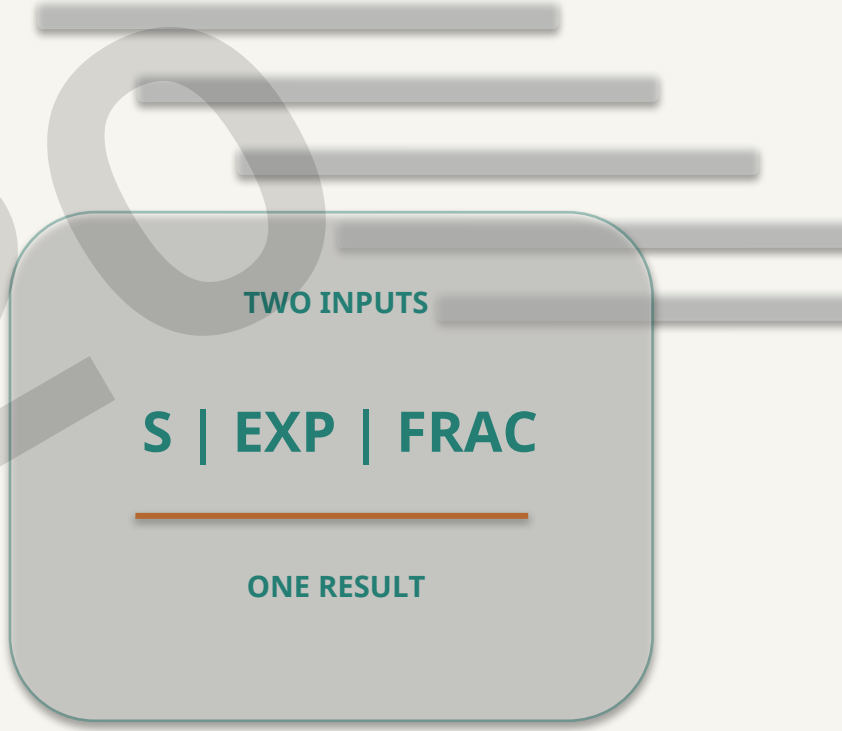
Building a 32-bit Floating-Point Multiplier



Our goal: multiply two floating-point values and pack the answer back into 32 bits.

24 × 24 MANTISSA

48-BIT PRODUCT

RNE ROUNDING

Made by Sidda, Neeraf, Kanav, Srijan and Md. Sahil

Group-20

So, what happens after A and B arrive?

We split the job into smaller modules, then let the top module choose the final 32-bit answer.

01 INPUT PROCESSING
Sidda

02 MANTISSA MULTIPLIER
Neeraf

03 NORMALIZE + ROUND
Kanav

04 SIGN + EXPONENT
Srijan

05 RESULT SELECT
Top module

THE NUMBER-CRUNCHING PART

Multiply the mantissas, round the fraction, and work out the new exponent.

THE EXCEPTION CHECKS

NaN, Infinity, zero and range errors can override the ordinary arithmetic result.

The blocks do not all wait in one long chain: the exponent and mantissa paths work from the unpacked inputs, and the top module brings their results together.

Group 20

A 32-bit float is three fields packed together

Before multiplying anything, we need to know what each group of bits represents.

Sign: 0 = positive
1 = negative

Exponent: stored with bias 127
(unbiased exponent = E − 127)

Fraction: precision bits after the
implicit leading bit for normal values

NORMAL VALUE

mantissa = 1.fraction

→ 24-bit internal value

SUBNORMAL VALUE

mantissa = 0.fraction

→ no hidden 1

Exponent 255 marks Infinity or NaN. Exponent 0 is used for zero and subnormal numbers.

group-20

make the inputs usable

Ispliteach32-bitinputintosign,exponentandfraction,buildtheinternalmantissa,and pass along a zero flag.

A AND B

[31] sign
[30:23] exp
[22:0] fraction

TAKE OUT THE FIELDS

sign_a = a[31]
sign_b = b[31]
exp_a / exp_b = [30:23]
fraction = [22:0]

BUILD 24-BIT MANTISSA

exp = 0
→ {0, fraction}

exp =L0
→ {1, fraction}

CHECK FOR ZERO

Either input is

+0 or −0

WHY I KEPTTHISSEPARATE

Other modules receive sign_a, exp_a and mant_a directly. Only normal inputs get the hidden 1.

Group 20

Small checks here affect the whole calculation

A

FIELDS I PASS ON

Sign is 1 bit

Exponent is 8 bits
Mantissa is 24 bits internally

B

WHEN THE LEADING 1 EXISTS

exp = 0 → add a 0

exp =E0 → add a 1
This keeps subnormal inputs distinct

C

WHY THE SIGN IS IGNORED

Check a[30:0] and b[30:0]
The sign bit is left out, so +0 and −0 both count as zero

D

WHERE NaN / INFINITY ARE CHECKED

This module flags zero. The top module checks exp = 255 and the fraction to tell NaN from Infinity.

Group-20

Neeraf’s part: build the 48-bit product

For each bit of mant_b, the code makes a shifted row. Those rows are added to get the full product.

mant_a × mant_b
24 bits × 24 bits

24 shifted rows
(each 48 bits wide)

Add one row at a time
partial_sum[i] = previous + pp[i]

WHAT THE SHIFTED ROWS LOOK LIKE

How the adder is built
The 48-bit adder is made from twelve 4-bit CLA blocks. The carry still passes from block to block, but each 4-bit block calculates its internal carries using lookahead logic.

One trade-off: the code is easy to follow, but 23 additions in a row can make the combinational path long.

Group-20

Kanav’s part: line up the product and round it

Bit47tellsuswhich24-bitwindowtokeep.Guard,roundandstickybitsthendecidewhethertoaddonebeforedropping the extra bits.

48-BIT PRODUCT
mant_prod[47:0]

CHECK BIT 47
bit47 = 1?

CHOOSE 24 BITS
raw_mant

ROUNDING BITS
round_up

FINAL FRACTION
mant_final

ROUND TO NEAREST, TIES TO EVEN

round_up = G & (R | S | raw_mant[0])

The retained least-significant bit breaks exact ties toward even.

TELL THE EXPONENT BLOCK

If the product starts with bit 47 set, or rounding carries out, norm_shift_flag tells the exponent block to increase the exponent.

Group-20

Srijan’s part: get the sign and exponent right

ThesignistheXORoftheinputsigns.Theexponentblockaddsthestoredexponents,removesonebias,appliesthe normalization adjustment, and checks the range.

sign_out = sign_a XOR sign_b

Eresult = Ea + Eb − 127 + norm_shift_flag

1 **Add the exponents**

Ea + Eb

2 **Subtract the bias**

−127

3 **Adjust if needed**

+ norm_shift_flag

4 **Check the result range**

underflow / overflow / final_exp

In this design: overflow becomes Infinity; underflow becomes zero; otherwise, use final_exp.

Group-20

What if one of the inputs is a special value?

The top module checks these conditions from top to bottom. The first matching branch decides result.

1 **If either input is NaN**

Return the canonical quiet NaN

2 **Infinitymultipliedby zero**

This case returns NaN

3 **Infinity,unlesszero was involved**

Return signed Infinity

4 **Zero input or underflow**

Return zero with the calculated sign

5 **Overflow**

Signed infinity

6 **No special case matched**

Join sign, exponent and fraction

Group-20

How we check whether it works

Sahil’stestbenchtriestenselectedcases,waitsforthecombinationallogictosettle, and compares the result bits with the expected pattern.

CASES WE TRY

1 Zero × Infinity → NaN
2 Zero × finite → Zero
3 Infinity × finite → Infinity
4 Infinity × Infinity → Infinity
5 1.5 × 1.5 → 2.25
6 2.0 × 3.0 → 6.0
7 −2.5 × 4.0 → −10.0
8 RNE rounding increment
9 NaN propagation
10 Exponent underflow

HOW A TEST PASSES

check_result() waits #1, compares the 32-bit values, and updates the pass count. For NaN, it accepts any result with an all-ones exponent and a nonzero fraction.

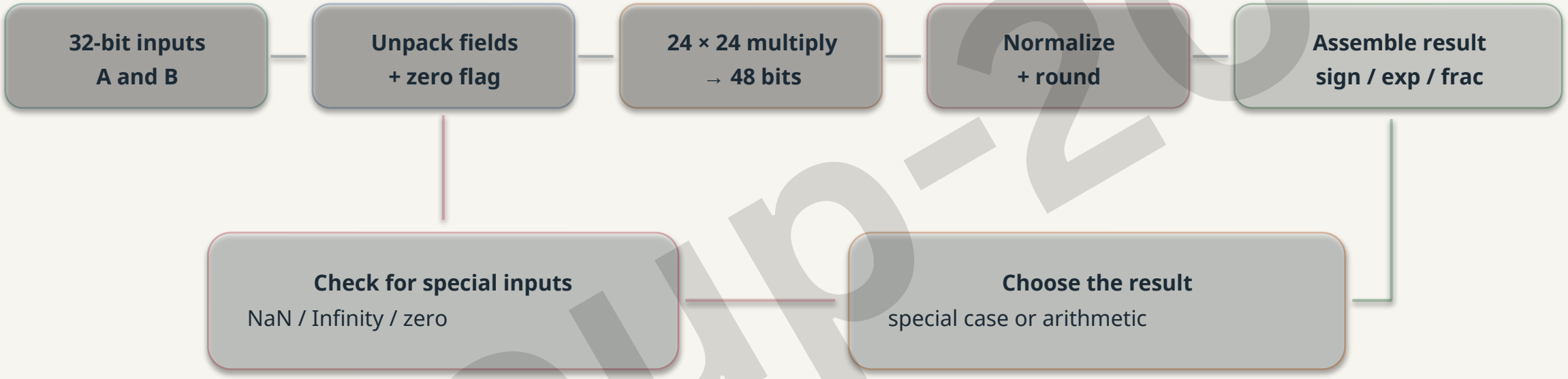
LOOK INSIDE WITH A WAVEFORM

The testbench writes dump.vcd. Opening it in a waveform viewer lets us follow signals and see where a result came from.

Group-20

Putting the whole design together

Themantissaandexponentpathscalculatetheirparts,whilethetopmodulehandlesspecial cases and assembles the final word.



At the end: a 32-bit output, a test summary, and a VCD file we can inspect when something goes wrong.

Group-20

What is working—and what still needs attention

The current code gives us a modular starting point, but it is worth being precise about its limits.

WHAT IS IN PLACE

- Each major job has its own module
- The same CLA structure is reused in several places
- Special results are handled in a clear priority order
- The testbench covers normal and special cases

CURRENT LIMITS

- The partial products are added one after another
- Subnormal results are not fully supported; underflow is flushed to zero
- Only selected test cases are included so far
- No randomized comparison against a reference is included

WHAT WE WOULD TRY NEXT

- Support gradual underflow if full IEEE-754 behavior is required
- Test negative zero and swap A/B in test cases
- Compare many random inputs with a trusted software result
- Check hardware area and timing after synthesis

Group 20

Who worked on which block?

Thissplitfollowstheownershipcommentsandmodulenamesinoursource files.

Sidda	**input_processing**	Splits the input fields, builds 24-bit mantissas and flags zero operands.
Neeraf	**mantissa_multiplier**	Makes the shifted partial-product rows and adds them into a 48-bit product.
Kanav	**normalization_unit& rounding**	Chooses the product bits to keep and applies guard/round/sticky rounding.
Srijan	**sign_exponent_unit**	Works out the result sign and exponent, then flags overflow or underflow.
Md. Sahil	**tb_fp_multiplier**	Runs ten selected tests, checks expected outputs, and creates the VCD waveform.

Group-20