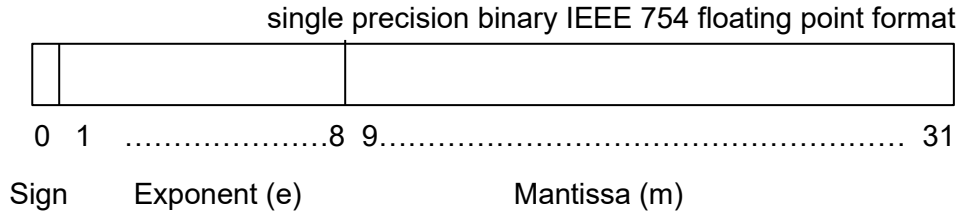
Class Notes 31/8/23 and 1/9/23 - On Vector Units and Pipelining Examples

I. Addition of two floating point numbers

Consider how addition of floating point numbers is done: firstly the mantissas of the two numbers to be added are adjusted so that the exponents are aligned, secondly, the mantissas are added, lastly the added number is 'normalized'. We will look at these three steps with the help of an example addition in single-precision IEEE-754 floating point numbers.

Recall that in single-precision IEEE-754 floating point scheme 32 bits are reserved for representation:



The number n that approximates the floating point number being represented is:

n = $\pm(1.\ m_1 m_2 .. m_{23})_2 \times 2^{(e_1 e_2 .. e_8)_2 - 127}$

Let us add 52.21875 and 2.75.

52.21875 is $(110100.00111)_2$ $(110100)_2 = 52$ and $(0.00111)_2 = 1/2^3 + 1/2^4 + 1/2^5$

2.75 is $(10.11)_2$ $(10)_2 = 2$ and $(0.11)_2 = 1/2 + 1/2^2$

Step 1: Adjust the mantissas such that there is only one digit to the left of the decimal point.

$(110100.00111)_2$ becomes $(1.1010000111)_2 \times 2^5$ and $(10.11)_2$ becomes $(1.011)_2 \times 2^1$.

Normalized mantissa is $(.1010000111)_2$ and $(.011)_2$ respectively. These bits are stored from index 9 to 31 above with 0s added as suffix. Exponent needs to be adjusted as per the formula $2^{(e_1 e_2 .. e_8)_2 - 127}$. So, the exponent bits for 52.21875, e – 127 = 5 => e = 132 = $(10000100)_2$. These bits are stored from index 1 to 8. Sign bit is 0.

Writing the IEEE-754 single-precision binary representation for 52.21875, we get 0 1000 0000 0110 0000 0000 0000 0000 000 (steps described in previous paragraph)

So, we have the following two numbers to be added:

0 1000 0000 0110 0000 0000 0000 0000 000 (0.25)

0 1000 0100 1010 0001 1100 0000 0000 000 (52.21875)

The highlighted exponent values in yellow are to be aligned (i.e. made equal) before adding. We usually take the lower-valued mantissa and add to it to make it equal to higher-value mantissa. In this example we need to add a number to 1000 0000 such that the sum is 1000 0100. Note that shifting the mantissa left by one bit decreases the exponent by 1 and right-shifting the mantissa increases the exponent by 1.

We find that we need to add $(0100)_2$ to align the radix. This means that we want to increase the exponent of 0.25's representation, $(1.011)_2 \times 2^1$, from 1 to 5 so that we get $(y.wza)_2 \times 2^5$.

As noted earlier, we right-shift the mantissa to increase the exponent. Also note that there is a hidden bit of 1 to the left of the decimal point as we are dealing with normalized numbers.

So,

Right-shifting $(.011)_2$ once we get 1011 0000 0000 0000 0000 000 (the hidden bit has appeared here as msb)

Second right-shift, we get 0101 1000 0000 0000 0000 000

Third right-shift, we get 0010 1100 0000 0000 0000 000

Fourth right-shift, we get 0001 0110 0000 0000 0000 000.

The mantissas are now adjusted.

Step 2: Add the mantissas (factor in the hidden bit as well)

	Sign(1)	Exponent(8)	hidden(1)	Mantissa (23)
	0	1000 0100	0.	0001 0110 0000 0000 0000 000
+				
	0	1000 0100	1.	1010 0001 1100 0000 0000 000
=	0	1000 0100	1.	1011 0111 1100 0000 0000 000

(here 0 appears immediately to the left of the decimal point because we right-shifted the bits of 0.25 to increase the exponent)

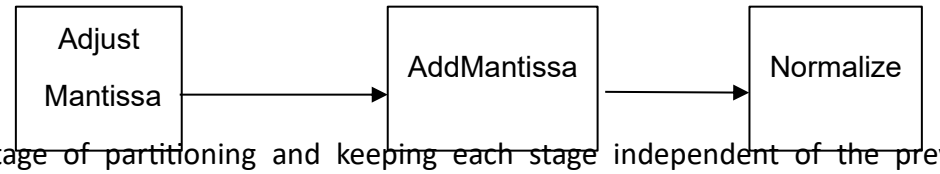
Step3: Normalize the exponents (get hidden bit to be a 1. They are already one in this example.)

Answer: 0 1000 0100 1011 0111 1100 0000 0000 000

II. Vector pipelines

Modern computer architectures have separate functional units (hardware) for performing floating point arithmetic. E.g. vector hardware. Typically, this hardware design is partitioned into

multiple “pipeline” stages. We can imagine 3 stages (step 1 to step 3) for performing this arithmetic as shown below:



The advantage of partitioning and keeping each stage independent of the previous (read: pipelining) is that we can have better throughput when multiple arithmetic operations need to be performed:

E.g., suppose we want to do:

$x1 + y1 = z1$

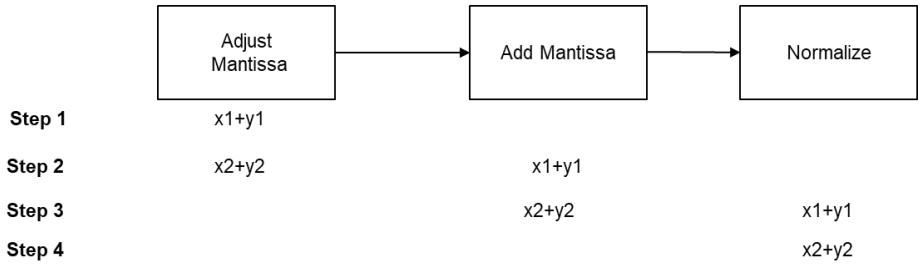
$x2 + y2 = z2$

Then we actually need to do:

$AdjustMantissa_{x1+y1}$, $AddMantissa_{x1+y1}$, $Normalize_{x1+y1}$

$AdjustMantissa_{x2+y2}$, $AddMantissa_{x2+y2}$, $Normalize_{x2+y2}$

These six operations can be scheduled on the 3 hardware sub-units as shown below:



Notice that in 4 steps we can perform two floating point additions. While AddMantissa is operating on x1+y1, AdjustMantissa unit is idle and hence, can operate on x2+y2 (during step 2 in the Figure). If the hardware unit was not pipelined, then we would have required 6 steps (3 each) to perform two additions. So, pipelining improves throughput in general.

Current Intel processor architecture has AVX512 units capable of supporting arithmetic on 512-bit wide registers (these are called vector registers). Now you can pack 16 float numbers (each occupying 4 bytes) into one vector register. With Fused-multiply-add units capable of delivering one floating point add and one multiplication operation per cycle (per cycle when there are a large number of such operations w.r.t the number of pipeline stages), we can get great throughput with “vectorizing” multiply-and-add operations such as below in a program:

```
C[i] = C[i] + A[i]*B[i]
```

III. Example on loop unrolling and vectorization from reference textbook:

Computing dot product:

```
sum = 0.0;
  for ( i = 0; i < n; i++ )
       sum = sum + a[i]*b[i];
```

after unrolling 4 times:

```
/* do inner product in blocks of length 4 */
      sum0 = sum1 = sum2 = sum3 = 0.0;
       for (i = 0; i < n; i += 4 ){
        sum0 = sum0 + a[i ]*b[i ];
        sum1 = sum1 + a[i+1]*b[i+1];
        sum2 = sum2 + a[i+2]*b[i+2];
        sum3 = sum3 + a[i+3]*b[i+3];
       }

      sum = sum0 + sum1 + sum2 + sum3;

     /* take care of fractional block:
         4*(n/4) = 4*floor(n/4) */
      for ( i = 4*(n/4); i < n; i++ )
           sum = sum + a[i]*b[i];
```

The former code requires 3n steps when the operations are to be performed on a 3-stage pipeline. The latter unrolled code reduces this by a factor of 3.

In the former code, when sum is to be computed in the next iteration, we need the latest value of sum. For this, result of a[i]*b[i] computed in the previous iteration must be available. This means that we have to wait for 3 steps until the pipeline produces the result of a[i] * b[i]. So, the next iteration cannot begin until the previous has completed.

In the latter code, because each of the four statements are independent, when the next iteration of the first `for` loop begins, latest value of sum0 is already computed in the previous iteration. So e.g., `sum0 = sum0 + a[i ]*b[i ]; for i = 4,` can immediately begin in the next step after `sum3 = sum3 + a[i+3]*b[i+3];` Similarly, immediately after `sum0 = sum0 + a[i ]*b[i ];` when `i = 4`, `sum1 = sum1 + a[i+1]*b[i+1];` can begin because `sum1` in the previous step has been computed. So, the first for loop finishes in O(n) reducing the number of steps by a factor of three.