**Floating-point numbers and implementing a floating-point multiplier**

Submitted in partial fulfilment of the requirements of

**EEE F266**

By

**Shreyas T. Bindumadhavan**

**ID Number. 2020B3A30837H**

Under the supervision of

**Prof. S. Gurunarayanan**

**BITS Pilani, Hyderabad Campus**

**Contents**

[**Acknowledgements** 3](#_Toc165637417)

[**Abstract** 4](#_Toc165637418)

[**Introduction** 5](#_Toc165637419)

[**The IEE 754 Standard: An Introduction** 5](#_Toc165637420)

[**Hardware Implementation of a Floating-Point Multiplier** 12](#_Toc165637421)

[**Conclusion** 16](#_Toc165637422)

[**Appendix: Verilog modules** 17](#_Toc165637423)

[**References** 19](#_Toc165637424)

# **Acknowledgements**

I would like to thank Prof. Gurunarayanan for all of his guidance and feedback throughout the entirety of this project which has proved invaluable and has advanced my learning in this field immnesely.

I would also like to thank my parents for their constant support and for always pushing me to be a better version of myself.

# **Abstract**

We introduce floating point numbers and their IEEE 754 standard representation along with the special representations used to encode zero, infinity and NaNs. We then demonstrate the multiplication algorithm used for floating point numbers. Finally, we implement a pipelined multiplier on hardware using a Booth radix-4 modified multiplier to generate partial products which are then passed through a compression tree and rounded before the final product is passed out.

# **Introduction**

High speed floating-point operations are required to meet the computation demands of processors today. It is estimated that nearly 45% of all floating-point operations involve multiplication [4]. Thus, there has been intensive research focused on the design of efficient floating-point multiplication over several years. This paper is a first attempt at replicating these highly-optimised floating-point units in order to obtain a deeper perspective into the design choices employed while implementing these units on commercial processors.

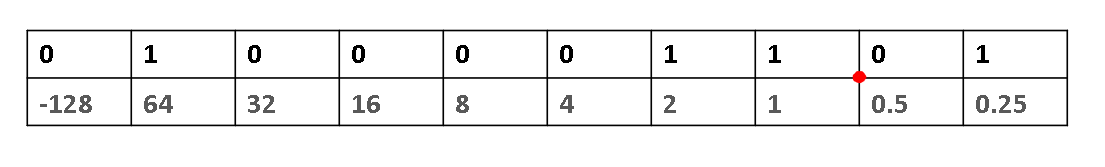
# **The IEE 754 Standard: An Introduction**

**Converting fractional decimals to binary**

Similar to how converting integer decimals to binary involves repeated division by 2 and reading the remainder bottom to up, converting the fractional part of a decimal involves repeated multiplication by 2 until we obtain a product 1.0, then reading the integer part (either 0 or 1) from top to bottom.

**Floating-point representations**

A naïve representation of fractional binary is fixed-point notation (shown below using a two’s complement representation).



*Fixed-point representation of floating-point numbers*

The drawback to this approach is that we are only capable of representing a *limited* range of numbers. This is not feasible when we have a limited amount of memory space to encode a number.

In a *floating-*point representation, we allow the radix point to *float*, allowing for a wider range of representation. This is similar to using scientific notation in base 10 representations to represent larger numbers. Just as in scientific notation (base 10), the exponent will be an appropriate power of 2.

In general, floating-point numbers can be represented in base-2 as *,* where *s* represents the sign of the number (0 is positive, 1 is negative), *m* is the *significand* (also called the *mantissa*) and *e* is the *signed exponent* of the floating-point number

**IEE 754 Standard**

The IEEE 754 standard for representing floating point uses the following rules to encode floating point numbers is shown below and uses the rules that follow to represent floating-point numbers.

|  |  |  |
| --- | --- | --- |
| Sign | Biased exponent | Normalised mantissa |

*The IEEE 754 floating point format*

**i) The *sign* bit**

This is similar to the sign bit used in twos’ complement representations: 1 represents a negative number, while 0 represents a positive number.

**ii) The *biased* exponent**

For a single precision (32-bit) format, the biased exponent is 8 bits long. This means that we have a range of exponent values from 00000000 (0) to 11111111 (255), i.e., 256 exponent values. The first and last values are reservedto represent special cases.

If we were to use these exponent values as is, we would only be left with positive exponents and would be unable to represent numbers with negative exponents. To solve this, we subtract a bias from every exponent calculated using the formula below.

For single precision, the bias is equal to 127 (=254/2). Subtracting the bias from the exponent value to obtain the biased exponent, we have a range of values from [-127,128], which covers both positive and negative exponent values.

**iii) The *normalised*** **significand**

All binary numbers (excluding zeros, subnormals, infinities and NaNs) can be represented as *.* We choose to omit the leading 1 in the mantissa since it is implied, giving us an extra bit of precision.

**Special cases: Zeroes, Infinities and NaNs**

We reserve the first and last exponent encodings (00000000 and 11111111) to handle four exceptional cases arising during computation.

**Zero:** Since the significand is in normalised, we reserve the encoding *e* = 8’d0*,* and *m =* 23’d0 to represent zero.

|  |  |  |
| --- | --- | --- |
| X | 8’b00000 | 23’b0000000000 |

*Zero*

This representation accounts for a *signed* zero – this is ignored when comparisons are to be performed, but is useful while performing multiplication and division on certain edge cases and dealing with underflow for functions that have a discontinuity at zero.

**Subnormal numbers**: Subnormal numbers are represented as *e* = 8’d0, and the significand is unconstrained (do not care).

Subnormal numbers are represented as

|  |  |  |
| --- | --- | --- |
| X | 8’b00000000 | X |

*Subnormals*

For a subnormal number, at least one bit of the significand must be set to 1.

= for the half-precision format

Notice the exponent is (1-bias) and not (0-bias) – the exponent is increasedby 1. This fills in the values between the two smallest values on either side of zero (that is, and provides *gradual underflow* behaviour for computations that result in small values that would otherwise underflow straight to zero.

**Infinity:** In similar fashion, we reserve the encoding *e = 8*’b11111111 (all 1s) and m = 23’d0 to represent +∞ and –∞.

|  |  |  |
| --- | --- | --- |
| X | 8’b11111111 | 23’d0 |

*Infinity*

***NaN:*** *Not a Number (*NaN*)* is a special case that is encoded using *e =* 8’b11111111 (all 1s) and is used inside processors to represent results of computations that cannot be expressed (for example, the computation of 0/0 or ∞/ ∞ in cases where the limit does not exist).

|  |  |  |
| --- | --- | --- |
| X | 8’b11111111 | X |

*NaN*

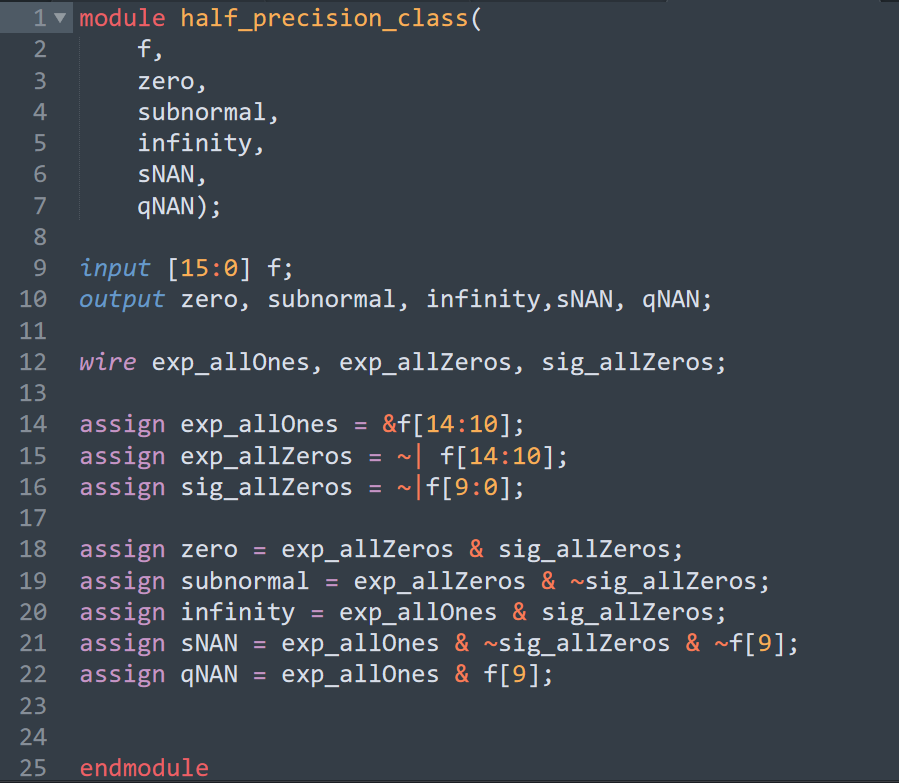
The standard defines two types of NANs – *quiet* NANs and *signalling* NANs.

A *quiet* NAN has the most significant bit of the significand cleared to zero – and at least one of the other bits must be set to one. The quiet NAN is typically the output from illegal operations such as divide by zero and is called ‘quiet’ since it is simply propagated through an operation without throwing any exceptions. The *signalling* NAN has the most significant bit of the significand set to one – and we do not care about the other bits in the significand. The signalling NAN will signal an *invalid exception* when it is presented as an operand.

**Building a toy multiplier**

We now write the logic for a control module for a toy multiplier (16-bits) to illustrate the multiplication process at a high level. This can be extended very easily to higher levels of precision by changing the length of the exponent and significand bit vectors in Verilog.

We first compute *exp\_allOnes* (set to 1 if all bits in exponent are set)*, exp\_allZeros* (set if all bits in exponent are cleared to zero)*,* and *sig\_allZeros* (set to 1 if all bits in significand are cleared to zero) and store the results into three wires.



*The Verilog code for the control module*

The pseudocode for each signal is given below.

*zero = (exponent should have all 0s) AND (significand should have all 0s)*

*subnormal = (exponent should have all 0s) AND (significand should have at least one bit set to 1)*

*infinity = (exponent should have all 1s) AND (significand should have all 0s)*

*sNAN = (exponent should have all 1s) AND (significand should have at least one bit set to 1) AND (MSB of significand should be 0)*

*qNAN = (exponent should have all 1s) AND (MSB of significand should be 0)*

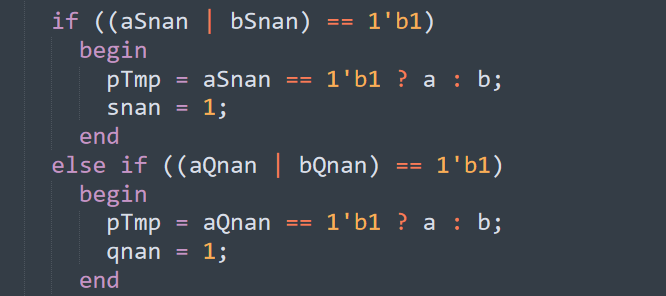


*Test bench for some selected floating-point numbers (marker shows* FC00 = –∞)

We now analyse each of the different cases that can arise.

**Case 0: One the operands is NAN**

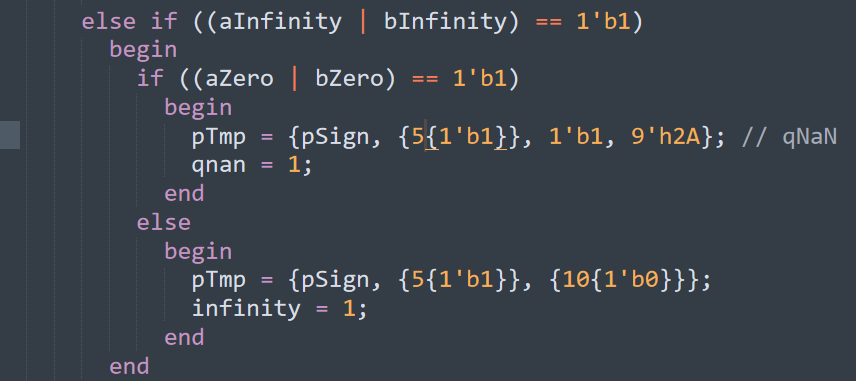
If the operand is a *quiet* NAN, it should simply be propagated through, but a *signalling* NAN should throw an exception. Since we are building a standalone multiplier, we will simply propagate both the signalling and quiet NANs through the multiplier.



*The logic used for dealing with sNANs and qNANs*

**Case 1: One of the operands is Infinity**

When infinity is multiplied by either infinity, a subnormal or normal number, the output should also be infinity. But when infinity is multiplied by zero, we output a *quiet* NAN to indicate that the result is undefined. In this implementation, we will output the simplest qNAN possible – the exponent field all set to 1, the most significant bit of the significand set to 1, and all the other bits in the significand cleared to 0.



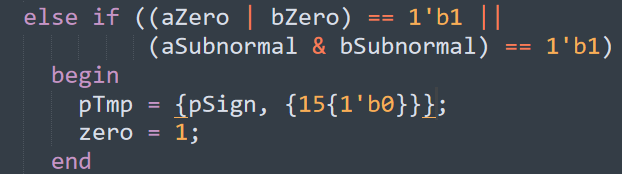
*The logic used for dealing with infinity as an operand*

**Case 2: One of the operands is zero**

If one of the operands is zero, simply output zero while preserving the sign that results from the operation.

**Case 3: Both of the operands are subnormal**

Consider the smallest normal number that can be encoded (which is 1 \* 2^ (-14)). When this is squared, the result is 1\* 2^ (-28), which is already smaller than the smallest subnormal number that can be encoded (which is 1 \* 2 ^ (-24)). Therefore, we can safely say that the product of two subnormal numbers will always be too small to be encoded as subnormal and is treated as zero.



*The logic for dealing with zero as an operand or both of the operands being subnormal (rounded to zero since it is too small)*

**Case 4: Both of the operands are normal**

If both the operands are normal, we will need to first extract the significand and exponent bits into different registers. First, the raw significand of the product will need 22 bits. (This is because 11 bits + 11 bits = 22 bits. The significand of each operand is 10 bits long, and we need an extra bit for the leading 1 that is implied).

Next, the exponent of the product needs a 7-bit register to store the sum of the exponents of the two operands. (This is because the largest exponent value possible is 15, the product can have a maximum exponent of 15 bits + 15 bits + 1 bit = 31 bits. The extra bit is added in case the product of the significands needed to be renormalized. This means that we should technically only require 6 bits, but for reasons explained later, we will use 7 bits to store the exponent of the product term.)

The raw product of the significands yields a 22-bit value (11 + 11 bits) – however, this needs to be encoded in 10 bits. For now, we will truncate the result and implement rounding behaviour later. To truncate, however, we first need to find the most significant bit set to 1 and consider that as our implied bit and the 10 bits following this MSB will be encoded as the significand of the product. One approach is to simply write a loop that keeps shifting left until we reach a non-zero MSB.

The other approach is to analyse the 22-bit raw significand that can arise from the product of the 11-bit significands. (consider that starting index is 0 and starts from the LSB)

We notice that squaring the largest significand requires us to renormalise by shifting the radix point left by one, but no such renormalization is needed for squaring the smallest significand. Thus, if bit 21 happens to be set, the radix point is moved left and the exponent is increased by 1 to account for the radix point shifting and if bit 21 is cleared, then we proceed without renormalizing. This observation makes our implementation much more efficient over a brute force approach to finding the MSB.

As seen earlier, the product of the two smallest normal numbers is too small to be stored even as a subnormal number – in this case, the result is rounded down to zero. If the product does happen to be subnormal, we store it as a subnormal number.

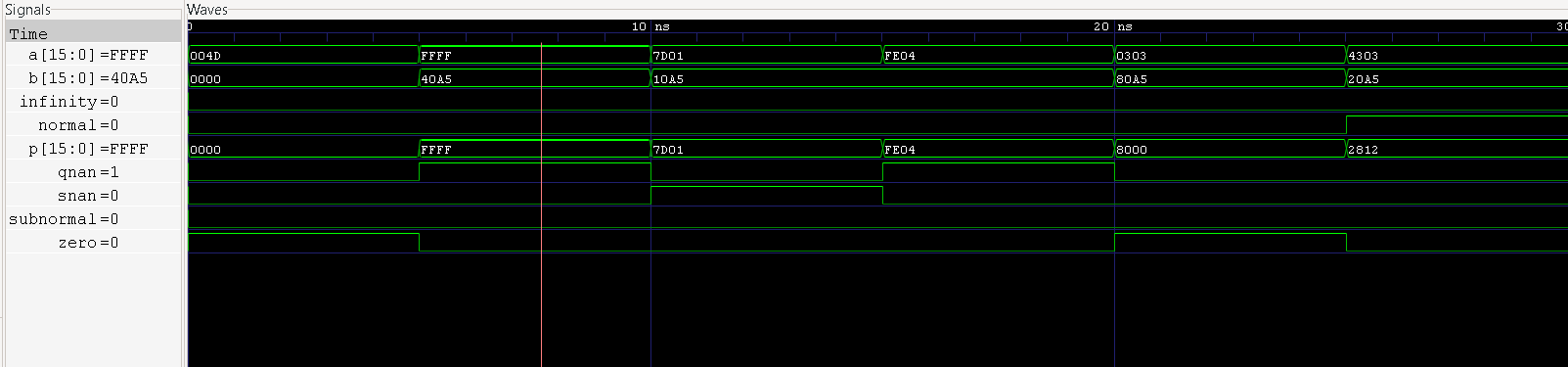
If the product of normal numbers is too large (exponent > 15), we store the product as infinity, and use the appropriate sign calculated earlier.

Finally, if the product happens to be a normal number, we add the bias value (in our case, 15) back to the exponent field and store the least significant 10 bits to the significand field and the appropriate sign bit calculated earlier.

**Case 5: One of the operands is subnormal, the other is a normal number**

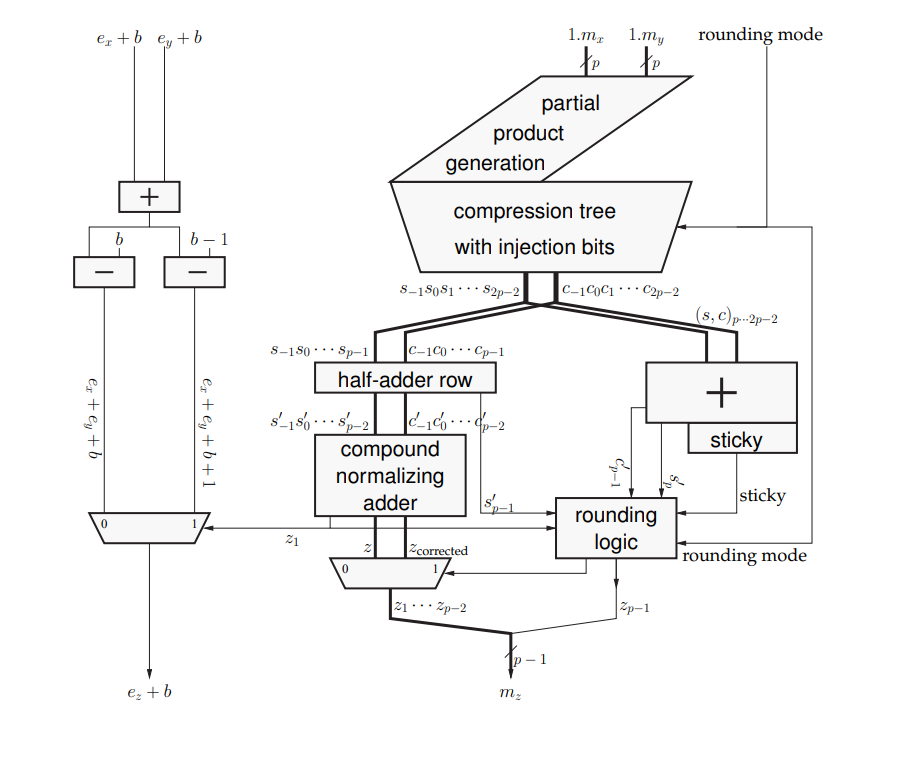
For this case, we use the results of the discussion above by normalizing the subnormal number and adjusting the exponent accordingly. The subnormal number now behaves just like a normal number – and all the results obtained above for the product of two normal number will follow.

The reason why the exponent field of the product is 7-bits wide is now clear – the smallest normal number (1\*2^-14) multiplied by the smallest subnormal number (1 \* 2^-24) is equal to (1\*2^-38). This will overflow if the exponent field is only 6 bits long (highest value being 31).



*Test bench for selected floating-point numbers (marker shows* –inf \* normal number = –inf)

# **Hardware Implementation of a Floating-Point Multiplier**

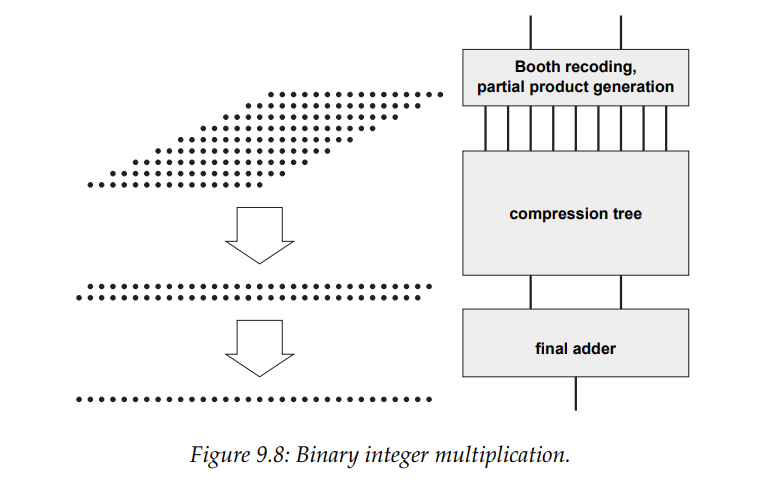


*The architecture of a floating-point multiplier (reproduced from [1])*

The exponent computation is not part of the critical path – therefore, it can be implemented using relatively slow carry propagate adders. The result of the multiplication, in the worst case, may need to be normalized by shifting left by 1 – therefore, there are two paths to account for both of these cases, which is controlled by a 2:1 multiplexor with the select signal coming from the multiplier architecture.

**The multiplier module**

The multiplier is implemented by *Booth recoding* one of the operands and generating a *partial product array,* which is then reduced to the final sum and carry using a *compression tree*. If we use the standard radix -2 multiplier, we would obtain a partial product for each bit in the multiplier operand. For a double precision representation, this would entail 53 partial products, one for each bit of the significands. This would require a large compression tree (typically comprised of several rows of carry-save adders) and a fast adder to sum the carry-save result into the final product.



*Binary multiplication (reproduced from [1])*

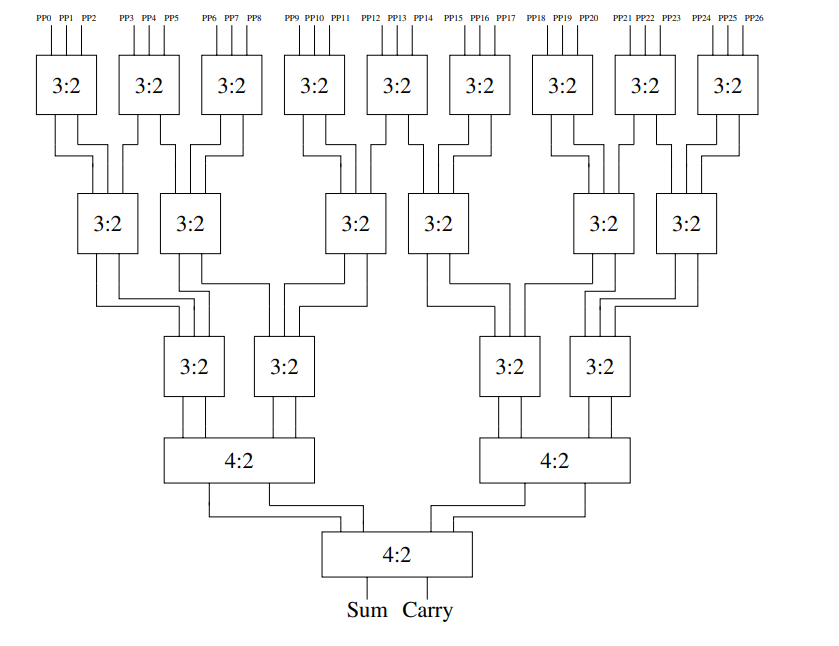
Most processors employ a *modified* Booth’s algorithm, which involves a radix-4 *recoding* of one of the operands. The larger radix halves the number of partial products generated to 27 partial products, reducing the size of the compression tree used. The recoding is performed by scanning a group of three bits from the operand at once, with a single bit overlap between each contiguous group. For a non-Booth radix-4 multiplier. each group would then be represented as a digit from the set [0, 1, 2, 3]. The problem with this representation is that while multiplication by [0,1,2] is trivial, the multiplication by 3 is non-trivial (since it is equivalent to multiplying by (2+1) and would require carry-save adder modules to compute the partial product). The modified Booth algorithm accounts for this by using the two’s complement representation which represents each group as a digit from the set [, ,0,1,2].

|  |  |  |  |
| --- | --- | --- | --- |
| b0 | b1 | b2 | Encoding |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 |
| 1 | 0 | 0 |  |
| 1 | 0 | 1 |  |
| 1 | 1 | 0 |  |
| 1 | 1 | 1 | 0 |

*The modified Booth radix-4 encoding*

**The compression tree**

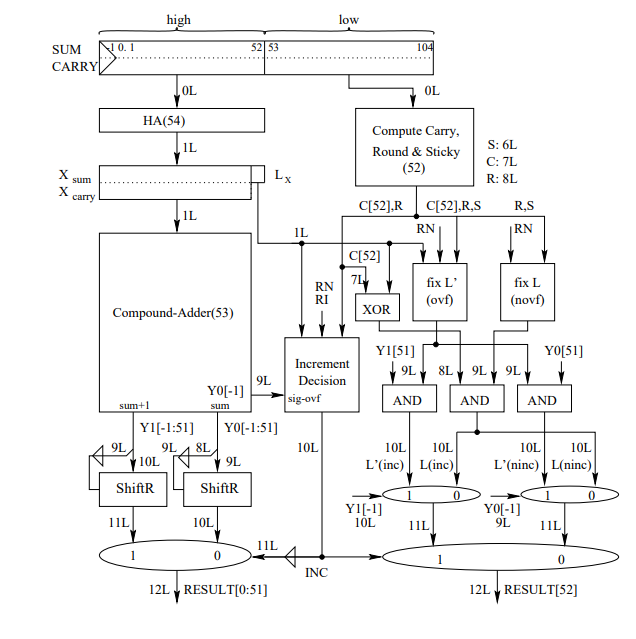
Once the partial products have been generated, they need to be summed using several rows of *compressors* (which are typically implemented as carry-save adders).



*The compression tree using 3:2 and 4:2 carry-save adders for a single-precision multiplier*

In the first level of the tree, the 27 partial products generated from the Booth multiplier are reduced to 18 partial products after being passed through the 3:2 compressors. In the second level, these 18 partial products are reduced to 12 partial products and similar reduction takes place in the third level to 8 partial products. At the final level, these 8 partial products are reduced to a sum and a carry output, which are then passed to the next stage for to generate the final product after rounding.

**The rounding logic**



*The rounding logic for a floating-point multiplier (reproduced from [3])*

The multiplier uses the ES rounding algorithm developed in [3]. The sum and carry bits from the compression tree are first divided into high [-1:52] and low bits [53:104]. The low part is then used to compute the carry, round and sticky bits given by the following equations:

The high bits are passed to a series of half-adders, which generates output sum and carry bits – these bits are then passed as input to a compound adder(generates both A + B and A + B + 1, since the rounded significand will take either A+B or its successor A+B+1 []). Depending on the control signal from the increment decision control unit, one of these is selected and passed through as the final product from the multiplexor. Note that Y [-1] = 1 implies that the result is in the range [2,4) and must be normalised.

*The logic for normalising the final product*

The IEEE 754 standard defines four rounding modes: round to zero (RZ), round to +∞, round to -∞ (RI), and round to nearest-even (RNE). RZ is the easiest among these to implement since we are simply truncating and normalising the result. Therefore, we add a few bits called *injection bits* to the partial product array that will allow us to treat all the other rounding modes as round to zero.

This hardware implementation is typically pipelined over several stages.

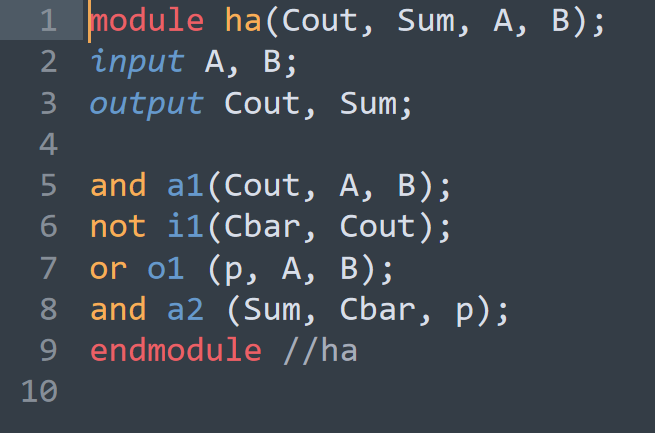
# **Conclusion**

This paper first looked at the IEEE 754 standard for floating point numbers and the properties that made it convenient to use in processors. We then looked at the floating-point multiplication algorithm and how each of the special cases of multiplication were dealt with. Finally, we looked at the hardware pipelined implementation of the multiplier and the various improvements made at each stage to reduce latency such as the implementation of the Booth modified radix-4 algorithm that generated partial products that were passed through a compression tree constructed out of several rows of carry-save adders. Finally, we looked at the rounding and normalizing algorithms employed to obtain the final product.

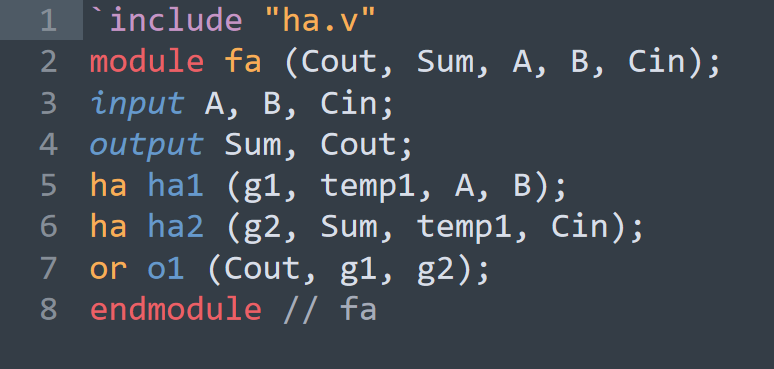
Further research involves looking at the binary *fused-multiply add* operator, which implements the binary operation A\*B + C, and first implemented in the IBM RS/6000[1].

# **Appendix: Verilog modules**

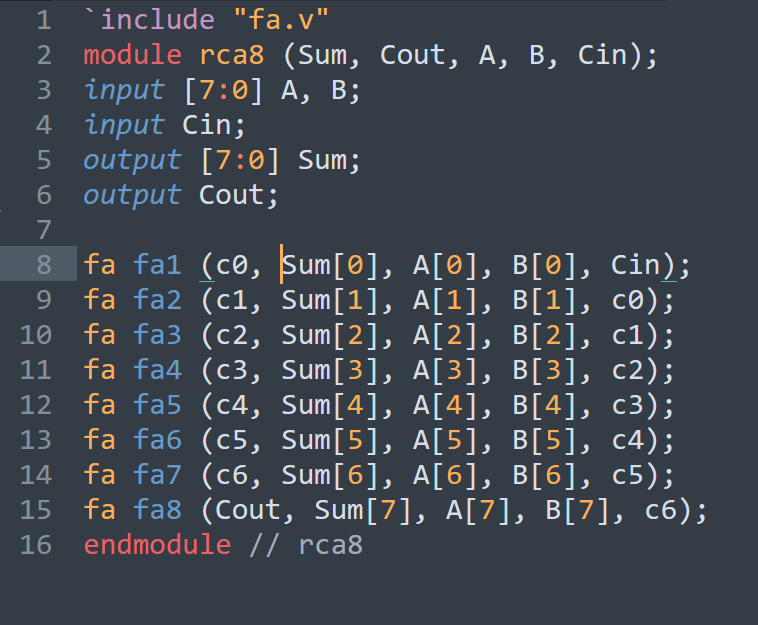
**Half Adder:**



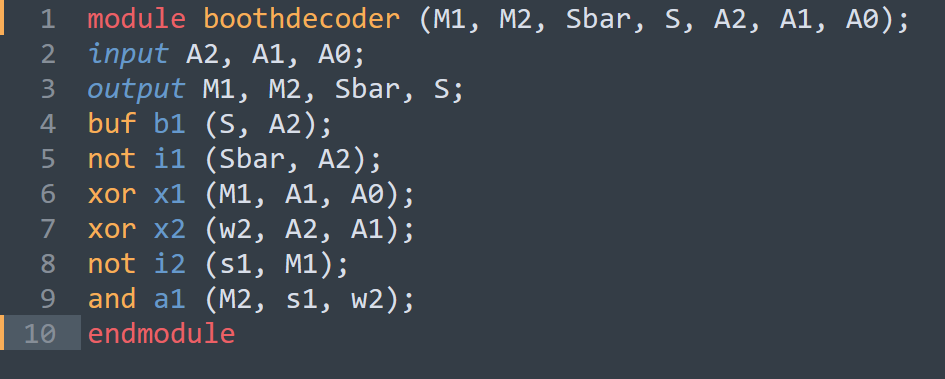
**Full Adder:**

****

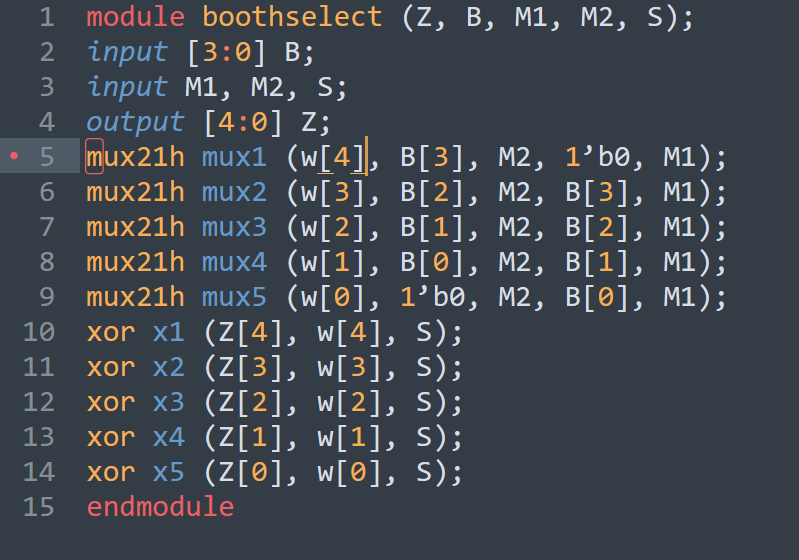
**Ripple Carry Adder:**

****

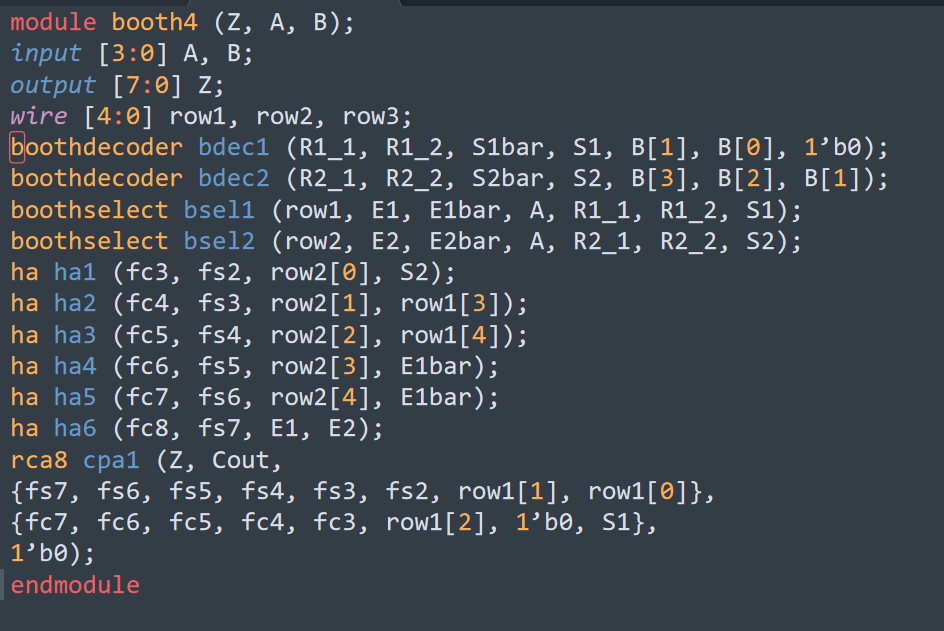
**Booth Decoder:**

****

**Booth Selector:**

****

**4-bit Booth radix-4 multiplier:**

****

# **References**

[1] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. 2018. Handbook of Floating-Point Arithmetic (2nd. ed.). Birkhäuser Basel.

[2] Son Dao Trong, Martin Schmookler, Eric. M. Schwarz, and Michael Kroener. 2007. P6 Binary Floating-Point Unit. In Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH '07). IEEE Computer Society, USA, 77–86. <https://doi.org/10.1109/ARITH.2007.26>

[3] E.M. Schwarz, “Binary Floating-Point Unit Design: the fused multiply-add dataflow”, High-Performance Energy-Efficient Microprocessor Design, Edited by V.G. Oklobdzija and R.K. Krishnamurthy, Springer, Chapter 8, 2006.

[4] S.F.Oberman, “Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7 Microprocessor,” Proc. of Fourteenth Symp. on Comput. Arith., pp. 106- 115, April 1999.

[5] Stine, James E. *Digital Computer Arithmetic Datapath Design Using Verilog HDLl*, Boston: Kluwer Academic Publishers, 2004. Print.