

Re: Assembly Language – Mathematics WITHOUT maths coprocessor

Source: <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-10/msg00304.html>

- *From:* "Richard Cooper" <spamandviruses@xxxxxx>
 - *Date:* Sat, 15 Oct 2005 08:57:01 GMT
-

On Sat, 15 Oct 2005 02:19:23 -0400, `ra\b` <al@xxx> wrote:

```
this is false 40 bits hold
[log10(240)+1]=13 decimal digits
```

Nope, I'm right.

```
no you are wrong [log10(x)+1] where []="integer part" is how many
digits is the number x. For x=240 the digits are 13 =[log10(240)+1]
```

But "how many digits is the number 2^x" isn't the same as "how many digits will x bits hold"...

Look again at what I said:

```
240 = 1099511627776
1012 = 1000000000000
1013 = 10000000000000
```

So, 13 decimal digits are capable of holding more possible values than 40 bits can hold, however, 40 bits can hold slightly more than what 12 decimal places can hold. So I think my answer of 12.04119983 is a little more correct, as 40 bits can hold as many values as 12 decimal digits, plus just a little bit more.

Maybe it's time for another example:

Re: Assembly Language – Mathematics WITHOUT maths coprocessor

That's what I said to do, except that you've made twice as much work out of it.

What you're doing is, by ignoring the "0." at the beginning of `s`, you're multiplying the fractional part by $10^{(\text{strlen}(s)-2)}$. Then, when you multiply by $2^{\text{precision}}$, you shift it left by `precision`. Then you divide it by $10^{(\text{strlen}(s)-2)}$, which is the same number that you effectively multiplied it by earlier.

So we've got what you're doing:

1. Separate number into integer and fraction parts.
2. Convert integer part to binary.
3. Shift integer part left by `precision`.
4. Convert fraction part to binary as if it were an integer.
5. Shift fraction part left by `precision`.
6. Divide fraction part by what step 4 effectively multiplied it by.
7. Add or subtract together the integer and fraction parts.

Then there's what I said to do:

1. Convert entire number to binary as if it were an integer.
2. Shift entire number left by `precision`.
3. Divide number by what step 1 effectively multiplied it by.

Steps 2 and 3 of what you're doing are the same process as steps 4 and 5, so if you just skip step 1 then you get to do them both at the same time, and as an added bonus, you no longer need step 7 to combine the two numbers back together again. So you effectively cut it down to just steps 4, 5, and 6, which make it just like my steps 1, 2 and 3.

You may, however, just want to stick with what you've got, as it may have optimizational benefits depending on the number of bits in the numbers. I'm still not sure what you're doing, so I can't say which way is faster in your particular case, but it depends on whether the divide in step 6 of your method is significantly easier than the divide in step 3 of my method.

.