Cairo University
Electronics and Communications Engineering

Computer Arithmetic, Lecture 11: Learning how to add two numbers

Hossam A. H. Fahmy

© 2009

Back to addition

Do you remember these decimal examples:

$$\begin{array}{rll} & 1.324 & \times 10^5 \\ + & 1.576 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 1.324 & \times 10^5 \\ + & 0.01576 & \times 10^5 \\ \hline & 1.33976 & \times 10^5 \\ \approx & 1.340 & \times 10^5 \end{array} \right.$$

$$\begin{array}{rll} & 9.853 & \times 10^7 \\ + & 1.466 & \times 10^6 \end{array} \left\{ \begin{array}{rll} & 9.853 & \times 10^7 \\ + & 0.1466 & \times 10^7 \\ \hline & 9.9996 & \times 10^7 \\ \approx & 1.000 & \times 10^8 \end{array} \right.$$

$$\begin{array}{rll} & 1.324 & \times 10^3 \\ - & 1.321 & \times 10^3 \end{array} \left\{ \begin{array}{rll} & 1.324 & \times 10^3 \\ + & 8.679 & \times 10^3 \\ \hline & 0.003 & \times 10^3 \\ = & 3.000 & \times 10^0 \end{array} \right.$$

One path algorithm for the IEEE binary standard

We need to perform the following steps for the significand of the result:

1. Find the exponent difference and decide on the smaller number.
2. Shift the smaller number to the *right* by the difference.
3. Add or subtract the two numbers depending on the *effective* operation.
4. In the case of an effective subtraction and when the exponents are equal, the "significand" of the result may become negative. If so, complement it.
5. In the case of an effective subtraction, find the location of the leading non-zero digit and shift the result to the *left* up to this location.
6. In the case of an addition with a carry overflow, normalize the result by a shift to the *right*.
7. Use the sticky digit, guard digit, and the *LSD* as well as the sign of the number and the rounding mode to decide on the appropriate rounding action then add this rounding digit.
8. If an overflow due to rounding occurs, renormalize the result.

What about the exponent and sign?

- The exponent of the result is that of the larger number adjusted according to the normalization.
  - For a subnormal result, the exponent is fixed to zero (saturated addition at the lower bound).
  - Depending on the rounding and traps, the case of overflow yields either the maximum exponent ($exp_{\mathbf{max}}$) or an indication of $\pm\infty$.
- The sign is that of the operand assumed to be the largest number.
  - If a complementation occurs the sign is flipped.
  - The rounding mode affects the sign: $\triangle(x-x) = +0$ while $\triangledown(x-x) = -0$ but $\triangle((-0)+(-0)) = -0$.

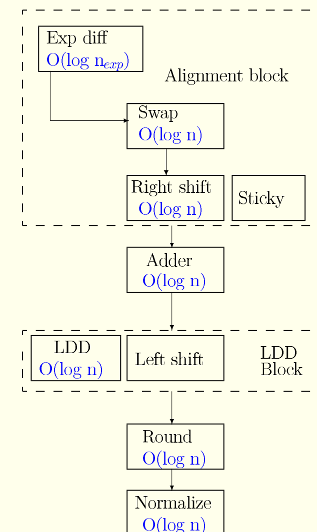
What are the main pieces?

This algorithm has a number of big parts:

- adders: exponent difference, significand addition, complementation, rounding;
- big shifters: alignment (right), leading digit normalization (left); and
- multiplexers: swap, short normalization shifts.

In addition to that, some logic blocks calculate the sticky digit, the rounding decision, the effective operation, invert the smaller operand for an effective subtraction, …

Simplified critical path analysis



Time delays in the blocks of an adder (one-path algorithm)

Do we really need all that?

- A large normalization to the left may occur only
  - for an effective subtraction, and
  - when the exponent difference is either zero or at most one.
- The need for complementation may occur only
  - for an effective subtraction, and
  - when the exponent difference is exactly zero.
- A large alignment shift occurs only when the exponent difference is large.

⇒ make two parallel paths, the "far" path and the "close" or "cancellation" path and run *speculatively* then choose the correct result at the end.

Close exponents may lead to cancellation

In the close-path, the exponent difference is either zero or one. Looking at the least significant two bits is enough to predict which is larger. *Why?*

We do not need to wait till the end of the addition to detect the leading non-zero digit, we can predict its approximate location while adding. *How?*

No rounding is needed if a cancellation occurs. *Why?*

Leading one prediction

Looking at two binary numbers:

$$\begin{array}{r} 1\;0\;1\;0\;0\;1\;1\;1\;\cdots \\ -\;1\;0\;1\;0\;0\;0\;0\;1\;\cdots \\ \hline 0\;0\;0\;0\;0\;1\;1\;0\;\cdots \end{array}$$

we see that the bits cancel when they are exactly the same.

Hence, form the bitwise *XOR* function and predict the location of the leading one.

Now, think about:

$$\begin{array}{r} 1\;0\;0\;0\;0\;1\;1\;1\;\cdots \\ -\;0\;1\;1\;1\;1\;0\;0\;1\;\cdots \\ \hline 0\;0\;0\;0\;1\;1\;1\;0\;\cdots \end{array}$$

For the general case, you need to consider all the bit patterns that might lead to a cancellation and detect them then shift accordingly.

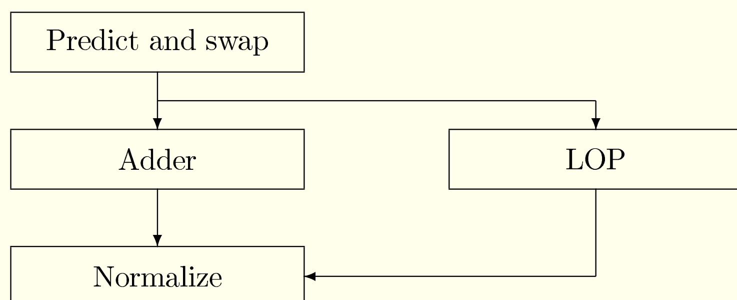
Rounding in the close path

- If the exponent difference is zero then the guard and sticky digits are zero. (Regardless of any cancellations.)
- If the exponent difference is one then the sticky digit is zero. If a left normalization shift is needed, the guard bit is shifted into the result.

In both cases, the result is exact and no rounding is needed.

$\Rightarrow$ Make the left shifting (in the case of an exponent difference of one) a condition to select the cancellation path and eliminate the rounding logic completely from that path.

Simplified close path



A simple view of the close path.

The large exponent difference

- We need to calculate the absolute difference of the exponents and shift the smaller number accordingly.
- Either add or subtract depending on the effective operation.
- Can we do the rounding decision in parallel to the adder since the guard and sticky digits are ready (*before* normalization)?

The issue of integrated rounding

The result of the adder is:

before normalization	(a)	N'	L'	G'	R'	S'
after normalization	(b)	N	L	G	S	
after rounding	(c)	N	L			

We want to decide the rounding based on (a) to get a correct result (c) as if it is determined using (b).

Normalization in the far-path

With an exponent difference larger than one we get the following possibilities for the final normalization:

Subtraction

No shift Simple rounding.

Shift left by one Must adjust the rounding. *Is this true?*

Addition

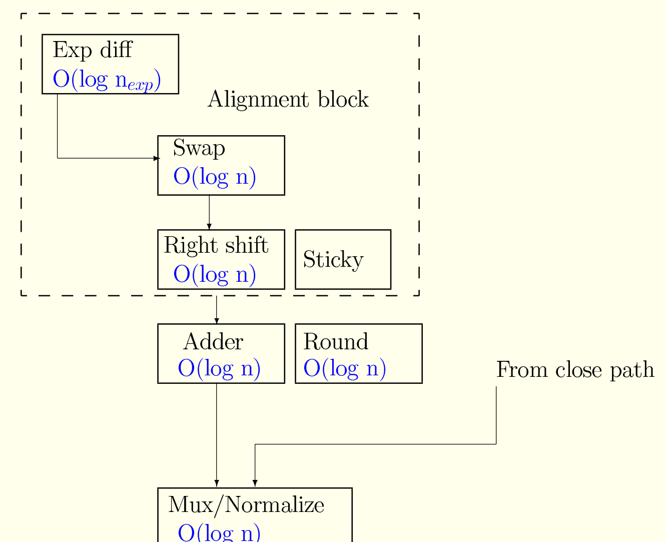
No shift Simple rounding.

Shift right by one Must adjust the rounding. *How?*

A compound adder

- The adder must provide both the *sum* and the $sum + 1$ and the rounding logic picks the correct result.
- With careful analysis, we deduce that for the case of RP and RM, the adder must also provide $sum + 2$. That leads to an extra row of half adders.

Far path



Time delays in the far-path of an adder (two-path algorithm)

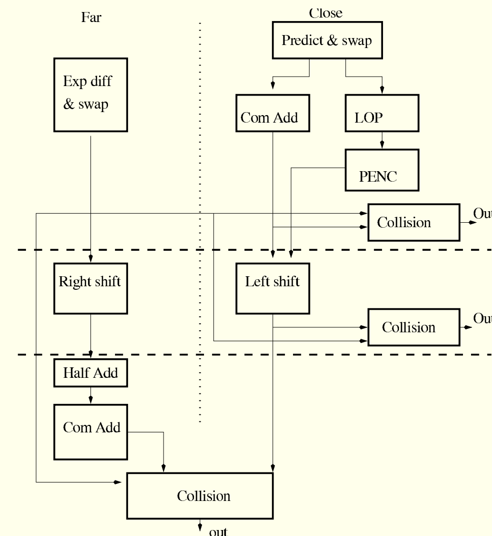
Variable latency adder

Yet another possible improvement uses a pipelined variable latency adder.

- If the close path is chosen and the output does not need to be left shifted, that output is then ready after just the first cycle.
- If it needs a left shift as indicated by the Leading One Predictor, LOP, and the priority encoder, PENC, it is available after the second cycle.
- Otherwise, it takes three cycles to finish.

A collision detection circuit prevents two outputs from getting out of the pipelined adder on the data bus at the same time.

Block diagram of the variable latency adder



The variable latency adder.

But, isn't variable latency bad?

There are a number of issues that limit the value of the variable latency adder:

1. How many collisions occur and the best way to avoid them?
2. The difficulties in current high performance processors to schedule a variable latency instruction.
3. How frequent are the results that finish earlier?

If, in a certain system, these are answered favorably then the variable latency adder leads to a lower average delay time.

Real adders

To sum it all up, the real adders used in your computers may use some of the previous optimizations but they must also

1. correctly get the sign and exponent,
2. handle exceptional inputs and outputs ($\pm\infty$, ± 0, NaN, and subnormals), and
3. generate the correct flags.

The adders usually also implement both the single and double precisions using the same unit.