Redundant Data Formats for Faster Floating Point Addition

Abstract

This paper presents an approach that minimizes the latency of a floating point adder by leaving outputs in a redundant form. As long as the redundant outputs are not needed by the following instruction there is time to convert them back to a non-redundant format before sending them to the register file/memory. The method of conversion acts a pipelined floating point adder with a few additional constraints that are elaborated on in the paper. This paper also deals with how to compensate when the successor instruction needs the result of the previous addition. This approach will be of additional use whenever a group of floating point additions needs to be done recursively such as is the case with conditional iterations.

I. Introduction

The IEEE floating point standard [1] has difficulty dealing with redundant formats. This problem has been addressed by other authors [2-8] who have used redundancy within the floating point adder in various ways. We have kept the redundant format within the floating point adder so that it seems transparent to anything outside of it, such as a register or memory where the output is being directed. When looking at the IEEE standard two path floating point adder [1], as shown in Fig. 1, it becomes apparent that there are two primary operations being carried out for each addition. A variable shift and a mantissa addition, each of which have an fundamental latency of O(lg n), are carried out in both the near and far paths in a reflective order. Both paths are carried out in parallel and the answer is chosen once both paths have completed their operations.
The focus on two operations carried out reflectively in parallel with two almost identical operations leads to some interesting obstacles when it comes to trying to reduce the latency of the overall algorithm. Firstly if it is possible to reduce the latency of either the variable shift or the mantissa addition below that of $O(\lg n)$, than it would not evenly affect the opposite paths latency and would therefore be a useless reduction. The other problem is that shifting is always a function of the length of the number and is very limited in how it can be reduced.

<table>
<thead>
<tr>
<th>Near Path</th>
<th>Far Path</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leading Zero Calculation in parallel with Mantissa Addition</td>
<td>Exponent subtraction, Alignment of Operands</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>Post-Normalization Shift</td>
<td>Mantissa Addition and Rounding</td>
<td>$O(\lg n)$</td>
</tr>
</tbody>
</table>

To further examine the problem it is a good idea to look at the two paths more closely. For this purpose we let $A = MA \cdot 2^e$ and $B = MB \cdot 2^e$ be the floating point numbers that are going to be added. Both of the paths are tried in parallel and work as follows:

(1) The near path assumes that the exponent difference of the two numbers is within 1 of each other, $|e_A - e_B| \leq 1$. This path can lead to catastrophic cancellation where a large number of zeroes would be left in the front of the output and will need to be shifted to the left by the number of leading zeroes to account for this. In this case, four answers are prepared: $(A - B), (B - A), (A - B/2), (B - A/2)$, each of which accounts for which of the two numbers is larger by up to one exponent so that the output is positive, and the correct one is selected. After the correct output is selected the output is shifted to normalize the answer, such that a 1 is in the MSB, when cancellation has occurred.

(2) The far path calculates the exponent difference, $(e_A - e_B)$, before adding the mantissas, then shifts the smaller operand to equate the exponents of the two numbers and performs
mantissa addition and rounding.

From this we can see that if we changed mantissa addition to have a latency of $O(1)$ and left the output in a redundant form it would not reduce the latency since the near path needs Leading Zero Calculation (LZC) which cannot be independent of word length (i.e., it cannot be $O(1)$).

II. Faster Floating Point Addition

Our goal became to answer this question: “So how do we reduce the $2 \log n$ latency further?” By performing redundant addition and leaving the output in un-shifted form we can reduce the $2 \log n$ latency down to $\log n$. The modified paths, as shown in Fig. 2, leave the outputs in a redundant format and pipeline the conversion of the outputs back to a normalized format to a later time since the output is not necessarily needed right away by another instruction.

<table>
<thead>
<tr>
<th>Near Path</th>
<th>Far Path</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leading Zero Calculation in parallel</td>
<td>Exponent subtraction,</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>with Mantissa Addition</td>
<td>Alignment of Operands</td>
<td></td>
</tr>
<tr>
<td>Redundant Output, Not Shifted</td>
<td>Mantissa Addition and Rounding,</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>Output in redundant form</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Modified Near and Far Paths with outputs left in a redundant and not re-normalized format.

We therefore note that if the result of this addition is not required by immediate successor instruction there is time to do redundant–to–non-redundant conversion so that what goes back to register file/memory is in usual non-redundant.

We are therefore tackling what happens if the successor instruction needs the result of this instruction. For two successive additions $(A+B)+C$, each individual addition could take the near (N) or Far (F) path. Accordingly there are 4 possibilities: F→F, N→F,
F→N, and N→N. For the first 3 cases, those involving the far path, even if the result of the first addition is left in a redundant and non-normalized format, it poses no problem:

(1) F→F: In this case, the post normalization shift is at most 1 place and it is calculated in $O(1)$ time (because it is guaranteed to be 0 or 1) in parallel with mantissa addition.

Now the actual shifting of the result of first addition ($A + B$) can be incorporated in the first step of the following addition. Note that in the far path ($A+B$) and $C$ would be aligned via a shift anyway. The delay of this alignment is used to mask the delay of shifting ($A+B$) to normalize it by doing both operations in parallel.

(2) N→F: For this case the main point is that at the end of the first step in the near path, the number of leading zeroes is known. Therefore, the amount of shift required to re-normalize ($A+B$) is known but, the actual shift has not been done by the shortened near path. This is no problem, since this shift can be folded into the alignment shift of the successor addition’s far path.

(3) N→F: Here the required shift is guaranteed to be no more than 1 bit and can therefore be done in $O(1)$ time. So this case does not cause a problem.

(4) N→N: This case causes a problem which we solve in the following manner.

Basically the near path leaves ($A+B$) in an un-shifted form so that in the next clock cycle one cannot directly add/subtract mantissas. Let ($A + B$) = $R = M R 2^{e R}$.

For the near path to be valid this condition must be met: $|e_A - e_B| \leq 1$. Assume $e_A \geq e_B$ and hence $e_R = e_A - \delta$, where $\delta$ equals the number of leading zeroes in the subtraction ($m_A - m_B$). The problem is that $\delta$ and hence $e_R$ is not known till the end of the first step in the near path.
One way out is to pre-shift the second operand for the next addition (i.e. $C$) to align its exponent with $e_A$. So the second addition can be written as:

$$(A+B)+C = M_r2^{-d} \cdot 2^{e_A} + MC \cdot 2^{e_C} = (M_r2^{-d} + MC \cdot 2^{e_C - e_A}) \cdot 2^{e_A}$$  \hspace{1cm} (1)$$

Note that $e_A$ is available very quickly toward the start of the near path, while the $\delta$ is not. The result of $(m_A - m_B) = M_r2^{-d}$ is (conceptually) contained within single-precision as depicted in Figure 3:

![Figure 3: Illustration of catastrophic cancellation valid for second addition.](image)

We now figure out the limit of shift $(e_C - e_A)$. For this we impose the condition that near path is:

$$|e_C - (e_A - \delta)| \leq 1 \text{ or } |(e_C - e_A) + \delta| \leq 1$$

$$1 \geq (e_C - e_A) \geq -\delta + 1$$  \hspace{1cm} (2)$$

Since $\delta$ is not known ahead of time and can be of length $n - 1$, when not accounting for rounding bits, the shift $(e_C - e_A)$ can be as large as a word length $- 1$. Taking into account that $e_C$ can be at most 1 more than $e_A$ and at the worst $n-1$ less than $e_A$, a $2n$ digit adder as is shown in Figure 4 is therefore required.
The resulting pipeline looks something like Figure 5:

**Clock 1:** Perform \((A - B)\) and in parallel pre-shift \(C\)

**Clock 2:** Add \((A - B)\) and pre-aligned \(C\)

Figure 5: Sample pipeline.

Note that this result could need to be post-normalized by up to \(2n\) digits in the worst case. This would seem to indicate that to perform third subsequent near path \((N \rightarrow N \rightarrow N)\) addition, the adder length would have to be \(3n\). However, this problem is circumvented in another way. Note the \((N \rightarrow N \rightarrow N)\) scenario arises when adding four numbers: \((A + B) + C + D\) using a single floating point adder.

To solve this problem, we first perform \((A + B)\) then perform \((C + D)\). Now there is time to shift the outcome of \((A + B)\) by any desired amount. Note that the \(\delta\) for the first operation is known before 2nd operation starts. In other words \(e_R\) resulting from \((A - B)\) is known. Hence the new shift required is to align \(e_R\) with \(e_C\) (if \((C - D)\) near path is valid then \(e_C\) and \(e_D\) differ by at most 1). Without the loss of generality assume \(e_C \geq e_D\) (so that \(e_D\) can be factored out). Now the problem is same as that original \(N \rightarrow N\) and can be handled by the \(2n\)-digit long adder. Thus the shifter could be of length up to \(2n\) also.
The resulting pipeline is sketched in Figure 6:

**Clock 1:** Perform $(A-B)$. Note that delta is known by the end of this cycle

**Clock 2:** Perform $(C-D)$ and in parallel shift $(A-B)$

**Clock 3:** Add $(C-D)$ to aligned $(A-B)$ (analogous to $N\rightarrow N$ in previous figure)

Figure 6: Once $N\rightarrow N$ is taken care of then $N\rightarrow N\rightarrow N$ is done by changing the order of operations.

Note that adding five or more operands are now covered as well. In order to perform $(((A+B)+C)+D)+E$ the pipeline would look as follows:

Clock 1: $A+B$
Clock 2: $C+D$, normalize $R1 = (A+B)$
Clock 3: $R1 + E$, shift $R2=(C + D)$ to align with $(R1+E)$  
Clock 4: Add $R1+E$ and aligned $R2$

**Result:** adder and shifter in near path need to be of length $2n$ then as per the above procedure, pipeline rate of 1 addition per $(\log n)$ time units can be achieved.

**III. Work in Progress**

We are currently developing a VLSI representation of our redundant data format algorithm, which will be ready by the time for final paper submissions. Having the VLSI representation will allow us to make comparisons of our method to the current IEEE standard as well as other proposed methods [2-8]. We will make the necessary timing, speedup, and other measurements to fully examine the algorithm. We will also be including figures and schematics of our VLSI implementation to further explain how our algorithm works.
References