Quickly Converting Decimal to Binary

Connor Baker

January 2017
Version 0.1a

Abstract

This paper briefly discusses ways to convert decimal numbers to their binary representation using several fast algorithms which exploit powers of two as a means of performing less arithmetic operations, with the caveat of using a look up table.

Contents

1 From Decimal to Binary

Given a number ABCDE in any base β, one can rewrite that number in what is called exponential form: that is, the sum of each digit multiplied by the base to a power. For our example ABCDE$_\beta$ (the notation means a number ABCDE in base β), we would write it as follows: A$\times\beta^4$+ B$\times\beta^3$+C$\times\beta^2$D$\times\beta^1$+E$\times\beta^0$. As an example, given the number 44675_{10}, we can rewrite it as $4\times10^4+4\times10^3+6\times10^2+7\times10^1+5\times10^0$.

In terms of changing the base of a number, the goal is to rewrite it as the sum of some numbers multiplied by the desired base to a power. Take for example the number 7_{10}. We know that we can rewrite it as 7×10^0, but how could we express this number in base two? Our goal is to end up with something that looks like $a_1\times2^n+a_2\times2^{n-1}+...+a_b\times2^0$. The question then becomes, how can we find those weights $\{a_1,a_2,...,a_b\}$ that make the sum seven? The answer is division.

In the table below, we perform the division necessary to calculate the binary representation of seven in base two. Each time we divide the decimal by two and get a remainder of one, we put a one in the rightmost place of our decimal representation. Likewise, if we have a remainder of zero, we append a zero. Once the value of the quotient is zero, we halt computation because we have found the binary representation in full.

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
7			
3	1	'1'	1
1	1	'1'	11
0	1	'1'	111

In the first row, we have the original decimal, an empty remainder since we have not begun to divide yet, and an empty binary representation. The second row has the whole number portion of the quotient, the remainder, and the beginnings of our binary representation. The third and fourth rows continue filling out binary representation. The fifth row is where we meet the halt condition, which is why the binary representation is unchanged.

So know that we know our binary representation is 111_2, we can say that $7_{10}=1\times2^2+1\times2^1+1\times2^0=111_2$.

2 Algorithms

2.1 A Simple Approach

This is the algorithm we used in Section 1 for converting 7_{10} to binary. To convert a decimal number to binary, we divide each resultant quotient by two repeatedly until we meet the halt condition (zero as a quotient), and then take the remainders at each step in reverse.

In terms of an analog to computer science, we can think of keeping track of the remainders as adding each new one to the top of a stack, and at the end, pulling from the top of the stack (which reads all entries in reverse). This means that the remainder of the last division operation is going to be the most significant bit in the binary representation.

Remainder	**Push to Stack**
0	'0'
1	'1'

Table 1: The 2^1 Algorithm

We should be dissatisfied with the speed of this algorithm. For smaller numbers, this algorithm is fine. But what if we needed to compute the binary representation of a larger number, like 37_{10} or 51_{10} (both examples we have done in class)?

Our goal then is to find a faster algorithm. The easiest way to start is by speculating. One thing that we could examine is what happens when we divide by a number that is larger than two – perhaps a power of two.

2.2 Exploiting Powers of Two

Given any number $n \in \mathbb{N}$, $n \textbf{ mod } 2$ will be either zero or one. With the algorithm we have above, we see that we account for both possibilities of the remainder. If we divide by a larger multiple of two, like four, we now have a different number of possible remainders.

Given any number $n \in \mathbb{N}$, $n \textbf{ mod } 4$ will be either zero, one, two, or three. As such, we have to take into account these possibilities. Interestingly enough, we must increase the number of bits that we push to the stack at once. The numbers zero and one can be represented in binary with a single bit. However, the numbers two and three require two bits. In general, if we divide by 2^n, we will have to take into account remainders for $[0, 2^n - 1]$, and the binary for each remainder must have n place values.

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^1 algorithm.

Remainder	**Push to Stack**
0	'00'
1	'01'
2	'10'
3	'11'

Table 2: The 2^2 Algorithm

Given any number $n \in \mathbb{N}$, n **mod** 8 will be in $[0, 7]$ (per the range of remainders of $[0, 2^n - 1]$). Again, we have to take into account these possibilities. Since we're dividing by 2^3, the binary for each remainder must have n place values.

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^2 algorithm.

Remainder	**Push to Stack**
0	'000'
1	'001'
2	'010'
3	'011'
4	'100'
5	'101'
6	'110'
7	'111'

Table 3: The 2^3 Algorithm

Given any number $n \in \mathbb{N}$, n **mod** 16 will be in $[0, 15]$ (per the range of remainders of $[0, 2^n - 1]$). Again, we have to take into account these possibilities. Since we're dividing by 2^4, the binary for each remainder must have n place values.

With this algorithm, given an arbitrarily large number, the number of iterations we perform is typically about half of the number we would perform using the 2^3 algorithm.

Remainder	**Push to Stack**
0	‘0000’
1	‘0001’
2	‘0010’
3	‘0011’
4	‘0100’
5	‘0101’
6	‘0110’
7	‘0111’
8	‘1000’
9	‘1001’
10	‘1010’
11	‘1011’
12	‘1100’
13	‘1101’
14	‘1110’
15	‘1111’

Table 4: The 2^4 Algorithm

3 Examples

In this section, we will be converting the numbers 37_{10}, 55_{10}, and 128_{10} to binary using each of the four algorithms covered in this paper to show the difference in speed.

3.1 The 2^1 Algorithm

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
37			
18	1	'1'	1
9	0	'0'	01
4	1	'1'	101
2	0	'0'	0101
1	0	'0'	00101
0	1	'1'	100101

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
55			
27	1	'1'	1
13	1	'1'	11
6	1	'1'	111
3	0	'0'	0111
1	1	'1'	10111
0	1	'1'	110111

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
128			
64	0	'0'	0
32	0	'0'	00
16	0	'0'	000
8	0	'0'	0000
4	0	'0'	00000
2	0	'0'	000000
1	0	'0'	0000000
0	1	'1'	10000000

3.2 The 2^2 Algorithm

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
37			
9	1	'01'	01
2	1	'01'	0101
0	2	'11'	100101

Value of Quotient	Remainder	Push to Stack	Binary Representation
55			
13	3	‘11’	11
3	1	‘01’	0111
0	3	‘11’	110111

Value of Quotient	Remainder	Push to Stack	Binary Representation
128			
32	0	‘00’	00
8	0	‘00’	0000
2	0	‘00’	000000
0	1	‘01’	10000000

3.3 The 2^3 Algorithm

Value of Quotient	Remainder	Push to Stack	Binary Representation
37			
4	5	‘101’	101
0	4	‘100’	100101

Value of Quotient	Remainder	Push to Stack	Binary Representation
55			
6	7	‘111’	111
0	6	‘110’	110111

Value of Quotient	Remainder	Push to Stack	Binary Representation
128			
16	0	‘000’	000
2	0	‘000’	000000
0	2	‘010’	010000000

We notice here that we have a leading zero. As it holds no value, it’s fine to discard from the final representation. This is a result of using a power that is larger than two, since we have more than a single bit.

3.4 The 2^4 Algorithm

Value of Quotient	Remainder	Push to Stack	Binary Representation
37			
2	5	‘0101’	0101
0	2	‘0010’	00100101

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
55			
3	7	‘0111’	0111
0	3	‘0011’	00110111

Value of Quotient	**Remainder**	**Push to Stack**	**Binary Representation**
128			
8	0	‘0000’	0000
0	8	‘1000’	10000000