Section 2 / What Are Floating Point Numbers?

Before we introduce floating point instructions in the AARCH64 ISA, it is worth going over exactly what a floating point value is. Integers are easy. They're just powers of two summed together with a single bit at one end determining the sign (if the integer is signed).

But what are floating numbers?

Key Point

Floating point values are approximations.

Sometimes they are spot on. Most of the time, they are close.

Floating Point Value Explorer

Here is source code to a program for you that takes floating point values (both single and double precision) apart.

Here are some examples:

```
Must supply a floating point value as a command line argument.
```

The above is what happens when you do not provide a value to examine.

Next, let's see what is output from the value of 1.

```
Component           Double                   Float                    Comment
Value:              1                        1                        Delta(F - D): 0
Sign:               0                        0
Exponent (hex):     3ff                      7f
De-biased (dec):    0                        0
Fraction (hex):     0                        0
Halves:             0                        0
Quarters:           0                        0
Eighths:            0                        0
Sixteenths:         0                        0
Thirty seconds:     0                        0
Full fraction:      0                        0
Equation:           1 x 2^0                  1 x 2^0
```

On the line marked "Value" you can see the values represented as double precision and as single precision. Under "Comment" you can see that there is no difference between the double and the single precision numbers. Remember the key thing about floating point numbers: they are approximations. Sometimes, as in the case of whole numbers like 1, the approximation is exact. When there is a difference, the difference will be small and printed in the Comment column.

The Sign field is 0. This indicates that the whole floating point value is positive. There are no other sign values including in the exponent. However, exponents can be negative… this is explained next.

First, notice that the double precision exponent is 11 bits wide while the single precision exponent is only 8 bits wide. Next, notice the values… 1023 and 127 respectively. The value of 1 is 1 raised to the power of 0 base 2. So why 1023 or 127?

There is no sign bit for the exponent yet the exponent must support negative numbers. It does this by incorporating an offset of 1023 and 127 respectively (both representing 0). Anything above 1023 and 127 are positive exponents. Anything below these values are negative exponents.

The De-biased line are the values of the exponent with their bias removed. Notice they work out to 0. So, the value of 1 is represented by 1 raised to the power of 0.

The Fraction has a value of zero. Where's the 1 that we've been talking about get stored? It isn't. A value of 1 is always assumed to be the only value in front of the decimal place in a `float` or `double`. Every floating point value is 1 plus a fraction all raised to some power of 2.

We thought we'd highlight a few of the bits in the fractional part of a floating point number. These can be illuminating when the value being shown is in the range of $-2 < x < 2$. Notice the the values of -2 and 2 are outside this range. In other words, showing the first few bits of the fraction are illuminating when the exponent works out to 0.

- Halves - There are no halves in the value of 1.
- Quarters - There are no quarters in the value of 1.
- Eighths - There are no eighths in the value of 1.
- Sixteenths - There are no sixteenths in the value of 1.
- Thirty Seconds - There are no thirty seconds in the value of 1.

Of course, there are more fractional values to `float` and `doubles` but listing them all wouldn't be a fun tasks and we're all about fun. :)

Finally, the Equation line rebuilds the floating point value in its actual "scientific" notation. The value of 1 is a 1 raised to the zeroth power of 2.

How about a value of 1.5?

```
Component          Double                Float                 Comment
Value:             1.5                   1.5                   Delta(F - D): 0
Sign:              0                     0
Exponent (hex):    3ff                   7f
De-biased (dec):   0                     0
Fraction (hex):    8000000000000         400000
```

```
Halves:             1                        1
Quarters:           0                        0
Eighths:            0                        0
Sixteenths:         0                        0
Thirty seconds:     0                        0
Full fraction:      0.5                      0.5
Equation:           1.5 x 2^0                1.5 x 2^0
```

The only difference is that there is a bit turned on in the fraction. It is the most significant bit… there is a half in one and a half.

How about 1.875?

```
Component           Double                   Float                    Comment
Value:              1.875                    1.875                    Delta(F - D): 0
Sign:               0                        0
Exponent (hex):     3ff                      7f
De-biased (dec):    0                        0
Fraction (hex):     e000000000000            700000
Halves:             1                        1
Quarters:           1                        1
Eighths:            1                        1
Sixteenths:         0                        0
Thirty seconds:     0                        0
Full fraction:      0.875                    0.875
Equation:           1.875 x 2^0              1.875 x 2^0
```

How about 8.5?

This is the first time we are looking at a value which increases the (de-biased) exponent to non-zero. Things get a little more complicated. Now, there isn't an obvious mapping of the fraction bits to the final number they represent. This is the impact of the non-zero exponent.

```
Component           Double                   Float                    Comment
Value:              8.5                      8.5                      Delta(F - D): 0
Sign:               0                        0
Exponent (hex):     402                      82
De-biased (dec):    3                        3
Fraction (hex):     1000000000000            80000
Halves:             0                        0
Quarters:           0                        0
Eighths:            0                        0
Sixteenths:         1                        1
Thirty seconds:     0                        0
Full fraction:      0.0625                   0.0625
Equation:           1.0625 x 2^3             1.0625 x 2^3
```

Even though there is a half in eight and a half, the Halves bit is 0. What is 8?

Eight is a 2 raised to the power of 3. In other words, the bit for the half in 8.5 is shifted to the right by three bits. Confirm this by looking at the Sixteenths. *There's our bit!*

Turn your attention to the Equation. 1.0625 multiplied by 8 is 8.5. Cool huh?

How about something harder? Like 8.51 - just a teensy bit different from the previous example.

```
Component           Double                   Float                    Comment
Value:              8.51                     8.510000229              Delta(F - D): 2.28881
Sign:               0                        0
Exponent (hex):     402                      82
De-biased (dec):    3                        3
Fraction (hex):     1051eb851eb85            828f6
Halves:             0                        0
Quarters:           0                        0
Eighths:            0                        0
Sixteenths:         1                        1
Thirty seconds:     0                        0
Full fraction:      0.06375                  0.06375002861
Equation:           1.06375 x 2^3            1.0637500286 x 2^3
```

For the first time we're seeing that 8.51 cannot be perfectly represented by `float`. `double` gets it right. The difference between the `double` and `float` is the very small number shown on the first line of output.

When a Number is Not a Number and How About Infinity?

`NaN` is an actual value. It means `not a number`.

Here is the source code to another program we have written that explores both `NaN` and `Inf`.

Let's examine `NaN` which is produced when you do naughty things like take the square root of a negative number.

```
Enter a number (-100 causes divide by 0, -200 causes sqrt(-1): -200
Using sqrt(-1)).
sign: 0
exp:  ff debiased: 128
frac: 0400000
NaN:  1
Inf:  0
```

`Nan` is true (for `float`) when its exponent is 0xFF and fraction is not zero.

You'll never get a `float` that is 2 raised to the power of 128 because that value is reserved for `NaN` and `Inf`.

How about `Inf`?

```
Enter a number (-100 causes divide by 0, -200 causes sqrt(-1): -100
Dividing by zero.
sign: 1
exp:  ff debiased: 128
frac: 0000000
NaN:  0
Inf:  1
```

Once again, notice the out-of-bounds value for the exponent: 0xFF. Secondly, the fraction is fully zero. The sign bit specifies negative or positive infinity.

Testing for Naughty Values

Thankfully, there exists two functions that will do the inspection for you, looking for `Nan` and `Inf`.

- `isnan(floating point value)` and
- `isinf(floating point value)`

Both of these functions work with `double` and `float`.

Once a variable goes `NaN` or `Inf`, all subsequent operations will remain `NaN` or `Inf` until the variable is reset to a valid number. That is, 1 + `Inf` is `Inf`, for example.