# Optimized Leading Zero Anticipators for Faster Fused Multiply-Adds

David R. Lutz
ARM Austin Design Center
david.lutz@arm.com

## Abstract

Leading zero anticipators (LZAs) predict the number of leading zeros in a difference, and have long been used to speed up floating-point add and fused-multiply add (FMA) operations. Typically the LZA prediction must be corrected for small exponents and possible carries, and only the corrected value is useful for normalizing the difference and computing rounding. We present new techniques to speed up LZA correction and rounding, allowing differences to be computed in only 60% of their previous best latency.

## I. Background

We perform IEEE 754-2008 compliant FMAs [6] as separate multiplies and adds, a strategy that improves the latency of addition, multiplication, and sums-of-products [3], [2]. For double precision (DP) this means the multiplier returns an unrounded result comprising sign, 11-bit exponent, and 105-bit fraction; one of the adder inputs also has this format, with the extra fraction bits zeroed for non-FMA adds.

Our adder uses the near/far split first proposed in [1], with the far path handling all like-signed adds (effective additions) as well as unlike-signed adds (effective subtractions) that have true exponent differences of two or more, and with the near path handling all unlike-signed adds with true exponent differences of zero or one. Our design has some extra complications due to subnormal support, and even more due to the double-length significand, which necessitates rounding even on the near path. Probably the greatest complication is due to the greatly reduced logic depth, with this design requiring only 50 or so FO4 (fanout-4) inverter delays for either add path.

Figure 1 shows a simplified far datapath.

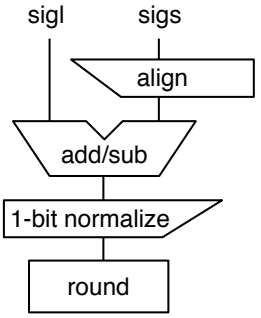


Fig. 1. Simplified Far Datapath

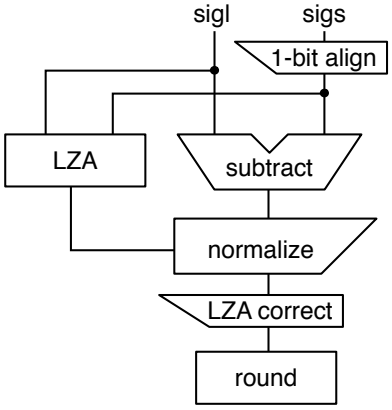


Fig. 2. Simplified Near Datapath

The significand with the smaller exponent (sigs) is right shifted until the exponent matches the significand with the larger exponent (sigl), the aligned significands are added or subtracted, a trivial 1-bit normalization is performed (necessary if there is a carry out of the sum or difference), and the result is rounded to the DP format. The triviality of the normalization step means there is no LZA on the far path, so we will not discuss it any further in this paper.

Figure 2 shows a simplified near datapath.

Because the exponents can differ by one, there is a trivial alignment step in which the smaller significand can be right shifted by one bit position. The aligned significands are subtracted, and at the same time the LZA block predicts how many leading zeros are in the difference. The LZA prediction is used to left-shift (normalize) the difference. The LZA can be off by one because it does not try to predict if there is a carry out of the low order bits of the difference, so a zero- or one-bit LZA correction shift is required. Rounding is also required because the 106-bit input significand can cause the difference to have more than 53 significant bits.

Our goal in this paper is to make the near-path adder as fast as possible. The basic near-path significand logic of subtraction and normalization does not seem to be reducible, so we need to make sure that the normalization shift is completely set up by the time the subtraction completes, which

is equivalent to saying that we need an optimal LZA. LZAs have been well studied [5], but some problems remain:

1) the predicted value can be higher than the maximum normalization shift if the exponents are small
2) the predicted value can be one higher than the actual value if there is a carry out of the low order bit of the difference
3) it is difficult to obtain rounding information from the difference before normalization

Problem 1 will be handled in the next section, and problems 2 and 3 will be handled by constructing masks as described in section III.

## II. Limiting the Normalization Shift for Small Exponents

Our basic LZA constructs a string $w$ based on the two n+1-bit inputs, and then counts the number of leading zeros in $w$. One elegant method [4] for constructing $w$ is to set $w[n] = 0$, and then

$w[n-1:0] = \overline{p[n:1] \oplus k[n-1:0]}$

where $p$ and $k$ are the propagate and kill signals at each bit position. The delay of constructing $w$ is essentially the delay of two XOR gates, or about four FO4 delays.

If the exponents are small, the LZA prediction can be too large. For example, suppose that the LZA predicts 15 leading zeros, but the biased exponent $expl$ of the larger input is smaller than that. Every one-bit left shift of the difference decreases the result exponent by one, but we cannot represent an exponent less than the biased minimum exponent $e_{min} = 1$.

This problem is discussed in the *Handbook of Floating-Point Arithmetic* [4], where it suggests contructing a second shift-limiting string $lim[n:0]$ by shifting a 1 to position $n - (e - e_{min})$. If we perform a logical OR operation, $lim[n:0]$ OR $w[n:0]$, then a leading zero count of that string will give us the correct limited shift distance (see figure 3).

The Handbook further notes a problem with this approach (our $expl = e$ is their $e_x$):

> Another solution would be to inject a dummy leading 1 in the proper position $(e_x - e_{min})$ before leading zero counting. Again, $e_x - e_{min}$ may be computed in parallel with the exponent difference, but bringing a 1 to this position is a shift. It is difficult to obtain this value before the leading zero count.

The contribution of this section is to show that we do not need to compute $e - e_{min}$, and that moving a 1 bit to an encoded location is not a shift. Setting a single bit to an encoded value is accomplished with simple logic, and given the fact that $e_{min} = 1$, the subtraction is easily dealt with by manipulating the assignments to chosen locations within $lim[n:0]$.

Consider a much abbreviated example where we have 6-bit biased exponents and 7-bit significands. If the exponent is greater than 7, then there is no limitation on the shift. For $e < 8$, the maximum normalization shift is is $e - 1$, so we place a one at location $n - (e-1) = n - e + 1$ to limit the shift. Because we can exactly identify $e$ with simple logic, the construction of $lim$ is straightforward:

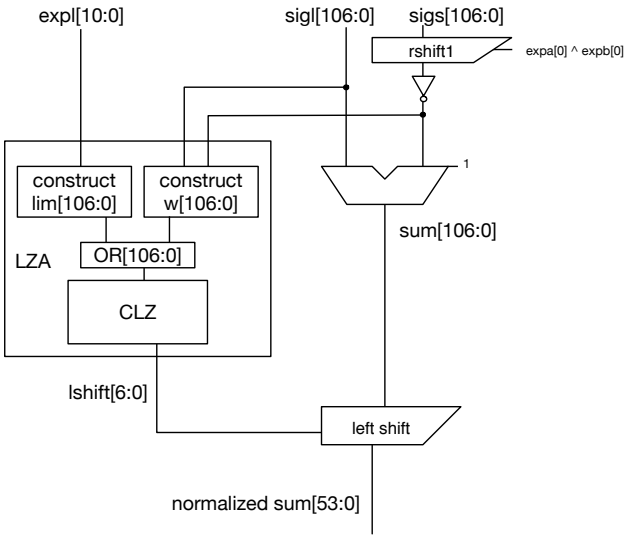


Fig. 3. LZA with limited normalization

```
e_gt_7 = e[5] | e[4] | e[3];              // e>7
lim[7] = ~e_gt_7 & ~e[2] & ~e[1];         // e=0 or 1
lim[6] = ~e_gt_7 & ~e[2] &  e[1] & ~e[0]; // e=2
lim[5] = ~e_gt_7 & ~e[2] &  e[1] &  e[0]; // e=3
lim[4] = ~e_gt_7 &  e[2] & ~e[1] & ~e[0]; // e=4
lim[3] = ~e_gt_7 &  e[2] & ~e[1] &  e[0]; // e=5
lim[2] = ~e_gt_7 &  e[2] &  e[1] & ~e[0]; // e=6
lim[1] = ~e_gt_7 &  e[2] &  e[1] &  e[0]; // e=7
lim[0] = 0; // never used
```

We handle $e - e_{min}$ not by computing it, but by adjusting the lim assignments. For example, if $e = 3$, then the maximum normalization shift is $e - e_{min} = e - 1 = 2$, so we set a 1 at $lim[5]$. No other bit is going to match the $e = 3$ logic, so all of the other $lim$ bits are zero. Note that both subnormal ($e = 0$) and minimum normal ($e = 1$) exponents allow no left shifting. Large exponents, in this case exponents greater than 7, don't set any bits of $lim$, and thus don't limit the LZA in any way.

Larger 11-bit exponent values and 107-bit limit strings work the same way. The $lim$ timing is set by the fanout of the exponent bits (4 FO4 inverters), then about 5 FO4 delays consisting of simple 2- and 3-input gates. Constructing $lim$ in only 9 FO4 delays is fast enough to take the enhanced LZA timing off of the critical path.

## III. Intelligent Masking

The problem with detecting LZA misprediction or gathering rounding information before normalization is that the desired bits are somewhere in the middle of the unshifted sum. Consider figure 4.

If the LZA has correctly predicted $k$ leading zeros, then the integer bit $i$ is at position $106 - k$. There are then 52 fraction

| sum[106:0] | 106 | 105 | 104 | ... | | 106-k | | ... | | 54-k | 53-k | 52-k | ... | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| sum (correct LZA prediction) | 0 | 0 | 0 | … | 0 | i | f | … | f | f | L | G | s | s |
| sum (when LZA mispredicts) | 0 | 0 | 0 | … | i | f | f | … | f | L | G | s | s | s |
| mask | 0 | 0 | 0 | … | 0 | 1 | 0 | … | 0 | 0 | 0 | 0 | 0 | 0 |
| smask | 0 | 0 | 0 | … | 0 | 1 | 1 | … | 1 | 1 | 1 | 1 | 1 | 1 |

Fig. 4. Finding the important bits

bits with the last fraction bit $L$ at position $53-k$, the guard bit $G$ at position $52-k$, and then $52-k$ trailing sticky bits. An LZA mispredict shifts everything left by one bit.

Our method for extracting these bits is to construct a mask[106:0] with a 1 in position $106-k$, then applying the mask to the unshifted sum to find $i$, $L$, and $G$. For example:

- compute the AND of $sum[106:1]$ and $mask[105:0]$ followed by a reduction OR to compute whether the LZA mispredicted.
- compute the AND of $sum[55:0]$ and $mask[106:51]$ followed by a reduction OR to compute $L$ in the LZA mispredict case.
- compute the AND of $sum[54:0]$ and $mask[106:52]$ followed by a reduction OR to compute $G$ in the LZA mispredict case or $L$ in the LZA correct prediction case.
- compute the AND of $sum[53:0]$ and $mask[106:53]$ followed by a reduction OR to compute $G$ in the LZA correct prediction case.

The method for extracting sticky bits is similar, but the sticky mask smask[106:0] needs to contain a string of ones from positions $106-k$ to 0 rather than a single one at position $106-k$.

We have used this mask scheme for several designs. Earlier designs constructed the masks (using logic) from the encoded LZA output. More recently we realized that encoding a shift value and then decoding that value to create a mask is unnecessary work. The value to be encoded by the CLZ block in figure 3 is very nearly a mask in its own right: it has a leading one in the correct place, followed by some mix of zeros and ones. All we need to do is replace the trailing ones with zeros to create mask[106:0], and to replace the trailing zeros with ones to create smask[106:0].

This replacement is easily done with logic, as shown in figures 5 and 6. We use a smaller 16-bit string to show the concept, but the logic is easily extended to the full masks needed for double precision. The input is a string w[15:0] with zeros followed by a leading one at some position, followed by any arbitrary bit sequence. Figure 5 simply computes some logical OR values for groups of bits needed later. Figure 6 zeros out all bits to the right of the first one in w. For example, mask bit m[14] is set if and only if w[14] is one and w[15] is zero. The logic to create smask[15:0] is similar, but the AND gates with one inverting input are all replace by OR gates.

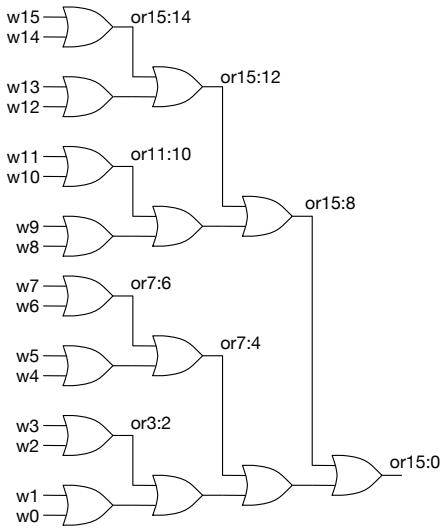


Fig. 5. OR tree for masks

## IV. Conclusion

We have shown useful LZA enhancements that enable much faster near-path addition. In particular, we have shown that a shift limiting string corresponding to $e - e_{min}$ can be constructed quickly by using simple logic. We have also shown that useful LZA-based masks can be constructed before LZA encoding, allowing both early detection of the need for LZA correction and early access to rounding information.

Our designs are all synthesized. We have used some wire-mitigating techniques (particularly in the CLZ block of figure 3), but in general the floating-point adder paths are logic dominated. Our top (read worst) paths have the equivalent of 40 FO4 delays in a cycle, this without being the frequency limiter for the core, so the wire delay on these paths is truly minimal. Since the longest adder path is only about 50 FO4 delays, we don't use much of the second cycle, and we are able to do extensive forwarding to other blocks. In particular, an FMA-based sum of products can forward the accumulator every other cycle, so we can do a left-to-right in-order sum of $n$ products using FMAs in only $2*n+1$ cycles. The add-to-add latency of two cycles is also notable compared to other designs.

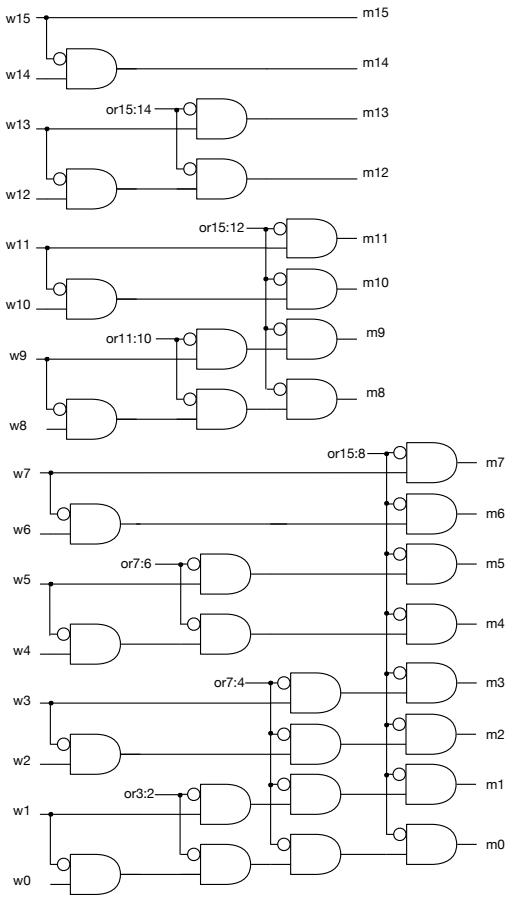


Fig. 6. First one mask

Can we go any faster? We think we are approaching the double-precision logic-depth limit, at least for FMA-capable designs that handle subnormals in hardware. If we don't require the adder to handle FMAs then we can do a bit better, due to the smaller significand adders and shifters and especially to the lack of the need to round on the near path. We probably could squeeze the logic for such a near-path adder into a single cycle, but it would not be able to do much forwarding in that cycle.

Our actual FP adder design is considerably more complicated than what we have presented here, but we have shown the main elements of the near path, formerly our worst path in a 3-cycle add. That it is no longer the worst in a 2-cycle add (same process and cycle time), speaks to some significant algorithmic improvements.

## REFERENCES


[1] Paul Michael Farmwald. *On the Design of High Performance Digital Arithmetic Units*. PhD thesis, University of California Livermore, 1981.

[2] Sameh Galal and Mark Horowitz. Latency sensitive fma design. In *20th IEEE Symposium on Computer Arithmetic*, pages 129–138, July 2011.

[3] David Raymond Lutz. Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines. In *20th IEEE Symposium on Computer Arithmetic*, pages 123–128, July 2011.

[4] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[5] Martin S. Schmookler and Kevin J. Nowka. Leading zero anticipation and detection – a comparion of methods. In *15th IEEE Symposium on Computer Arithmetic*, pages 7–12, June 2001.

[6] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, 2008.