

Chapter-3

"Arithmetic for Computers"

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

$$\begin{array}{r} & 1 \swarrow & 1 \swarrow & 0 \swarrow \\ 0 & | & | & | \\ + & 0 & 1 & 1 \\ \hline 1 & (1) & 1 & (1) 0 & (0) 1 \end{array} \quad \left. \begin{array}{l} \text{\# bits with this color represent} \\ \text{carry forward.} \end{array} \right\}$$

adding bits right to left.

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

$$\begin{array}{r} 6 = 110 \\ + 6 = 0110 \\ = 1001 \\ + 1 \\ \hline - 6 = \underline{(1010)}_{10} \end{array}$$

$$\begin{array}{r} + 7 = 1 \swarrow 0 \swarrow 1 \swarrow 1 \swarrow 0 \swarrow \\ 0 \quad | \quad 1 \quad | \quad 1 \quad | \quad 1 \quad | \\ - 6 = \hline 1 & (1) & 0 & (1) & 0 & (1) & 0 & (0) & 1 \end{array}$$

Overflow (signed)

Addition:

\Rightarrow Add two same signed numbers

if (answer also has same sign):

No overflow

else

Overflow

\Rightarrow Sub two same signed number

Never Over Flow.

\Rightarrow Add two different signed number

Never Over Flow.

\Rightarrow Sub two different number

$$+A - (-B) = +A + B \Rightarrow \text{rule 1}$$

$$-A - (+B) = -A + (-B) \Rightarrow \text{rule 1}$$

In case of unsigned integers, overflows are ignored.

are used to indicate memory addresses

Compilers detect unsigned overflows using simple branch instruction.

$$\begin{array}{r} 6 = \boxed{110} \\ 7 = \boxed{111} \\ \hline 1101 \end{array}$$

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

Overflow

if the difference is greater

than the minuend.

$$\begin{array}{r} 13 \\ - 3 \\ \hline 10 \end{array}$$

ALU = Arithmetic Logic Unit

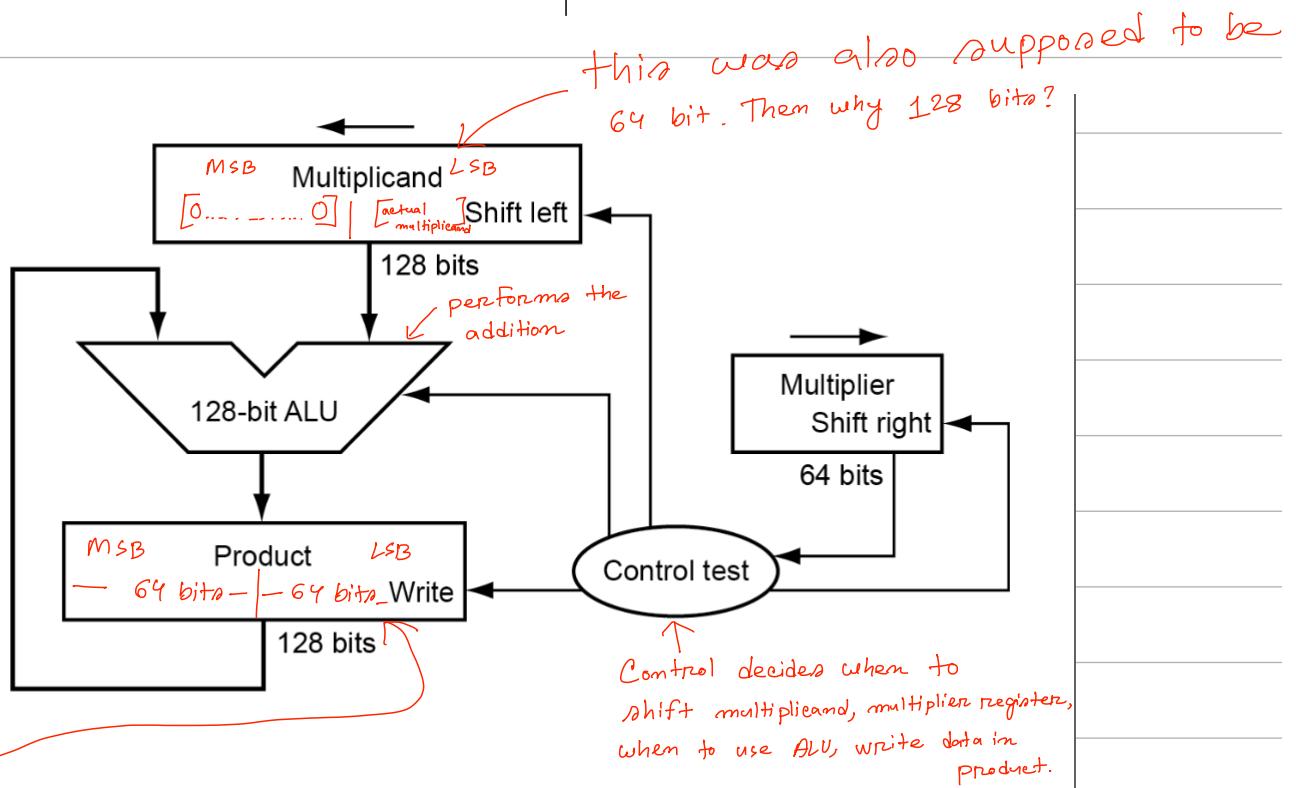
Long Multiplication

$$\begin{array}{r}
 1000 \rightarrow \text{Multiplicand} \\
 \times 1001 \rightarrow \text{Multiplier} \\
 \hline
 1000 \\
 0000 \\
 0000 \\
 1000 \\
 \hline
 1001000 \rightarrow \text{Product}
 \end{array}$$

Maximum,

length of the product

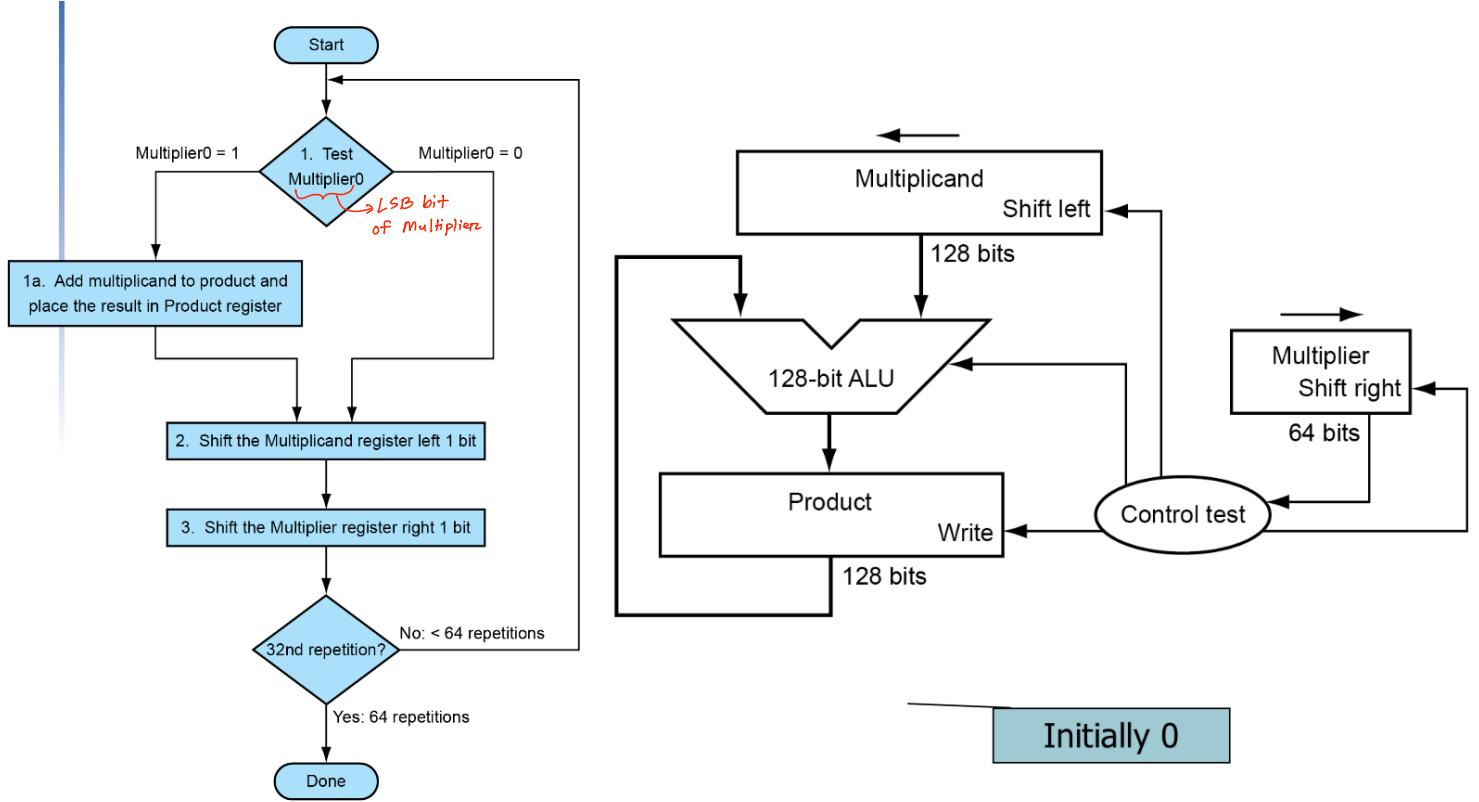
$$\begin{aligned}
 &= (\text{length of multiplicand} \\
 &+ \text{length of multiplier})
 \end{aligned}$$



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 64 bit values

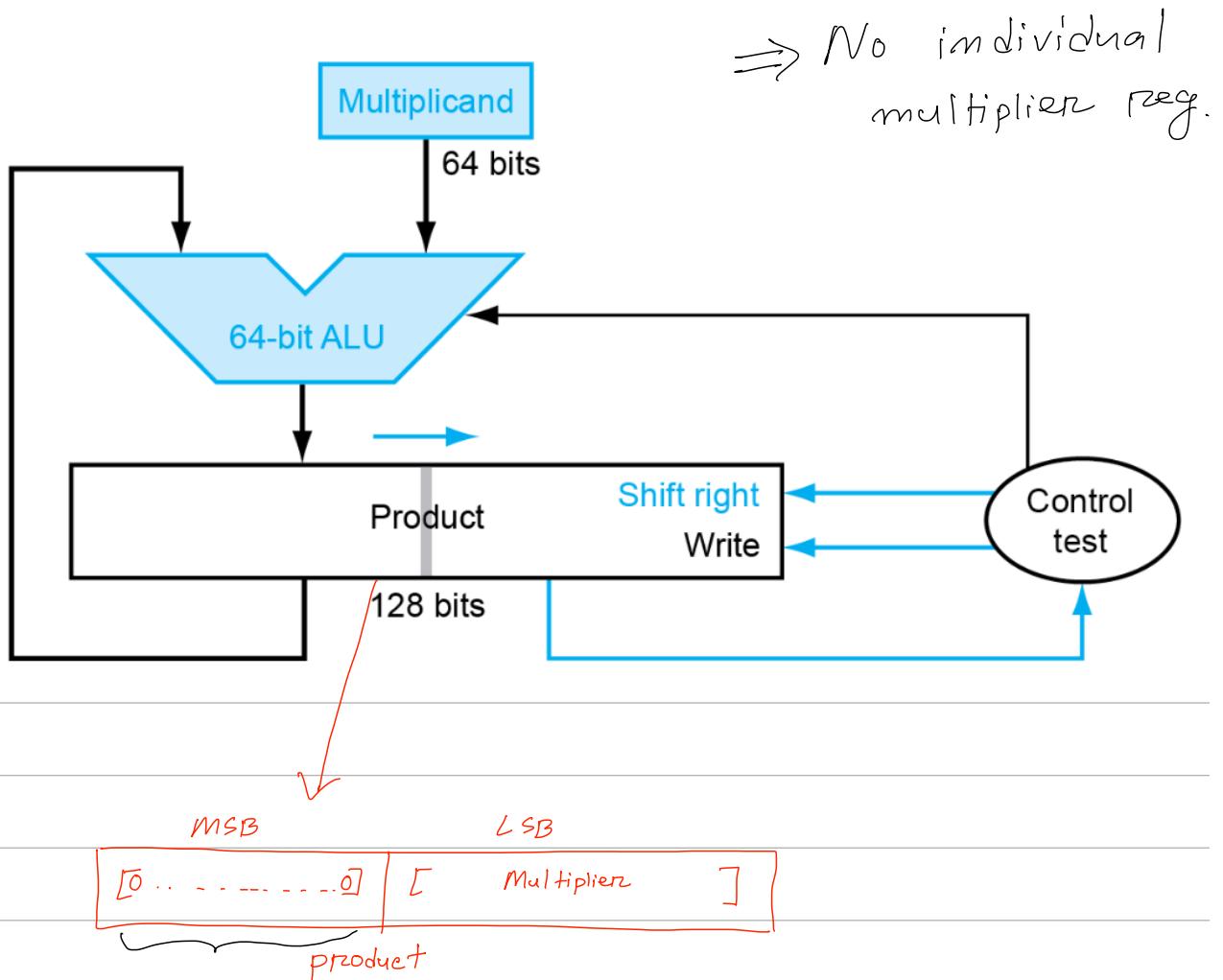


Number of iteration = Number of bits in **Multiplier**

$\Rightarrow 100 \times 100$

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

Optimized Multiplication



Number of iteration = Number of bits in **Multiplier**

Logic:

if (iteration <= multiplier bit length):

 if (multiplier0 == 1):

 product_MSB = Multiplicand + product_MSB

 product = right shift product by 1

 elif (multiplier0 == 0):

 product = right shift product by 1

$$8 \times 9 = 1000 \times 1001$$

iteration	multiplier	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 numbers.

$$\Rightarrow 4500000 = \underbrace{4.5}_{\text{Coefficient}} \times \underbrace{10^6}_{\text{Base}} \quad \underbrace{6}_{\text{Exponent}}$$
$$\Rightarrow 0.00453 = \underbrace{4.53}_{\text{Coefficient}} \times \underbrace{10^{-3}}_{\text{Base}}$$

Decimal

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

Normalized.

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

Not Normalized.

$$\Rightarrow +087.02 \times 10^9$$

In Binary,

$$\pm 1. \times \times \times_2 \times 2^{888}$$

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 numbers.

$$\Rightarrow 4500000 = \underbrace{4.5}_{\text{Coefficient}} \times \underbrace{10^6}_{\text{Base}} \quad \underbrace{6}_{\text{Exponent}}$$
$$\Rightarrow 0.00453 = \underbrace{4.53}_{\text{Coefficient}} \times \underbrace{10^{-3}}_{\text{Base}}$$

Decimal

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

Normalized.

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

Not Normalized.

$$\Rightarrow +087.02 \times 10^9$$

In Binary,

$$\pm 1. \times \times \times_2 \times 2^{888}$$

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 :

- Only one digit before the decimal point.
- And that digit must be a non-zero number.

64.8×10^0 ✗ 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.

$$\Rightarrow 112.54 \times 10^{35}$$

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

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

$$\Rightarrow 0.0065$$

$$\Rightarrow 0.0065 \times 10^0$$

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

Binary Point (representing Binary Numbers)

⇒ A binary number is Normalized if :

- Only one digit before the binary point.
- And that digit must be a non-zero number.

$11.00101 \times 2^{35} \times 10^0$ ✗ not a normalized number.

$1.100101 \times 2^{37} \times 10^0$ ✓ 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.

$$\Rightarrow 1.10.111 \times 2^{35}$$

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

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

$$\Rightarrow 0.00110$$

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

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

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)

Sign Bit	Exponent	Fraction
1	8	23

Sign Bit = 0 ⇒ positive number

1 ⇒ negative

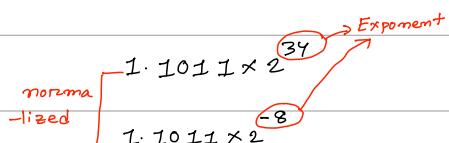
Exponent = It will be represented as unsigned number.

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 IEE 754 single precision format.

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

→ If no. then normalize it and proceed.

$1.1011 \times 2^{34} \Rightarrow$ normalized number.

Actual Exponent of
the binary number

$$\text{Bias} = 2^{(m-1)} - 1 = 2^{8-1} - 1 = 127. \quad | \quad \therefore \text{Biased Exponent} = 34 + 127 = 161 = 1010\ 0001 \quad (\text{Ans})$$

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

Solⁿ: 11.1011×2^{-8}

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

actual exponent

$$= 1.11011 \times 2^{-7} \quad \left. \begin{array}{l} \\ \end{array} \right\} \text{Normalized}$$

$$\text{Bias} = 2^{(m-1)} - 1 = 2^{8-1} - 1 = 127. \quad | \quad \therefore \text{Biased Exponent} = -7 + 127 = 120 = 1011\ 0100 \quad (\text{Ans})$$

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 IEEE-754 Floating point representation :-

$$50 = 11\ 0010$$

$$\cdot 6749 \times 2 = \underline{1.3498}$$

$$\cdot 6749 = 10\ 1011\ 0011\dots$$

$$\cdot 3498 \times 2 = \underline{0.6006}$$

(i) $50.6749 = (11\ 0010 \cdot 10\ 1011\ 0011\dots)$

$$\cdot 6006 \times 2 = \underline{1.3002}$$

$$= 11\ 0010 \cdot 10\ 1011\ 0011\dots \times 2^0$$

$$\cdot 3002 \times 2 = \underline{0.7084}$$

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

$$\cdot 7084 \times 2 = \underline{1.5068}$$

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

$$\cdot 5068 \times 2 = \underline{1.1036}$$

$$\therefore \text{Biased exponent} = 5 + 127 = 132 = \underline{1000\ 0100}$$

$$\cdot 1036 \times 2 = \underline{0.3872}$$

(iv) Positive number; sign bit = 0

$$\cdot 3872 \times 2 = \underline{0.7744}$$

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

$$\cdot 7744 \times 2 = \underline{1.5488}$$

Fraction

$$\cdot 5488 \times 2 = \underline{1.0976}$$

Fraction = 1001 0101 0110 011 00000000
rest of the bits
will be filled up by 0s.

(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)

(Biased)		
Sign Bit	Exponent	Fraction
1	11	52

Sign Bit = 0 \Rightarrow positive number

1 \Rightarrow negative

Exponent = It will be represented as

unsigned number.

$$11 \text{ bit unsigned binary range} = 0 \text{ to } 2^{11}-1 \\ = 0 \text{ to } 2047$$

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 m bits, Bias = $2^{m-1}-1$

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.

Sol^{n:}

$$(i) -0.0232 = -0.0000010$$

$$(ii) -0.0000010 = 1.0 \times 2^{-6}$$

actual exponent

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

$$\therefore \text{Biased Exponent} = -6 + 7 = 1 = 0001$$

(iv) Sign Bit = 1.

(v) Fraction = 000 0000

1	0001	000 0000
---	------	----------

$$-0.000232 = 1000 1000 0000$$

$$= 0x880 \text{ (Ans)}$$

Floating Point to Decimal:

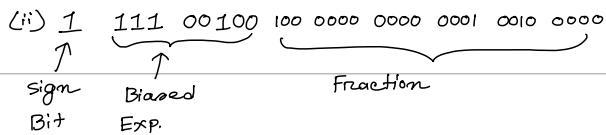
- (i) Hex to Binary.
- (ii) Arrange the binary according to the format.
- (iii) Determine the sign
- (iv) Find out the exponent from biased exponent.
- (v) Convert fraction to decimal.

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

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

Solⁿ: 0xF2400120

(i) 1111 0010 0100 0000 0000 0001 0010 0000

(ii) 
↑ ↑ ↓
Sign Biased Fraction
Bit Exp.

(iii) sign = -

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

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

$$\therefore \text{Exponent} = 228 - 127 = 101$$

(v) Fraction = 100 0000 0000 0001 0010 0000

$$= 0.100 0000 0000 0001 0010 0000$$

$$= 0.5000343323$$

$$(vi) \text{ Decimal Value} = (-1)^1 \times (1 + 0.5000343323) \times 2^{101}$$
$$= -1.5000343323 \times 2^{101}$$

$$= -3.80303889 \times 10^{30}$$

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

$$\text{u} \quad 6 \quad \text{u} \quad \text{u} \quad \text{u} \quad \text{without} \quad \text{u} \quad = -3.803038 \times 10^{30}$$

Floating Point Addition / Subtraction

A and B both are floating point numbers.

$\Rightarrow A + B$ (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.

v) Round if necessary.

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

$$= 99.99 + 0.1610$$

$$= 1100011 \cdot 111110101 + 0.0010100100$$

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

$$= 1.10001111110101 \times 2^6 + 1.0100100 \times 2^{-3}$$

$$\text{Biased Exp.} = 3 + 6 = 9$$

$$= 1.10001111110101 \times 2^6 + 0.00000000010100100 \times 2^{-6}$$

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

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

$$= 1 \text{ to } 6 \quad [\text{reserved } 1 \text{ and } 7]$$

upper range

$110100 \cdot 111011 \times 2^8 + 10110 \cdot 11111 \times 2^7$

$$= 1.10100111011 \times 2^{13} + 1.0110111111 \times 2^{11}$$

$$9 > 6 \Rightarrow \text{So,}$$

overflow

$$= 10.00000011010 \times 2^{13}$$

$$= 1.000000011010 \times 2^{14} \quad (\text{Ans})$$

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 : $[1 \leq \text{Biased Exp} \leq \text{upper Range}]$

No over/under flow

Floating Point Multiplication

A and B both are floating point numbers.

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

i) Normalize both A and B. Ex: $1.110 \times 2^5 \times 1.11 \times 2^{-5}$

ii) Add the exponents. $= 1.110 \times 1.11 \times 2^{5+(-5)}$

iii) Now multiply accordingly. $= 11.0001 \times 2^0$

iv) Normalize the result. $= 1.10001 \times 2^1$ (Ans)

v) Round if necessary.

v) Determine the sign from the operation.

F.P instructions in RiscV

Suppose, two single prec. floating point numbers A, B are stored in memory. The memory locations are directly stored in registers 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:

f1w f₁, 0(x_{10}) ; f₁ = A

f1w f₂, 0(x_{11}) ; f₂ = B

fadd.s f₃, f₁, f₂ ; f₃ = A + B

fsw f₃, 0(x_{13})

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	$f0 = \sqrt{f1}$	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	$f0 = \sqrt{f1}$	FP square root (double precision)
Comparison	FP equality single	feq.s x5, f0, f1	$x5 = 1 \text{ if } f0 == f1, \text{ else } 0$	FP comparison (single precision)
	FP less than single	flt.s x5, f0, f1	$x5 = 1 \text{ if } f0 < f1, \text{ else } 0$	FP comparison (single precision)
	FP less than or equals single	fle.s x5, f0, f1	$x5 = 1 \text{ if } f0 \leq f1, \text{ else } 0$	FP comparison (single precision)
	FP equality double	feq.d x5, f0, f1	$x5 = 1 \text{ if } f0 == f1, \text{ else } 0$	FP comparison (double precision)
	FP less than double	flt.d x5, f0, f1	$x5 = 1 \text{ if } f0 < f1, \text{ else } 0$	FP comparison (double precision)
	FP less than or equals double	fle.d x5, f0, f1	$x5 = 1 \text{ if } f0 \leq f1, \text{ else } 0$	FP comparison (double precision)
Data transfer	FP load word	f1w f0, 4(x5)	$f0 = \text{Memory}[x5 + 4]$	Load single-precision from memory
	FP load doubleword	f1d f0, 8(x5)	$f0 = \text{Memory}[x5 + 8]$	Load double-precision from memory
	FP store word	fsw f0, 4(x5)	$\text{Memory}[x5 + 4] = f0$	Store single-precision to memory
	FP store doubleword	fsd f0, 8(x5)	$\text{Memory}[x5 + 8] = f0$	Store double-precision to memory