x)-(1-x)/s}{(1+x)+(1-x)/s}\right)$$

Fixed-point scale 60: $(1+X) \pm ((1-X) << n)$ yields exact value

Some implementations:

- get $1 \pm x$ and their quotient in dd
- feed to special `log` that takes dd inputs
- which in turn scales and transform in dd
- computes poly. approx. of atanh

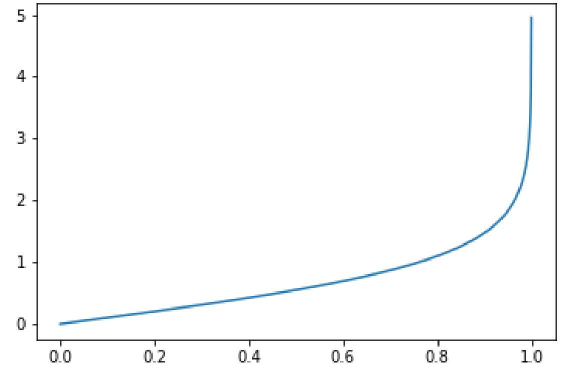
VecLibm implementation:

- use `vfrec7` to get n
- computes numer, denom in fixed point

Precision Preservation: An example

Compute for $0 < x < 1$

$$\operatorname{atanh}(x) = \frac{1}{2}\log\left(\frac{1+x}{1-x}\right) \qquad \log(y) = 2\operatorname{atanh}\left(\frac{y-1}{y+1}\right)$$



$$= (n/2)\log(2) + \frac{1}{2}\log\left(s\frac{1+x}{1-x}\right),\ s = 2^{-n}\ (n \text{ cheap to get})$$

$$= (n/2)\log(2) + \operatorname{atanh}\left(\frac{(1+x)-(1-x)/s}{(1+x)+(1-x)/s}\right)$$

Fixed-point scale 60: $(1 + X) \pm ((1 - X) << n)$ yields exact value

Fixed-point NUMER, DENOM
$\rightarrow$ (a_hi,a_lo), (b_hi,b_lo)

Obtain dd quotient (r, r_lo)

$$(n/2)\log(2) + r + r\_lo + r^3 p(r^2)$$

VecLibm implementation:
- use `vfrec7` to get n
- computes numer, denom in fixed point
- get extra-precise FP input to atanh poly
- use floating-point computation onwards

VecLibm: Current Status At-a-Glance

Library Functions				Maximum Deviation in ulps			
exp	exp2	exp10	expm1	0.56	0.56	0.75	0.77
	when result underflows			0.77	0.77	0.82	N/A
log	log2	log10	log1p	0.55	0.57	0.56	0.66
pow	cbrt			0.55	0.52		
sin	sinpi	cos	cospi	0.79	0.76	0.76	0.77
tan	tanpi			0.62	0.61		
sinh	cosh	tanh		0.67	0.59	0.76	
asin	asinpi	acos	acospi	0.66	0.71	0.64	0.65
atan	atanpi	atan2	atan2pi	0.55	0.55	0.55	0.55
atan2pi underflows				0.75			
asinh	acosh	atanh		0.55	0.56	0.54	

https://github.com/rivosinc/veclibm