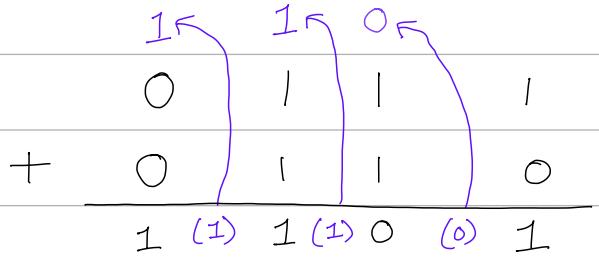
Chapter-3

"Arithmetic for Computers"

Addition: $(7)_{10} + (6)_{10}$



bits with this color represent carry forward.

adding bits right to left.

Subtraction: $(7)_{10} - (6)_{10} = (7)_{10} + (-6)_{10}$

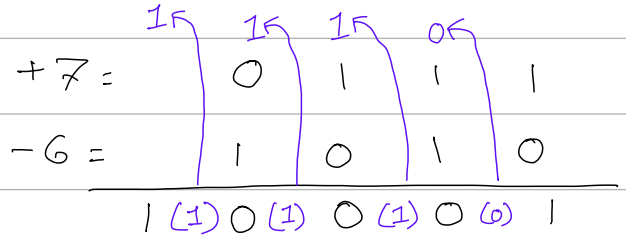
$6 = 110$

$+6 = 0110$

$= 1001$

$+1$

$-6 = (1010)_{2s}$



Overflow (signed)

Addition:

=> Add two same signed number

if (answer also has same sign):

No overflow

else

Overflow

=> Sub two same signed number

Never Over Flow.

=> Add two different signed number

Never Over Flow.

=> Sub two different number

$+A - (-B) = +A + B$ => rule 1

$-A - (+B) = -A + (-B)$ => rule 1

In case of unsigned integers, overflows are ignored.

are used to indicate memory addresses

Compilers detect unsigned overflows using simple branch instruction.

$6 = 110$

$7 = 111$

1101

logic: If the sum is less than either of the addends.

Overflow

if the difference is greater than the minuend.

13 ← Subtrahend

-3

10

ALU = Arithmetic Logic Unit

Long Multiplication

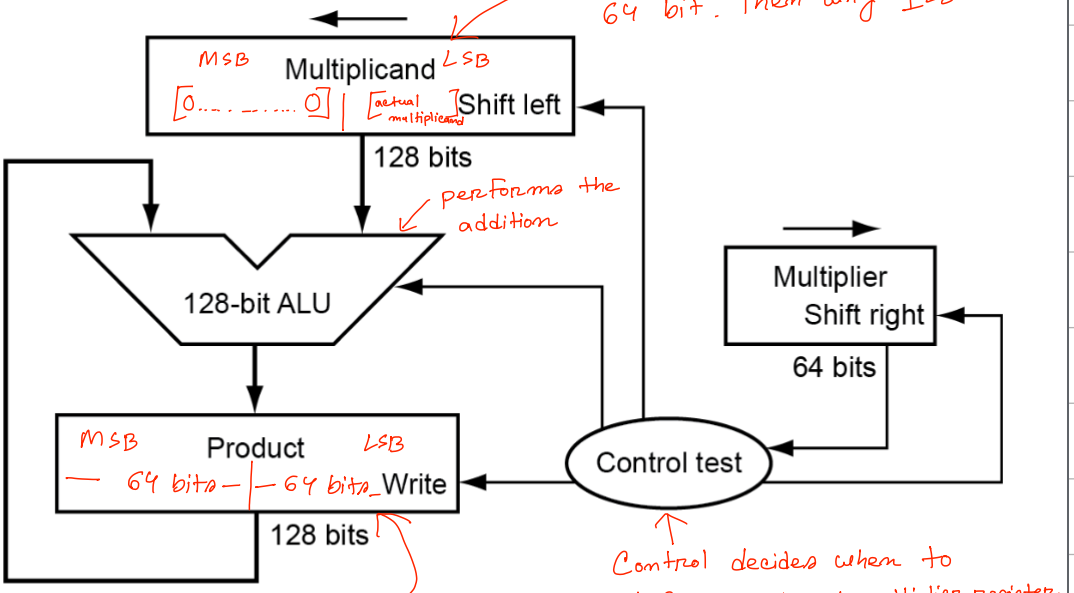
```
  1000   → Multiplicand
x 1001   → Multiplier
------
  1000
 0000
00 00
1000
------
1001000  → Product
```

Maximum,

length of the product

= (length of multiplicand + length of multiplier)

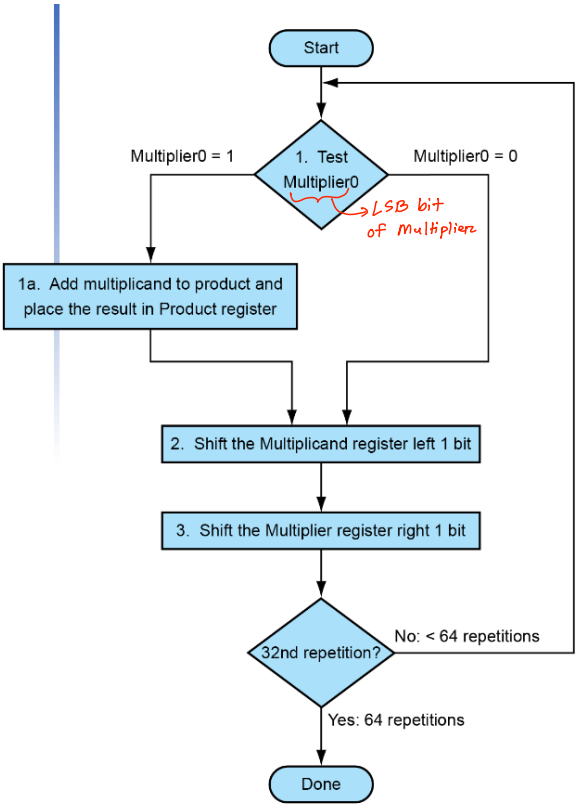
this was also supposed to be 64 bit. Then why 128 bits?

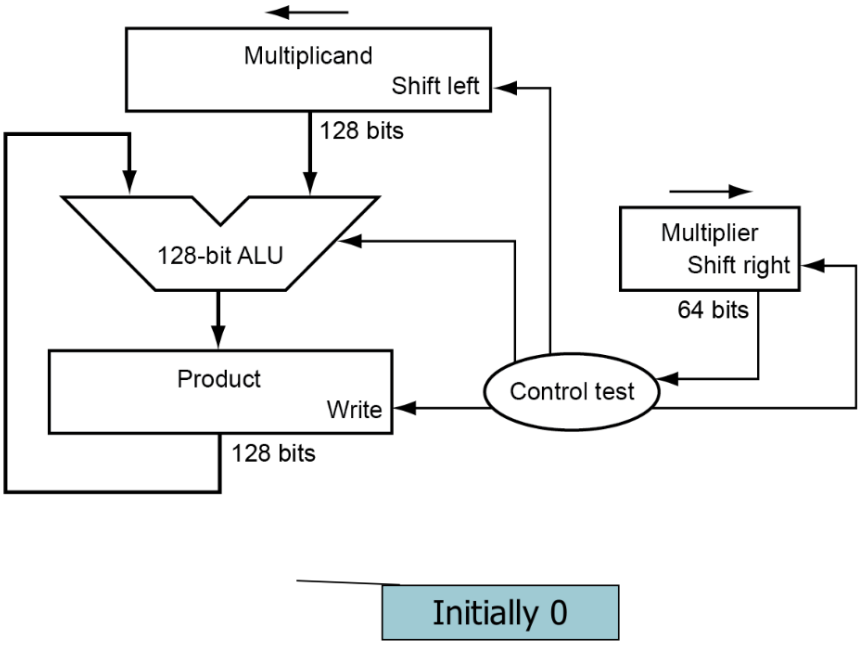


Multiply two 64 bit values, product length can be (64+64)=128 bit.

But we do not have any 128 bit registers in RISC-V

Hence, we use two registers to store the values





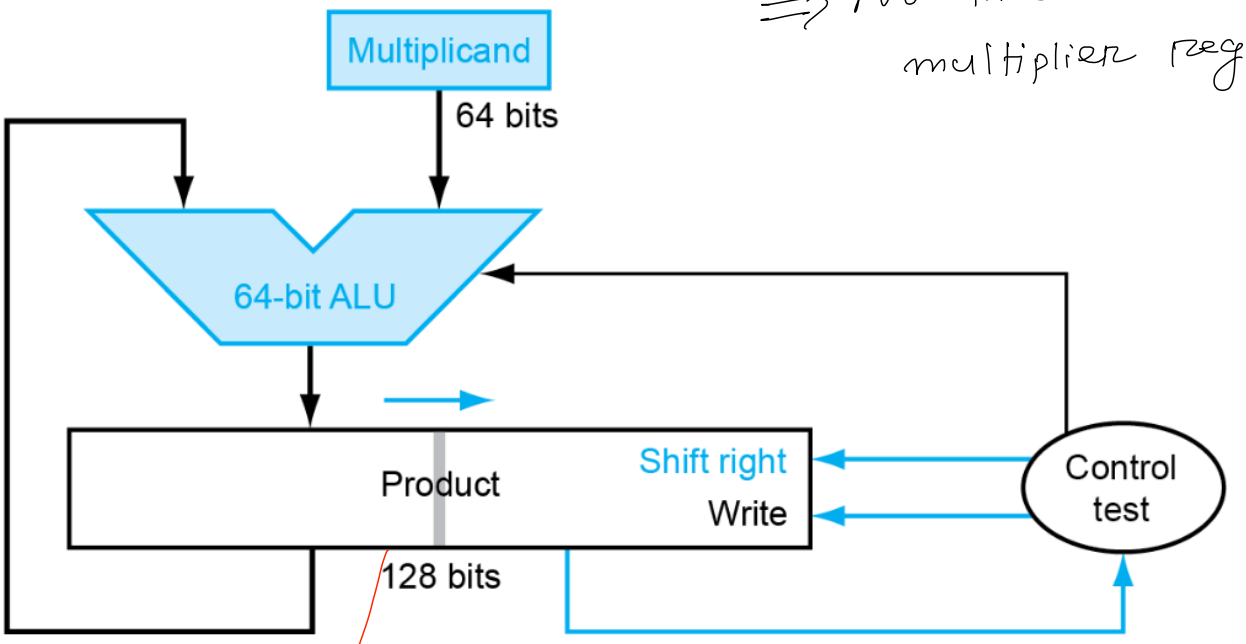
Number of iteration = Number of bits in Multiplier

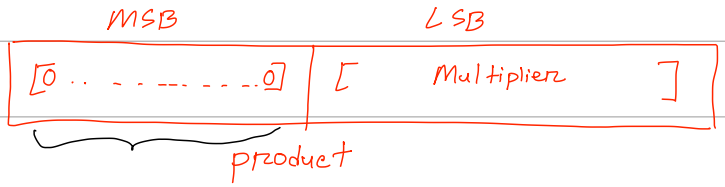
⇒ 1000 × 1001

iteration	Multiplier	Multiplicand	Product
0	1001	0000 1000	0000 0000
	1001	0000 1000	0000 1000
1	1001	0001 0000	0000 1000
	0100	0001 0000	0000 1000
	0100	0001 0000	0000 1000
2	0100	0010 0000	0000 1000
	0010	0010 0000	0000 1000
	0010	0010 0000	0000 1000
3	0010	0100 0000	0000 1000
	0001	0100 0000	0000 1000
	0001	0100 0000	0100 1000
4	0001	1000 0000	0100 1000
	0000	1000 0000	0100 1000

Optimized Multiplication

⇒ No individual multiplier reg.





Number of iteration = Number of bits in Multiplier

logic:

```
if (iteration <= multiplier bit length):
    if (multiplier_0 == 1):
        product_MSB = Multiplicand + product_MSB
        product = right shift product by 1
    elif (multiplier_0 == 0):
        product = right shift product by 1
```

$8 \times 9 = 1000 \times 1001$

iteration	multiplicand	product_MSB	product
0	1000	0000	1001
1	1000	1000	1001
		0100	0100
2	1000	0010	0010
3	1000	0001	0001
4	1000	1001	0001
		0100	1000

Floating Point

How does RISC-V support numbers with ~~fractions~~ ?

Scientific Notation is just a way to represent very large or very small number.

$\Rightarrow 4500000 = 4.5 \times 10^{6}$

Coefficient, Base, Exponent

$\Rightarrow 0.00453 = 4.53 \times 10^{-3}$

Decimal

- $\Rightarrow 5.64 \times 10^{33}$
- $\Rightarrow -2.34 \times 10^{56}$

→ Normalized.

- $\Rightarrow 109.64 \times 10^{33}$
- $\Rightarrow 0.002 \times 10^{-4}$
- $\Rightarrow +987.02 \times 10^{9}$

→ Not Normalized.

In Binary,

$\pm 1.xxxx_2 \times 2^{yyy}$

IEEE-754 → defines floating point standard.

- Single precision. (32-bit)
- Double precision. (64-bit)

In case of double precision, using more bits, you can represent a larger or a smaller number than single precision.

Floating Point

How does RISC-V support numbers with ~~fractions~~ ?

Scientific Notation is just a way to represent very large or very small number.

$\Rightarrow 4500000 = 4.5 \times 10^{6}$

Coefficient, Base, Exponent

$\Rightarrow 0.00453 = 4.53 \times 10^{-3}$

Decimal

- $\Rightarrow 5.64 \times 10^{33}$
- $\Rightarrow -2.34 \times 10^{56}$

→ Normalized.

- $\Rightarrow 109.64 \times 10^{33}$
- $\Rightarrow 0.002 \times 10^{-4}$
- $\Rightarrow +987.02 \times 10^{9}$

→ Not Normalized.

In Binary,

$\pm 1.xxxx_2 \times 2^{yyy}$

IEEE-754 → defines floating point standard.

- Single precision. (32-bit)
- Double precision. (64-bit)

In case of double precision, using more bits, you can represent a larger or a smaller number than single precision.

Normalized Number

Decimal Point (representing Decimal Numbers)

⇒ A decimal number is Normalized if :

i) Only one digit before the decimal point.

ii) And that digit must be a non-zero number.

64.8 ✗ not a normalized number.

6.48×10^{1} ✓ a normalized number.

To normalize a number you need to shift the decimal point (.) left or right until you have a single non-zero digit before the decimal point.

⇒ If you shift left, the number of times you left shifted will be added with the exponent. ⇒ 112.54×10^{35}

⇒ $1.1254 \times 10^{35+2}$ ⇒ 1.1254×10^{37}

⇒ If you shift right, the number of times you right shifted will be subtracted from the exponent. ⇒ 0.0065

⇒ 0.0065×10^{0}

⇒ $6.5 \times 10^{0-3} = 6.5 \times 10^{-3}$

Binary Point (representing Binary Numbers)

⇒ A binary number is Normalized if :

i) Only one digit before the binary point.

ii) And that digit must be a non-zero number.

⇒ 11.00101×2^{35} ✗ not a normalized number.

⇒ 1.100101×2^{37} ✓ a normalized number.

To normalize a number you need to shift the binary point (.) left or right until you have a single non-zero digit before the binary point.

=> If you **shift left**, the **number of times** you left shifted will be **added** with the exponent. => 110.111×2^{35}

$\Rightarrow 1.10111 \times 2^{35+2}$

=> If you **shift right**, the **number of times** you right shifted will be **subtracted** from the exponent. => 0.00110

$\Rightarrow 0.00110 \times 2^{0}$

$\Rightarrow 1.10 \times 2^{0-3} = 1.10 \times 2^{-3}$

IEEE Floating-Point Format

	Sign Bit	Exponent	Fraction
Single P.	1 bit	8 bits	23 bits
Double P.	1 "	11 "	52 "

Single Precision (32 Bit)

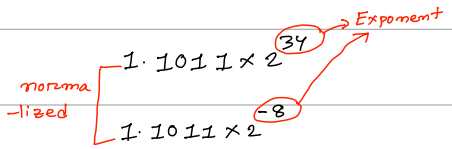
(Biased)

Sign Bit	Exponent	Fraction
1	8	23

Sign Bit = 0 => positive number

1 => negative "

Exponent = It will be represented as unsigned number.



8 bit unsigned binary range = 0 to 2^8-1

= 0 to 255

But 0000 0000 and 1111 1111 are reserved, so the range for biased exponent is 1 to 254

If the size of biased exponent field is n bits, Bias = $2^{(n-1)}-1$

Hence, for 8 bit biased exponent, bias = $2^7-1 = 127$

Biased Exponent = Actual exponent of the binary number + Bias

If you find the actual exponent within (-126 to +127) only then that number can be represented using Single Precision Format.

Ex: 1.1011×2^{34}; Find the Biased Exponent of the given number in IEEE 754 single precision format.

Soln: Check if the number is in normalized format or not.

↳ If no, then normalize it and proceed.

1.1011×2^{34} => normalized number.

Actual Exponent of the binary number

Bias = $2^{(n-1)} - 1 = 2^{8-1} - 1 = 127$.

∴ Biased Exponent = 34 + 127 = 161 = 1010 0001 (Ans)

Ex: 11.1011×2^{-8}; Find the Biased Exponent of the given number in IEEE 754 single precision format.

Soln: 11.1011×2^{-8}

$= 1.11011 \times 2^{-8+1}$

$= 1.11011 \times 2^{-7}$ (-7: actual exponent; } Normalized)

Bias = $2^{(n-1)} - 1 = 2^{8-1} - 1 = 127$.

∴ Biased Exponent = -7 + 127 = 120 = 1011 0100 (Ans)

Decimal to Floating Point Conversion:

(i) Convert the decimal number to binary number.

(ii) Normalize the binary number.

(iii) Find the biased exponent.

(iv) Sign Bit

(v) Find the fraction.

Ex: Convert 50.6749 to 32 bit IEE-754 Floating point representation :-

$50 = 11\ 0010$

$.6749 = 10\ 1011\ 0011\ldots$

(i) $50.6749 = (11\ 0010.10\ 1011\ 0011\ldots)$

$= 11\ 0010.10\ 1011\ 0011\ldots \times 2^0$

(ii) $= 1.1001\ 0101\ 0110\ 011\ldots \times 2^5$ ← actual exponent

(iii) Bias $= 2^{8-1} - 1 = 2^7 - 1 = 127$

$\therefore$ Biased exponent $= 5 + 127 = 132 = 1000\ 0100$

(iv) positive number; sign bit = 0

(v) $1.1001\ 0101\ 0110\ 011\ldots \times 2^5$

fraction

Fraction = 1001 0101 0110 011 00000000

rest of the bits will be filled up by 0s.

$.6749 \times 2 = 1.3498$

$.3498 \times 2 = 0.6996$

$.6996 \times 2 = 1.3992$

$.3992 \times 2 = 0.7984$

$.7984 \times 2 = 1.5968$

$.5968 \times 2 = 1.1936$

$.1936 \times 2 = 0.3872$

$.3872 \times 2 = 0.7744$

$.7744 \times 2 = 1.5488$

$.5488 \times 2 = 1.0976$

(Biased)

Sign Bit	Exponent	Fraction
0	1000 0100	1001 0101 0110 011 00000000

$50.6749 = 0100\ 0010\ 0100\ 1010\ 1011\ 0011\ 0000\ 0000$

$= 0x424AB300$

Double Precision (64 Bit)

Sign Bit	Exponent (Biased)	Fraction
1	11	52

Sign Bit = 0 ⇒ positive number

1 ⇒ negative "

Exponent = It will be represented as unsigned number.

11 bit unsigned binary range = 0 to $2^{11}-1$

= 0 to 2047

But 000 0000 0000 and 111 1111 1111 are reserved, so the range for biased exponent is 1 to 2046

If the size of biased exponent field is n bits, Bias = $2^{(n-1)}-1$

Hence, for 11 bit biased exponent, bias = $2^{10}-1 = 1023$

Convert −0.0232 to 12 bit IEEE-754 floating point representation, where biased component is 4 bits.

Soln:

(i) $-0.0232 = -0.00000\,10$

(ii) $-0.00000\,10 = 1.0 \times 2^{-6}$ ← actual exponent; 1.0 ← fraction

(iii) Bias = $2^{4-1}-1 = 7$

∴ Biased Exponent = −6 + 7 = 1 = 0001

(iv) Sign Bit = 1.

(v) Fraction = 000 0000

1	0001	000 0000

−0.000232 = 1000 1000 0000

= 0x880 (Ans)

Floating Point to Decimal:

(i) Hex to Bimary.

(ii) Arrange the bimary according to the format.

(iii) Determine the sign

(iv) Find out the exponent from biased exponent.

(v) Convert fraction to decimal.

(vi) Decimal Number = $(-1)^{\text{sign bit}} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent})}$

Example: 0xF2400120; convert this single precision floating point number to decimal.

Soln: 0xF2400120

(i) 1111 0010 0100 0000 0000 0001 0010 0000

(ii) 1 111 00100 100 0000 0000 0001 0010 0000

1 — Sign Bit; 111 00100 — Biased Exp.; 100 0000 0000 0001 0010 0000 — Fraction

(iii) Sign = −

(iv) Biased exp. = 111 0010 0 = 228

Bias = $2^{8-1} - 1 = 127$

∴ Exponent = 228 − 127 = 101

(v) Fraction = 100 0000 0000 0001 0010 0000

= 0.100 0000 0000 0001 0010 0000

= 0.5000343323

(vi) Decimal Value = $(-1)^1 \times (1 + 0.5000343323) \times 2^{101}$

$= -1.5000343323 \times 2^{101}$

$= -3.80303884 \times 10^{30}$

Extension: Upto 6 decimal points with rounding = -3.803039×10^{30}

" 6 " " without " = -3.803038×10^{30}

Floating Point Addition / Subtraction

A and B both are floating point number.

⇒ A + B (Make sure the number is in binary)

i) Normalize both A and B.

ii) Align the bin point so that the lower exponent match with the higher exponent.

iii) Now add / sub accordingly.

iv) Normalize the result.

iv) Round if neeessary.

Ex: $0.999 \times 10^{1} + 1.610 \times 10^{-1}$; size of exponent field is 3 bits

$= 99.99 + 0.1610$

$= 11000 11.1111\ 1101\ 01 + 0.0010\ 1001\ 00$

$= 1.1000\ 1111\ 1111\ 0101 \times 2^{6} + 1.0100\ 100 \times 2^{-3}$

$= 1.1000\ 1111\ 1111\ 0101 \times 2^{6} + 0.0000\ 0000\ 1\ 0100\ 100 \times 2^{6}$

$= 1.10010 \times 2^{6}$ (Ans)

Bias $= 2^{3-1} - 1 = 3$

Biased Exp. $= 3 + 6 = 9$

Range = 0 to $2^{3}-1$

= 0 to 7

= 1 to 6 [reserved 0 and 7]

upper range

$9 > 6 \Rightarrow$ So, overflow

$11010.0\ 11011 \times 2^{8} + 1011\ 0.1111\ 1 \times 2^{7}$

$= 1.1010\ 0111\ 011 \times 2^{13} + 1.0110\ 1111\ 1 \times 2^{11}$

$= 1.1010\ 0111\ 011 \times 2^{13} + 0.0101\ 1011\ 111 \times 2^{13}$

$= 10.0000\ 0011\ 010 \times 2^{13}$

$= 1.0000\ 0001\ 1010 \times 2^{14}$ (Ans)

Overflow / Underflow detection.

Given number is too small to represent using the mentioned system.

Step 1: Find the biased exponent of the answer.

Step 2: " " range of the biased exponent of the given system. (1 to upper Range)

Step 3: Detection:

```
if (Biased exponent < 1):
    underflow
else if (Biased exponent > upper Range)
    overflow
else : [1 ≤ Biased Exp ≤ upper Range]
    No over/under flow
```

Floating Point Multiplication

A and B both are floating point number.

$\Rightarrow$ A × B (Make sure the number is in binary)

i) Normalize both A and B.

ii) Add the exponents.

iii) Now multiply accordingly.

iv) Normalize the result.

iv) Round if necessary.

v) Determine the sign from the operation.

Ex: $1.110 \times 2^{5} \times 1.11 \times 2^{-5}$

$= 1.110 \times 1.11 \times 2^{5+(-5)}$

$= 11.0001 \times 2^{0}$

$= 1.10001 \times 2^{1}$ (Ans)

F.P instructions in RiscV

Suppose, two single Preci. floating point numbers A, B are stored in memory. The memory locations are directly stored in register X_{10}, X_{11}. Write necessary code to store the result of A+B in the memory address that is stored in X_{13}.

Soln:

```
flw     f1, 0(X10)   ; f1 = A
flw     f2, 0(X11)   ; f2 = B
fadd.s  f3, f1, f2   ; f3 = A+B
fsw     f3, 0(X13)
```

RISC-V floating-point assembly language

Arithmetic	FP add single	`fadd.s f0, f1, f2`	`f0 = f1 + f2`	FP add (single precision)
	FP subtract single	`fsub.s f0, f1, f2`	`f0 = f1 - f2`	FP subtract (single precision)
	FP multiply single	`fmul.s f0, f1, f2`	`f0 = f1 * f2`	FP multiply (single precision)
	FP divide single	`fdiv.s f0, f1, f2`	`f0 = f1 / f2`	FP divide (single precision)
	FP square root single	`fsqrt.s f0, f1`	`f0 = √f1`	FP square root (single precision)
	FP add double	`fadd.d f0, f1, f2`	`f0 = f1 + f2`	FP add (double precision)
	FP subtract double	`fsub.d f0, f1, f2`	`f0 = f1 - f2`	FP subtract (double precision)
	FP multiply double	`fmul.d f0, f1, f2`	`f0 = f1 * f2`	FP multiply (double precision)
	FP divide double	`fdiv.d f0, f1, f2`	`f0 = f1 / f2`	FP divide (double precision)
	FP square root double	`fsqrt.d f0, f1`	`f0 = √f1`	FP square root (double precision)
Comparison	FP equality single	`feq.s x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (single precision)
	FP less than single	`flt.s x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (single precision)
	FP less than or equals single	`fle.s x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (single precision)
	FP equality double	`feq.d x5, f0, f1`	`x5 = 1 if f0 == f1, else 0`	FP comparison (double precision)
	FP less than double	`flt.d x5, f0, f1`	`x5 = 1 if f0 < f1, else 0`	FP comparison (double precision)
	FP less than or equals double	`fle.d x5, f0, f1`	`x5 = 1 if f0 <= f1, else 0`	FP comparison (double precision)
Data transfer	FP load word	`flw f0, 4(x5)`	`f0 = Memory[x5 + 4]`	Load single-precision from memory
	FP load doubleword	`fld f0, 8(x5)`	`f0 = Memory[x5 + 8]`	Load double-precision from memory
	FP store word	`fsw f0, 4(x5)`	`Memory[x5 + 4] = f0`	Store single-precision from memory
	FP store doubleword	`fsd f0, 8(x5)`	`Memory[x5 + 8] = f0`	Store double-precision from memory