Three algorithms to convert fractional parts from binary to decimal[1]

P. Montuschi, January 15, 2016 (version 2.0)

IMPORTANT NOTE: all algorithms below assume that the part to be converted is fractional only, i.e. that the integer part has already been converted.

Observation

We observe that

$2^{-1} = 0.1_2$	=	0.5_{10}	(1 binary fract. digit → 1 decimal frac. digit)
$2^{-2} = 0.01_2$	=	0.25_{10}	(2 binary fract. digits → 2 decimal fract. digits)
$2^{-3} = 0.001_2$	=	0.125_{10}	(3 binary fract. digits → 3 decimal fract. digits)
$2^{-4} = 0.0001_2$	=	0.0625_{10}	(4 binary fract. digits → 4 decimal fract. digits)
$2^{-5} = 0.00001_2$	=	0.03125_{10}	(5 binary fract. digits → 5 decimal fract. digits)

and so on.

It can be demonstrated that if a binary fractional part requires *n* fractional digits, then its converted decimal fractional part requires n fractional digits as well (the converse is not true).

In order to better present the algorithms, let's pick up a numerical example, where we focus on the fractional part only. According to the definition of weighted number systems, we have

$0.11001_2=1*2^{-1}+1*2^{-2}+0*2^{-3}+0*2^{-4}+1*2^{-5}=(0.5+0.25+0.03125)_{10}=0.78125_{10}$

First algorithm (general, sufficiently straightforward, although some attention is requested while going to its implementation)

GOAL: *The goal is to extract one by one the fractional bits, in a "similar way" as it is done for the integer part.*

ADVANTAGE: *In theory it is rather genera, but its implementation could be tough for values on several fractional bits*

DRAWBACK: *It requires both a careful attention to the representation at each step of the computations, as well as a full development of its implementation (no previous implementations can be directly used)*

The Theory (better to have a look even if you are not so passionate)

Given $B = 0.11001_2 = 0.78125_{10}$ what is the operation which is necessary to "extract" the first decimal fractional digit?

[1] Please send a note to paolo.montuschi@polito.it if you find any typos/errors. Thanks.

The answer is straightforward: it is sufficient to multiply B by 10_{10} and then to isolate the "newly produced" integer digit, i.e.

FIRST DECIMAL FRACTIONAL DIGIT = TRUNC (B * 10_{10}) = 7_{10}

We can observe that the part of (B * 10_{10}) which is being truncated equals 0.8125_{10}. To extract the second decimal fractional digit is necessary to:

1. Take the truncated (fractional) part coming from the previous operation
2. Multiply it by 10_{10}
3. Truncate the result; the "now-integer" part is the next digit, while the truncated part will be used by the next iteration

So, basically, steps 1-3 "solve" the conversion problem.

How many times should steps 1-3 be executed? The answer is, again, straightforward; as it has been shown before, *n* fractional binary digits require *n* fractional decimal digits as well; therefore it is necessary to execute steps 1-3 *n* times.

<u>The Implementation (this is important and a bit more "tough")</u>

Let's explain the implementation by means of the numerical example. Let us assume that the fractional part is stored in AL. It is mandatory that the integer part is zero, i.e. that the fractional part only is stored in AL, where the integer bits are all zeroes. Let us assume to have CL storing $10_{10}=1010_2$

AL = 000 . 11001
integer . fractional

The contents of AL is now ready for step 2 to run.

Step 2: multiply AL by CL; the result is AX = 00000000111.11010_2

Please observe that the position of the fractional point is the same as above, i.e. the result AX=AL*CL has 5 fractional binary digits.

Step 3: Truncate AX implies the need to separate the two parts:

- the left-to the-point integer part 00000000111_2, which corresponds to the first extracted digit, i.e. $0111_2=7_{10}$
- the right-to the-point fractional part $.11010_2$ to be used for next digit extractions

In order to get ready for the next iteration, it is necessary to switch from AX back to AL and to "clear" the integer part of AX. This can be done by dropping away AH and the simply masking the non-fractional bits of AL.

Step 1 (next iteration)

- From AX = 00000000111.11010_2 drop AH and stay on AL only; AL = 111.11010_2
- Mask (i.e. set to zero) the integer bits of AL; AL AND 000.11111_2 = 000.11010_2

Now we are ready for another Step 2.

Step 2 (next iteration): multiply AL by CL; the result is AX = 00000001000.00100_2

Step 3 (next iteration): truncate AX:

- the left-to the-point integer part 00000001000_2, which corresponds to the first extracted digit, i.e. 10002=810
- the right-to the-point fractional part $.00100_2$ to be used for next digit extractions
- now AL becomes 000.00100_2

The conversion then proceeds with 3 more iterations of Steps 1-3 as above.

This method is rather general and can, in theory, be used for any number of fractional bits. However, if the fractional binary part is stored into AL, then it is clear that this method works up to 8 fractional binary digits. For larger number of bits it would be necessary to use wider registers, i.e. with more bits.

Second algorithm (brute force attack; easily working for up to 8 fractional bits; not recommended for more than 8 fractional bits)

GOAL: *The goal is to do the conversion with the help of a suitably initialized lookup table*

ADVANTAGE: *As it is just a matter of accessing a predefined table, the necessary effort is almost null; a particular care should be taken to properly initialize the table, but no other operations than table access are requested*

DRAWBACK: *The number of entries of the table increases exponentially with the number n of fractional bits to be converted, where the size of each entry increases linearly with n. This results in a feasibility of this method that is possible when the number n is sufficiently small, i.e. up to 8*

The Theory (you can skip if you are not interested)

There is not too much theory for this method. Instead of computing or extracting the single decimal digits, this algorithm uses a table where all possible cases of conversions, i.e. 2^n have been stored already in the converted form.

The Implementation (this is the really important part)

For sake of simplicity, let us focus on the problem of converting 3 binary fractional bits into the corresponding 3 decimal fractional digits. There are 8 possible cases with 3 bits and each one corresponds to the conversion of 3 decimal fractional digits.

entry #	fractional bits	dec."integer"	1st dec. digit	2nd dec. digit	3rd dec. digit
0	0.000	0.	0	0	0
1	0.001	0.	1	2	5
2	0.010	0.	2	5	0
3	0.011	0.	3	7	5
4	0.100	0.	5	0	0
5	0.101	0.	6	2	5
6	0.110	0.	7	5	0
7	0.111	0.	8	7	5

If, for example, we would like to convert 0.101_2, we must take entry #5 (i.e. the sixth row) where we find 0.625_{10}, which corresponds to what we were looking for. Please observe that there is a direct correspondence between the number we need to convert and the number of the entry. The number of the entry can be simply achieved by "neglecting" the fractional point of the binary representation: $0.101_2 \rightarrow 101_2 = 5_{10} \rightarrow$ entry #5.

Please consider that, when switching to an assembly implementation:

- It is advised to store in the table, not the values of the digits, but their Ascii code, in order to simplify a later printing of the results of the conversion.
- As the Ascii code of a single character, uses one byte, then each of the 2^n lines has to store n elements, each one of one byte; the overall size of the table is $n*2^n$ bytes.
- As in the assembly language arrays are defined and accessed (mostly) by bytes, it is necessary to keep in mind that each entry in NOT made by one byte but by n bytes.

By looking at the previous table, here below it is proposed the corresponding implementation of the first cells (as a matter of space) of the array organized in bytes; please observe that in this case we have stored the first decimal fractional digit in the first byte of the entry, and so on; other choices are also possible.

Entry #	Byte #	Contents (ascii code)
0	0	48 (equiv. to 0)
	1	48 (equiv. to 0)
	2	48 (equiv. to 0)
1	3	49 (equiv. to 1)
	4	50 (equiv. to 2)
	5	53 (equiv. to 5)
2	6	50 (equiv. to 2)
	7	53 (equiv. to 5)
	8	48 (equiv. to 0)

Third algorithm (transformation and then integer conversion; theory is not so straightforward, but implementation is easy; easily working for up to 6 fractional bits)

GOAL: *The goal is to transform the fractional part into an integer number. Once the integer number has been obtained, then the conversion from integer binary to integer decimal can be run.*

ADVANTAGE: *Once the transformation has been done, the well-known software and algorithms for binary integer to decimal integer can be used (provided that the digits that are obtained are placed at the right of the fractional point)*

DRAWBACK: *In theory it could work for any number of fractional bits, but in the practice, it works better for up to 6 fractional bits.*

<u>The Theory (you can skip if you are not interested)</u>

To transform B = $0.11001_2 = 0.78125_{10}$ into a decimal integer number requires it being multiplied by C= $(10^5)_{10}=100000_{10}$.

8086 architecture does not offer support for binary fractional numbers, that are therefore B is represented in fixed point assuming that the point has been placed somewhere. In the case of the example, $B=0.11001_2$ is stored in a byte "as an integer" as $A=00011001_2$; in other words, in the byte it is stored the value $0.11001*2^5$=A and therefore

B (which is the exact fractional value) = A (which is the integer stored value) * 2^{-5}

This can be double checked by observing that $A=00011001_2=25_{10}$ and therefore

B= $25_{10}/32_{10}=0.78125_{10}$ (which is the correct value for B). Let us call $D=2^5$

At this point, the transformation algorithm is straightforward, as it only requires to

compute (A*C)/D

The result will be an integer number corresponding to the full decimal fractional part.

With reference to our previous example, as $A=25_{10}$, $C=10^5$, $D=2^5{}_{10}=32_{10}$ we have:

$(25*100000)_{10}$ / $32_{10}=78125_{10}$ which corresponds to the initial fractional part "B", now became an integer value. From 78125, the next steps are straightforward.

Please consider that although the last 3 lines have considered values the decimal visualization of the operations, the matter of fact is that, independently of the number system used (and therefore this includes the binary system as well), the computation (A*C)/D provides us with an integer value which corresponds to the string of digits to be written right to the point in the decimal representation (i.e. ."78125").

A direct implementation in the 8086 assembly language, involves one multiplication and one division. Let's look at the operands sizes.

A = 25 requires 5 bits, and therefore A can be certainly stored on 8 bits;

C = 10^5 requires at least 17 bits and therefore 16 bits are not sufficient

D = 32 requires 6 bits and therefore well fits into 8 bits

The problem is that 8086 does not offer support for a 8 bits * 17 bits multiplication; even if we extend the second operand to 32 bits, still 8086 it does not offer support also for 8 bits * 32 bits multiplication. This observation could lead to a first (wrong) conclusion could be that this method cannot be used for 5 fractional bits.

Let's see if we can find some workaround. Let us rearrange the order of execution of the multiplication and division.

(A*C)/D = A*(C/D), i.e. we compute first C/D and then multiply by A.

This could be a good idea if C/D produces an exact quotient, otherwise it could be a mess to manage the fractional part of C/D. We show that C/D always produces an exact quotient.

By definition, $C=10^5$, $D=2^5$ and therefore $(C/D) = (10/2)^5 = 5^5$. In general, for *n* fractional digits we have $(C/D) = 5^n$.

How much difficult is to compute C/D in 8086 assembly language? It is not difficult at all, and does not even require a division, as, once *n* is known, 5^n is a constant. Let us denote it as $K=5^n$

In our specific case, $5^5=3125$ and then our computations become

$(A*C)/D = A*(C/D) = A*K = A*3125$

A is on 8 bits and K=3125 is on 16 bits; in order to compute A*K it is necessary to implement a 16 by 16 bits multiplication (extending A to 16 bits). The result will be our initial fractional part now converted into integer number ready to be converted by using the classical well-known algorithms.

Back to our numerical example, A*K = 25*3125 = 78125 which is the transformed into integer initial fractional part.

How many bits out of the 32 coming out from the multiplication A*K have to be considered?

As A*K is the transformation into integer of the initial binary fractional part on 5 bits, then A*K will be an integer decimal number on 5 digits.

We can observe that, in theory, the maximum integer decimal number on 5 digits is 99999, which requires 17 bits to be represented in binary and therefore, 17 is a "sufficient" response to our question.

A tighter bound, i.e. a necessary and sufficient bound, can be observed that the maximum 5 bits fractional number is $0.11111_2 = (1-2^{-5})_{10}$, thus corresponding to $A=(2^5*(1-2^{-5}))_{10}=31_{10}$. Therefore the maximum value for

$A*K$ (on 5 digits) $= (31*3125)_{10} = 96875_{10}$ which requires 17 bits to be represented in binary (actually, observe that at most the fractional part B is 0.11111_2 which corresponds to 0.96875_{10}).

In general, the formula for the tighter (necessary and sufficient) bound is ceil $(\log_2[(2^n-1)*5^n])$

The Implementation (this is the really important part)

Given a binary fractional number B, represented on *n* bits in fixed point into a register, i.e. as $A=B*2^n$, its transformation into integer value ready for binary to decimal conversion requires the computation of A*K with $K=(5^n)_{10}$ on a suitable number of bits. The result is computed on a number of bits, depending on the value of *n*. The table below reports the most common cases. For sake of simplicity it is assumed that B<1 and is stored in fixed point in AX and that the value of the constant $K=5^n$ is stored into CX.

n	AX bits (fixed point)	K value (stored in CX) = 5^n	Bits of (AX*CX) to be considered (i.e. the rightmost ??? bits), i.e. ceil ($\log_2[(2^n-1)*5^n]$)
1	0000…000 **. x**	5	3
2	0000…000 **. xx**	25	7
3	0000…000 **. xxx**	125	10
4	0000…000 **. xxxx**	625	14
5	0000…000 **. xxxxx**	3125	17
6	0000…000 **. xxxxxx**	15625	20
7	0000…000 **. xxxxxxx**	78125 (requires 17 bits)	24
8	0000…000 **. xxxxxxxx**	390625 (requires 19 bits)	27

For the example $B=0.11001_2$ corresponds to the representation into AX=0000000000000011001 , i.e. where the fractional point is between the sixth and fifth rightmost bits. Actually, the value stored in AX (without considering the fixed point) is A = $11011_2 = 25_{10}$. If we pick the row with *n*=5 in the table, we have that CX should be initialized to 3125 and then AX*CX should be computed. The result will be provided into DX:AX and the rightmost 17 bits will represent my "transformed-into-integer" value, ready to be bin-to-dec transformed.

Back to our numerical example, AX*CX = 25*3125 = 78125, which corresponds to what we were looking for.