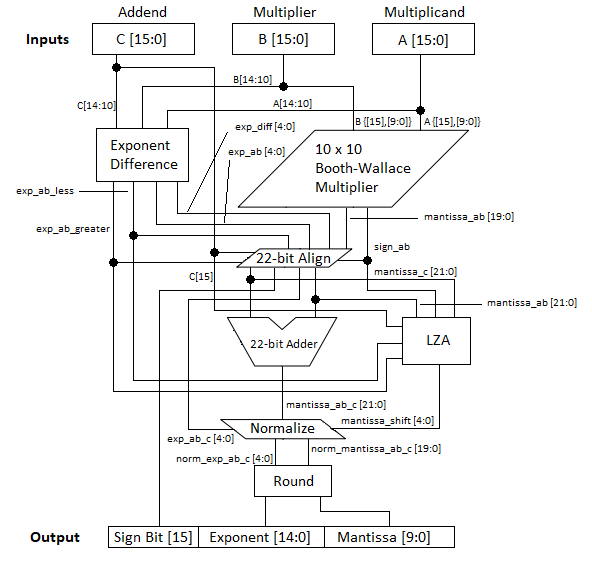
**Half-Precision Floating-Point Fused Multiplier-Adder Architecture**

By: Kaya Erdin

The architecture for a half-precision floating-point fused multiplier-adder (FMA) or multiplier-accumulator (MAC) is shown in Figure 1 below.



**Figure 1:** The architecture for a half-precision floating-point fused multiplier-adder.

The input to this unit is three half-precision floating-point numbers; namely, A, B, and C. The output is a single half-precision floating-point number. The unit’s function is to multiply two numbers (A and B) and to add the product to another number C.

The purpose of this document is to give an overview of the architecture presented above. A brief explanation of each block in the unit will be provided.

Author’s note: In most cases, addition is performed using the Kogge-Stone architecture (see *High-Speed Adders* for more information). Standard ripple-carry adders are only used in select cases where the bit widths are low (typically below 5). At this level, Kogge-Stone adders and ripple-carry adders have similar speeds, but ripple-carry adders are less complex and consume less power.

10 x 10 Booth-Wallace Multiplier:

Inputs:

* Sign bit of A (A[15])
* Sign bit of B (B[15])
* Mantissa of A (A[9:0])
* Mantissa of B (B[9:0])

Outputs:

* Sign of AB (sign\_ab)
* Product of AB (mantissa\_ab [19:0])

The function of this block is straight-forward. It multiplies the mantissas of A and B to produce a 20-bit product using the Booth-Wallace scheme. For more information on this, refer to the document that explains the Booth-Wallace multiplication (*High Speed Multipliers*). This block also includes the logic for determining the sign of the product of A and B, which is simply an XOR of the sign bits of A and B.

Exponent Difference:

Inputs:

* Exponent of A (A[14:10])
* Exponent of B (B[14:10])
* Exponent of C (C[14:10])

Outputs:

* Exponent difference (exp\_diff [4:0])
* Exponent of AB (exp\_ab [4:0])
* Flag indicating that the exponent of AB is greater than the exponent of C (exp\_ab\_greater)
* Flag indicating that the exponent of AB is less than the exponent of AB (exp\_ab\_less)

This block first calculates the exponent of the product of A and B by adding the exponents of A and B. It then finds the absolute difference between the exponents of AB and C. In the meantime, flags are also generated to indicate whether the exponent of AB is greater than or less than the exponent of C.

22-bit Align:

Inputs:

* Exponent difference (exp\_diff [4:0])
* Sign of AB (sign\_ab)
* Exponent of AB (exp\_ab [4:0])
* Mantissa of AB (mantissa\_ab [19:0])
* Flag indicating that the exponent of AB is greater than the exponent of C (exp\_ab\_greater)
* Flag indicating that the exponent of AB is less than the exponent of AB (exp\_ab\_less)
* Sign bit of C (C[15])
* Exponent of C (C[14:10])
* Mantissa of C (C[9:0])

Outputs:

* Sign bit of AB+C (sign\_ab\_c)
* Exponent of AB+C (exp\_ab\_c [4:0])
* Aligned mantissa of AB (mantissa\_ab [21:0])
* Aligned mantissa of C (mantissa \_c [21:0])

The 22-bit Align block extends the most significant bit of the mantissa of AB by prepending it with the implicit bits {01} if the exponent is not all zeros or {00} if the exponent is all zeros for subnormal numbers. It also extends the mantissa of C by prepending it with {01} or {00} and appending it with 10 zeros to match the current size of the mantissa of AB. If the exponent of AB is greater than the exponent of C, the current mantissa of C is right-shifted by the exponent difference between the two, and exp\_ab\_c is set to the exponent of AB. If the exponent of AB is less than the exponent of C, the current mantissa of AB is right-shifted by the exponent difference between the two, and exp\_ab\_c is set to the exponent of C.

The sign of AB+C is determined by comparing the exponents of AB and C, or the mantissas of AB and C if the exponents are equal, as well as the respective sign bits. For instance, if both the sign bits of AB and C are “0” (i.e. positive), then the result of AB+C will also be “0”. If both the sign bits of AB and C are “1” (i.e. negative), then the result of AB +C will also be “1”. In the case where the sign bits differ, the exponents or the mantissas (if the exponents equal) of AB and C are compared in order to determine which value dominates, and to set the sign bit of AB+C accordingly.

Finally, the two’s complement forms of the aligned mantissas are found and output depending on the state of the sign bits.

22-bit Adder:

Inputs:

* Aligned mantissa of AB (mantissa\_ab [21:0])
* Aligned mantissa of C (mantissa\_c [21:0])

Output:

* Mantissa of AB+C (mantissa\_ab\_c [21:0])

This block adds the aligned mantissas output from the 22-bit Align block and outputs a 22-bit result. If the most significant bit is one, which indicates a value in two’s complement form was added, the two’s complement form of the sum is reverted to standard form.

LZA:

Inputs:

* Sign bit of AB (sign\_ab)
* Sign bit of C (sign\_c)
* Flag indicating that the exponent of AB is greater than the exponent of C (exp\_ab\_greater)
* Flag indicating that the exponent of AB is less than the exponent of AB (exp\_ab\_less)
* Aligned mantissa of AB (mantissa\_ab [21:0])
* Aligned mantissa of C (mantissa\_c [21:0])

Output:

* Mantissa shift amount (mantissa\_shift [4:0])

The leading-zero anticipator (LZA) is a block that works in parallel with the 22-bit adder. Its function is to predict the mantissa shift amount based on the sign bits, sizes, and aligned mantissas of AB and C. At its most basic level, it is a leading-zero (or one for negative values) counter.

In terms of sign bits, there are four possible cases: (1) both AB and C are both positive; (2) AB is negative while C is positive; (3) AB is positive while C is negative; and (4) AB and C are both negative. The first case will always yield a positive result while the last case will yield a negative result. The middle two cases depend on the magnitudes of both addends. For instance, suppose AB is positive and C is negative. If the magnitude of AB is larger than the magnitude of C, then addition of AB and C will yield a positive result. On the other hand, if the magnitude of C is larger than the magnitude of AB, AB+C will be negative. Such cases must also be taken into account in the LZA.

The LZA uses the P, G, and Z signals that define the bit-to-bit relations of the two operands mantissa\_ab and mantissa\_c:

P[i] = mantissa\_ab[i] ^ mantissa\_c;

G[i] = mantissa\_ab[i] & mantissa\_c;

Z[i] = ~(mantissa\_ab[i] | mantissa\_c).

Let’s start off by taking a look at the first case. Since both operands are positive, they are bound to have leading zeros. Therefore, the LZA should be counting Z’s from the most significant bit (bit 21) downward and stop counting when another state is encountered. The shift amount to be output is the counter value MINUS the carry of the addition of the two bits at the first non-Z position (a carry of “1” will reduce the number of shifting positions by one) MINUS 1 since we’re trying to bring the leading “1” to bit 20 so that the implicit two most significant bits are {01}.

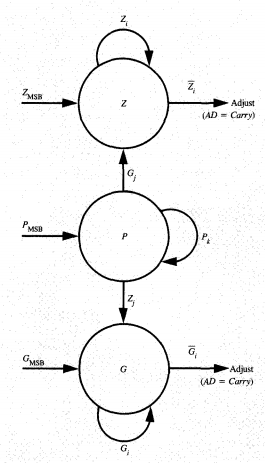
The last case is almost identical to the first case except that we are counting G’s or leading ones, instead.

The second case is where the fun begins. Here, the signs of the operands differ, so we cannot conclude whether the result of AB+C will be positive or negative, yet. Instead, we have to look at the magnitudes of the two operands. If AB is greater, the sum will be negative. Since AB is negative and C is positive, we also know that mantissa\_ab will have leading ones while mantissa\_c will have leading zeros. Therefore, we start off by counting P’s. Assuming that we have counted P’s so far, the next state we should detect is one Z. Once we have detected it and incremented our counter, we should then anticipate and count G’s until we reach another state, at which point we stop counting and output the counter value minus 1, adjusted by the carry at the bad state.

If AB is less than C, the sum will be positive. Since AB is negative and C is positive, we also know that mantissa\_ab will have leading ones while mantissa\_c will have leading zeros. Therefore, we start off by counting P’s. Assuming that we have counted P’s so far, the next state we should detect is one G. Once we have detected it and incremented our counter, we should then anticipate and count Z’s until we reach another state, at which point we stop counting and output the counter value minus 1, adjusted by the carry at the bad state.

The third case uses the same conditions as the second case, except AB is now positive while C is now negative.

The state diagram of the bit-serial leading-zero/one anticipator for adding positive and negative numbers is shown in Figure 2 below.



**Figure 2:** The state diagram of the bit-serial leading-zero/one anticipator for adding positive and negative numbers.

For more information on the theory behind the anticipator, refer to [1].

Normalize:

Inputs:

* Exponent of AB+C (exp\_ab\_c [4:0])
* Mantissa of AB+C (mantissa\_ab\_c [21:0])
* Mantissa shift amount (mantissa\_shift [4:0])

Outputs:

* Normalized exponent of AB+C (norm\_exp\_ab\_c [4:0])
* Normalized mantissa of AB+C (norm\_mantissa\_ab\_c [19:0])

First, the normalizer looks at the most significant bit of the mantissa of AB+C. If it is “1”, then the mantissa must be right-shifted by one position in order to shift the implicit one to the correct position. Consequently, the exponent is incremented by one. If the most significant and second-most significant bits are both zero, and the exponent of AB+C is not all zeros, then the mantissa of AB+C is shifted by the amount predicted by the LZA. Otherwise, the mantissa is not shifted. Since the implicit most significant two bits are no longer needed by the unit, they are discarded and the bottom twenty bits are output.

Round:

Inputs:

* Normalized exponent of AB+C (norm\_exp\_ab\_c [4:0])
* Normalized mantissa of AB+C (norm\_mantissa\_ab\_c [19:0])

Outputs:

* Rounded exponent of AB+C [4:0]
* Rounded mantissa of AB+C [9:0]

This block could be merged with the normalizer depending on timing constraints, but it has been separated here for illustrative purposes. The rounder applies “round to nearest; ties to even” rounding or “banker’s rounding” to the normalized mantissa. This rounding scheme was chosen since it has the least bias amongst all the rounding schemes. The rounder works by comparing the least significant bit position of the to-be rounded value (bit 10) with the bit right below and all other bits below that. If bit 9 is “1”, and there is a “1” in one of the bits below, the upper 10 bits of the mantissa are incremented by one (i.e. rounded up), and the bottom 10 are discarded. If bit 10 is “1”, bit 9 is “1”, and there is no “1” in one of the bits below, the upper 10 bits of the mantissa are still incremented by one in order to round to the nearest even value (i.e. rounded up), and the bottom 10 are discarded. Conversely, if bit 10 is “0”, bit 9 is “1”, and there is no “1” in one of the bits below, the mantissa is truncated to the upper 10 bits (i.e. rounded down) in order to keep the value even. In all other cases the mantissa is truncated to the upper 10 bits of the mantissa.

Note that if after incrementing the mantissa a carry emerges, the exponent must also be incremented by 1.

One Final Thing...

Before the FMA outputs the final half-precision floating-point value, amendments must be made to the special case where either one of the exponents of A, B, or C is 5’b11111 (i.e., A, B, or C is infinite or NaN). Obviously, anything added to infinity is infinity, so if the exponent of C is 5’b11111, then the exponent of the final result is infinity (or NaN if trying to subtract infinity from infinity). Likewise, any non-zero number multiplied by infinity is infinity. On the other hand, zero multiplied by infinity yields NaN (this seems to be mathematically incorrect as the correct answer is zero, but IEEE 754 classifies it as such).

**References:**

[1] E. Hokenek and R. K. Montoye, “Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-

point execution unit,” *IBM Journal of Research and Development*, vol. 34, no. 1, pp. 71–77, 1990.