PL/SQL 101 – DATATYPES – NUMBERS

Author: BluShadow
Last Updated: 9th July 2015

CONTENTS

1. INTRODUCTION

Amongst the common questions we get on the community, some relate to issues that are introduced by the developers own lack of understanding of Datatypes. This article is going to look specifically at the NUMBER datatype, for which we often get developers wanting to store numeric values on their database with 'leading zeros', and will then give some more in-depth knowledge about the NUMBER datatype.

2. HOW CAN I STORE NUMBERS WITH LEADING ZEROS?

Briefly, the answer to this is simple… **you don't**.
For example a numeric value of 123 is exactly the same numeric value as 00123…

```
SQL> select 00123 from dual;
     00123
----------
       123

1 row selected.
```

The issue that is really trying to be resolved isn't one of how data is stored on the database, but rather one of how the data should be displayed to a user; often being part of a requirement such as "The amount should be displayed showing 5 digits, with leading zeros where applicable". If the requirement has come from a business user stating "The amount should be **stored**…" then consider that users shouldn't be providing the technical requirements, and they really mean that's how they want to see the data presented. How you store data is up to you as the technical expert.

In these cases, the developer should consider that often, in larger/ongoing projects, the initial requirement may have been stipulated by one user or department, and later on there may be other requirements that want to use the same value, but display it differently, e.g. the Finance Department provided that initial requirement, but later the Payroll department say they want to provide the same data on cheques to be paid to people, and they don't want leading zeros on that. If you'd stored it with leading zeros what would you then do? You'd have to strip leading zeros off it to display it differently.

The thing to remember here is that, how data is stored and how it is displayed should be kept separate. Store the data in one consistent and most appropriate format, and then allow the displaying of the data to vary depending on requirements (or even individual user settings!).

What about things like Telephone Numbers or Social Security Numbers?

In this case, these are not numeric values. You wouldn't take the telephone number or social security number and try and, for example, multiple it by 76 to get a result. These types of 'numbers' don't need computation done on them. Instead they are more of an identifier, and it is correct and perfectly acceptable to store these in a VARCHAR2 datatype.

So why not store all numbers in a VARCHAR2 datatype?

When you store something as a VARCHAR2, every digit in that number becomes a character in the string, and is stored in its own byte. We can see this using SQL's DUMP function:

```
SQL> select '12300' as num, dump('12300') as dmp from dual;
NUM   DMP
----- ----------------------------------------
12300 Typ=96 Len=5: 49,50,51,48,48
```

Each of those bytes represents the ASCII character value of each digit, e.g. 49 = '1', 50 = '2' etc. If we look at the same value as a NUMBER datatype:

```
SQL> select 12300 as num, dump(12300) as dmp from dual;
       NUM DMP
---------- ----------------------------------------
     12300 Typ=2 Len=3: 195,2,24
```

It doesn't take as many bytes to store it. So, that's the first benefit of using a NUMBER over a VARCHAR2, but what else?

Well, numbers stored as strings cannot be sorted in a numeric sense. Let's order some data stored as VARCHAR2:

```
SQL> select num from myvarcharnums order by num;
NU
--
1
10
11
2
20
21
3
```

Why are they not sorted into their numeric order? Let's look at the bytes for them:

```
SQL> select num, regexp_substr(dump(num),'[^ ]+$') as dmp from myvarcharnums order by num;
NU DMP
-- ----------------------------------------
1  49
10 49,48
11 49,49
2  50
20 50,48
21 50,49
3  51
```

When you look at the ASCII values of each digit, you can see more clearly where the ordering has come from. Everything is ordered by the first character's ASCII value, then within those groups ordered by the second characters ASCII value etc. Ordering of VARCHAR2 strings is done on a character by character basis.

Apart from VARCHAR2 strings taking up more bytes, and the sorting of the data being broken, is there any other good reason? There certainly is.

Oracle provides in built validation for the datatype you use. If you try and store a non-numeric value in a NUMBER datatype, you'll correctly get an exception raised. Conversely, if you try and store a non-numeric value in a VARCHAR2 datatype, there won't be any exception… at least not until you try and use that value as a number and find your code breaks because it can't work out what, for example, 100*'APPLE' evaluates to.

So, storing numbers in a VARCHAR2 datatype is, as you can see, definitely a bad idea, and may introduce bugs that you won't know about until some user has entered some invalid data without any exception raised.

3. How are NUMBER values stored?

As you've seen above, storing numeric values as NUMBER datatype is certainly preferred over using something like VARCHAR2, or any other character based datatype. But how does Oracle store the numbers in those bytes?

> ***For the following, I've developed a quick procedure that will display the byte's values of my numbers (like SQL's DUMP function), and also show the bytes in their binary bitwise representation. If you don't know what bytes and bits are… stop reading right now, and go browse the internet to learn what these are, otherwise you're going to struggle to follow this.***

In the above example, we saw the SQL DUMP function used to show us the bytes that are stored. You'll also notice that Oracle stores some information about the Datatype ("TYP=…") and the Length of the value in bytes ("LEN="). For this article, we'll just focus on the bytes of the value itself, as we know we're dealing with NUMBER type and we're no longer particularly concerned with the actual length.

General Byte Structure

In general, NUMBER datatypes are stored with 1 byte containing information about the value (known generally as the 'exponent' byte), and up to 20 bytes containing the actual value (known as the 'mantissa'). Maximum 21 bytes.

Bytes:

Exponent	$Mantissa_1$	$Mantissa_2$	$Mantissa_3$	…	…	$Mantissa_{19}$	$Mantissa_{20}$

Each of those 20 bytes is 'base 100' meaning it will store a value from 0-99, which you can also consider to be 2 digits from the original number. The value is also stored in 'scientific' format, so the first byte represents digits to the left of the decimal place, and the remaining bytes are digits after a decimal place.

This may sound confusing, but things will become clearer as we look at some specific examples:

Zero

Zero is a special case. Let's take a look:

```
SQL> exec bytewize.show_bytes(0);
Bytes:
       128

1
2 6 3 1
8 4 2 6 8 4 2 1

1 0 0 0 0 0 0 0
```

Zero just takes a single byte, and has no bytes for its value, only the exponent byte.
The topmost bit (128) of this byte is set to a value of 1. This bit is common amongst all numbers and represents whether the value is positive or negative (1 = positive, 0 = negative)

Zero is the only value to have a single byte representation and no 'mantissa' bytes.

Positive Numbers

Let's take a look at a positive number:

```
SQL> exec bytewize.show_bytes(123.45);
Bytes:
       194              002              024              046

 1                1                1                1
 2 6 3 1          2 6 3 1          2 6 3 1          2 6 3 1
 8 4 2 6 8 4 2 1  8 4 2 6 8 4 2 1  8 4 2 6 8 4 2 1  8 4 2 6 8 4 2 1

 1 1 0 0 0 0 1 0  0 0 0 0 0 0 1 0  0 0 0 1 1 0 0 0  0 0 1 0 1 1 1 0
```

The Exponent byte

As with the Zero value, we can see the top bit (128) of the first byte is 1, indicating that our value is a positive number.
The remaining 7 bits of that first byte has a value of 66;
64 is added to positive exponents, so we subtract 64 giving us a base 100 exponent of 2.

```
Exponent = 2
```

The Mantissa Bytes

Remaining bytes are the 'mantissa', each representing 2 numeric digits (hence if < 10 then consider a leading 0)

```
Byte2 = 02
Byte3 = 24
Byte4 = 46
```

Note, however, that all bytes (including the exponent) have 1 added to them. This is to prevent NUL bytes (a NUL byte is a byte with a value of zero, and is used in the C language, for example, to represent the end of a string – Oracle is generally written in the C language internally). So, let's subtract 1 from all our bytes:

```
Exponent = 1
Byte2    = 01
Byte3    = 23
Byte4    = 45
```

Considering our number in scientific notation we now have 01.2345 with a base 100 exponent of 1

```
1.2345 * power(100,1) = 1.2345 * 100 = 123.45
```

Let's check that…

```
SQL> select 01.2345 * power(100,1) from dual;
01.2345*POWER(100,1)
--------------------
              123.45
```

Sure enough, there's our original number.

Negative Numbers

For negative numbers, you may think it's just a case of that top bit in the first byte having a value of 0. If only it were that simple. (Actually it's not that different, there's just an 'inverse' way of looking at the bytes)

Let's take a look at a negative number:

```
SQL> exec bytewize.show_bytes(-123.45);
Bytes:
       061             100             078             056             102
┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐ ┌─────────────┐
│1            │ │1            │ │1            │ │1            │ │1            │
│2 6 3 1      │ │2 6 3 1      │ │2 6 3 1      │ │2 6 3 1      │ │2 6 3 1      │
│8 4 2 6 8 4 2 1│ │8 4 2 6 8 4 2 1│ │8 4 2 6 8 4 2 1│ │8 4 2 6 8 4 2 1│ │8 4 2 6 8 4 2 1│
├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤
│0│0│1│1│1│1│0│1│ │0│1│1│0│0│1│0│0│ │0│1│0│0│1│1│1│0│ │0│0│1│1│1│0│0│0│ │0│1│1│0│0│1│1│0│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

The Exponent Byte

Here we can see that the top bit (128) is set to 0 indicating our negative number. Great, we expected that.
The remaining 7 bits have a value of 61
The exponent value, when it is stored, is determined as

```
Byte = (127-exponent)-64
```

Rewriting that equation we get:

```
exponent + 64 = 127-byte
```

(as with positive number it has had 64 added to it)

So, for our value of 61:

```
exponent + 64 = 127-61
exponent + 64 = 66
exponent = 66-64
exponent = 2
```

The mantissa bytes are stored as 100-'2 digit value'

```
byte = 100-value
```

So, rewriting that we get

```
value = 100-byte

byte2 = 100-100 = 00
byte3 = 100-78 = 22
byte4 = 100-56 = 44
byte5 = 100-102 = -2
```

The exponent has had 1 added to it (as with positive numbers), so we subtract 1 from it, however, unlike positive number the other bytes have had 1 subtracted from them, so we add 1 to each of them. (If you prefer you could also have looked at the above as being value = 101-byte to incorporate this step)

That gives us:

```
Exponent = 1
Byte2    = 01
Byte3    = 23
Byte4    = 45
Byte5    = -1
```

Let's ignore byte5 for a moment. When we put those values together, as we did with the positive number it leaves us with:

```
01.2345 * power(100,1) = 1.2345 * 100 = 123.45
```

And taking account of our top bit of the exponent byte, which indicated it was negative, our number is

```
-123.45
```

What about that last byte with a value of -1 that we ignored?

With negative numbers the last byte is always specified with a value of 102 (representing -1). I'm not sure of the reason for this, though I guess it's something internal to Oracle's processing of numbers, or maybe some legacy thing used as an indicator to terminate the bytes. There is also an exception to this, and that is when the number of bytes is at the limit of the bytes permitted for the NUMBER datatype. In that case there will not be a -102 value byte. In that respect, we can just ignore that byte as it has no real value to us.

Let's just quickly see that with SQL's DUMP function; Firstly, a number just on the limit of the bytes…

```
SQL> select dump(-1.23456789012345678901234567890123456789) from dual;
DUMP(-1.23456789012345678901234567890123456789)
--------------------------------------------------------------------------------
Typ=2 Len=21: 62,100,78,56,34,12,100,78,56,34,12,100,78,56,34,12,100,78,56,34,102
```

You can see the 102 byte on the end.
And one more digit to just push it over…

```
SQL> select dump(-1.234567890123456789012345678901234567890) from dual;
DUMP(-1.234567890123456789012345678901234567890)
--------------------------------------------------------------------------------
Typ=2 Len=21: 62,100,78,56,34,12,100,78,56,34,12,100,78,56,34,12,100,78,56,34,21
```

The 102 byte is no longer there, but the number is still ok.

20 bytes Mantissa?

Now I know what you're thinking, we have 20 bytes to store our mantissa, and each of those bytes can store 2 digits of the number, so in theory we can create numbers with 40 digits. Yet the Oracle documentation clearly states that the maximum we should specify is NUMBER(38) and not NUMBER(40).

The last mantissa byte can be subject to roundings etc. as part of numeric calculations internally, so should not be relied upon, and hence why Oracle guarantees maximum precision of 38 (19 bytes with 2 digits each) for numbers using this type.
If the datatype is just specified as NUMBER, the precision is actually for 40 digits, but the last 2 digits may be subject to unreliable values, though most people don't notice as they're not trying to store numbers of such large magnitudes.

Why 2 digits per byte?

If a byte can store a value from 0-255 and multiple bytes can hold values up to 256^{bytes}-1, then isn't Oracle being wasteful just storing values from 0-99 in each byte, and taking up more bytes than necessary? Surely it be more efficient to just store numeric values in something like a 64 bit (8 byte) value, or a 128bit (16 byte value)?

There are various reasons Oracle does what it does.

1) Having fixed numbers of bytes for a number datatype would mean that even the small numbers would take up e.g. 8 bytes or 16 bytes. That would likely prove to be more wasteful on space in the long run.
2) Storing floating point numbers is not as easily done in binary, which is more suited to whole integers. Sure, there are datatypes that can do binary floating point numbers (such as BINARY_FLOAT) but there can be issues with accuracy around working with such numbers.
3) Oracle's internal NUMBER format, works based on decimal precision rather than Binary precision. This can be easily seen if we try and convert just a simple number between the two datatypes:

```
create table tst (x binary_float);

Table created.

insert into tst (x) values (0.1);

1 row created.

select * from tst;
         X
----------
  1.0E-001
```

As a Binary Float number it looks ok (my output giving us the scientific notation by default 1 times 10 to the power of -1, which is 0.1). What if we convert that to a NUMBER datatype...

```
select to_number(x) from tst;
TO_NUMBER(X)
------------
  .100000001
```

Hmmm, it's not as accurate as we would like. Our value seems to have increased slightly.
(Note: This is not a bug in Oracle, just an inaccuracy caused between binary and decimal with floating point numbers.)

Oracle's NUMBER datatype, using decimal precision is actually better for floating point mathematics than those datatypes that use Binary precision.

4. WHAT ABOUT OTHER NUMERIC DATATYPES?

In the above section, we've already seen the use of BINARY_FLOAT and how it doesn't quite have the same mathematical precision as using Oracle's NUMBER datatype.

For the NUMBER datatype, Oracle have developed in internal library that specifically handles decimal precision mathematics, so in most cases using NUMBER is the best choice. (And in truth, for most practical applications, I have rarely had any need to use BINARY_FLOAT (32bit), or BINARY_DOUBLE (64 bit) numbers).

The one datatype that I have used that is not NUMBER datatype, is PLS_INTEGER. This is not an SQL datatype, so is only used in PL/SQL code, and in earlier versions of Oracle (9i and before) it was known as BINARY_INTEGER. The two names are synonymous now, so the advice is to use PLS_INTEGER as it's the newer standard.

What do we use PLS_INTEGER for?

Documentation Reference: http://docs.oracle.com/database/121/LNPLS/datatypes.htm#i10726

PLS_INTEGER can be useful if we're only dealing with integer (non floating point) values in PL/SQL code. Its value is limited by the fact it's a 32bit value, so has a range -2,147,483,648 through 2,147,483,647. (It also has some sub-types, you can read about in the documentation, but I won't cover here)

This range, like any signed binary value, is determined by the number of bits used. In this case we have 32 bits, and the very top bit is used to indicate whether the value is positive or negative (unlike Oracle's NUMBER datatype, binary numbers use a value of 1 in the top bit to represent negative and 0 to represent positive), so the range is limited to 31 bits:

```
SQL> select power(2,31) from dual;
POWER(2,31)
-----------
 2147483648
```

Thus, for positive numbers the range is 0 to 2,147,483,647 (giving us 2,147,483,648 values), and for negatives we have -1 to -2,147,483,648 (giving us 2,147,483,648 values in that direction).

Side Note: This sort of range calculation will appear familiar if we were to look at the range of a 16 bit (2 byte) signed binary number:

```
SQL> select power(2,15) from dual;
POWER(2,15)
-----------
      32768
```

giving us a positive range of 0 to 32767. Think what the maximum size of a VARCHAR2 can be in PL/SQL (and SQL since from 12c onwards)?

As long as we're dealing with numbers in PL/SQL within that range, we can use a PLS_INTEGER. The benefit of doing this, is that if we perform any integer mathematics on this value (providing the result remains within the range), then the mathematics is done at the low level within the computer's processor, rather than a higher level library (like NUMBER uses), and thus any heavy processing on such numbers can be much more performant.

Most of the time, however, you will just see people using PLS_INTEGER as the index type for Associative Arrays.

5. Just for Fun (Binary! Binary! Binary!)

Ok, so above I've covered the details about Oracle's NUMBER datatype, and how it's stored internally (and why numbers should be stored as NUMBER and not VARCHAR2), and I've touched on the Binary datatypes that Oracle also provides (though are typically used less). Just for fun, I thought I'd also show how binary numbers are stored, and how some basic binary mathematics is performed with the maths processing units of a typical computer. You don't necessarily need to know this for Oracle, but it's a good basis to understanding the core concepts of how computers deal with their bits and bytes, if it's not something you're already familiar with, and can give you a better appreciation of the difficulties early software developers had when having to write code in low level languages (especially when we were on 8bit or 16bit computers and wanted to deal with large numbers – Oh Yes! I remember those days!)

Positive Binary Numbers

So, let's start by looking at a single byte with a positive number in it:

```
SQL> exec bytewize.show_binary_bytes(123,1);
Bytes:
       123
```

1							
2	6	3	1				
8	4	2	6	8	4	2	1
0	1	1	1	1	0	1	1

That's fairly straightforward. 64+32+16+8+2+1 = 123

Negative Binary Numbers

What about a negative value:

```
SQL> exec bytewize.show_binary_bytes(-123,1);
Bytes:
       133
```

1							
2	6	3	1				
8	4	2	6	8	4	2	1
1	0	0	0	0	1	0	1

Hmmm… so what does that show us?

Similar to Oracle's NUMBER datatype, the top bit (in this case bit 128) indicates whether it's a negative number or not. Unlike the NUMBER datatype it uses a 1 to indicate negative and 0 for positive (It's Oracle's NUMBER datatype that is doing it backwards).

So, bit 128 is indicating our negative number, but then what about the other 7 bits…

```
4+1 = 5
```

Well, that's not 123.
But, if we look, we can see that 5 is actually 128-123. The reality is that, including the sign bit, this is the 2's complement of an 8 bit signed number (2^{bits}-number) e.g. 2^8-123 = 256-123 = 133, which you can see is the positive value of that byte we got.

Another way to look at this (and how it is typically done internally) is to take the 1's complement (inverting all the bits of the positive value of the number) and then adding 1.

So, let's take 123, and apply 1's complement to it…

```
1
2 6 3 1
8 4 2 6 8 4 2 1
+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
+---+---+---+---+---+---+---+---+

1's complement becomes (invert all the bits):

1
2 6 3 1
8 4 2 6 8 4 2 1
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
+---+---+---+---+---+---+---+---+

And add 1 to give:

1
2 6 3 1
8 4 2 6 8 4 2 1
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
+---+---+---+---+---+---+---+---+
```

Which is the value we saw in our byte.

We also get the same byte if we provide a positive value of 133:

```
SQL> exec bytewize.show_binary_bytes(133,1);
Bytes:
       133
1
2 6 3 1
8 4 2 6 8 4 2 1
+---+---+---+---+---+---+---+---+
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
+---+---+---+---+---+---+---+---+
```

So, how do we know if our value is a positive number or a negative number?

The answer is, we don't. We just have to know if we are working with "signed" or "unsigned" values. If we know we're working with signed values, then we know that we have a range, in a single byte of -128 to +127. If we know we're working with unsigned values, then we know our range is 0 to 255.

WHAT ABOUT USING MULTIPLE BYTES?

Multiple bytes work in the same way as a single byte. If we are dealing with signed numbers, then the topmost bit of the topmost byte indicates if the number is positive or negative. Let's look at our same value if it's stored in 2 bytes (16 bits):

```
SQL> exec bytewize.show_binary_bytes(123,2);
Bytes:
       000               123
┌─────────────────┐ ┌─────────────────┐
│1                │ │1                │
│2 6 3 1          │ │2 6 3 1          │
│8 4 2 6 8 4 2 1  │ │8 4 2 6 8 4 2 1  │
├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤
│0│0│0│0│0│0│0│0│ │0│1│1│1│1│0│1│1│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘

SQL> exec bytewize.show_binary_bytes(-123,2);
Bytes:
       255               133
┌─────────────────┐ ┌─────────────────┐
│1                │ │1                │
│2 6 3 1          │ │2 6 3 1          │
│8 4 2 6 8 4 2 1  │ │8 4 2 6 8 4 2 1  │
├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤
│1│1│1│1│1│1│1│1│ │1│0│0│0│0│1│0│1│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

You should be able to see from this how the negative number is the 1's complement of the positive number, with 1 added to it. The bit 128 of the top most (left most) byte, is the indicator for the negative number. In this case we have a range of values from -32768 to 32767. Let's just check out those two values:

```
SQL> exec bytewize.show_binary_bytes(32767,2);
Bytes:
       127               255
┌─────────────────┐ ┌─────────────────┐
│1                │ │1                │
│2 6 3 1          │ │2 6 3 1          │
│8 4 2 6 8 4 2 1  │ │8 4 2 6 8 4 2 1  │
├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤
│0│1│1│1│1│1│1│1│ │1│1│1│1│1│1│1│1│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘

SQL> exec bytewize.show_binary_bytes(-32768,2);
Bytes:
       128               000
┌─────────────────┐ ┌─────────────────┐
│1                │ │1                │
│2 6 3 1          │ │2 6 3 1          │
│8 4 2 6 8 4 2 1  │ │8 4 2 6 8 4 2 1  │
├─┬─┬─┬─┬─┬─┬─┬─┤ ├─┬─┬─┬─┬─┬─┬─┬─┤
│1│0│0│0│0│0│0│0│ │0│0│0│0│0│0│0│0│
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
```

WHY 2'S COMPLEMENT?

So, we've seen how positive and negative numbers are represented in binary, but why are negative numbers stored in 2's complement? Why not just the positive value, with the topmost bit set to indicate the sign of the number?

A) 2's Complement for Inline Binary logic

Well, one of the reasons stems from the early computing languages such as machine code and assembly language (and you'll still find this in the C language and some others). Storing data in those languages wasn't always strongly typed, so there was little checking around whether you were dealing with numbers, or characters or whatever; everything was just bytes.

If we wanted to store whether something was 'true' or 'false' then these Boolean values had to be stored as binary. As a first thought you may just think of storing 0 for false and 1 for true. That would seem simple enough, and sure, that can work… in some contexts. However we were dealing with bytes, and often using binary logic (NOT, AND, OR, XOR (exclusive OR) etc.), so there were some tricks we could use.

Consider the following:

We have a value stored in a byte (or bytes), and we want to add another value to it depending on whether two other values are equal or not. Now, in PL/SQL you'd do something like:

```
if a = b then
  val = val + val2;
end if;
```

Wouldn't it be cool though if we could do something like:

```
val = val + (val2 AND (a=b));
```

In PL/SQL you can't do that, but in some earlier languages you could, and the reason why you could was because of two things:

1. AND was a binary logical operator
2. The Boolean values of TRUE and FALSE were represented by -1 for TRUE and 0 for FALSE.

Why are those two things significant? Well, let's demonstrate:

Scenario 1: Where a = b:

Val = 1
Val2 = 2
A = 3
B = 3

Because A=B, the result of the comparison is TRUE, so that is stored as -1:

```
SQL> exec bytewize.show_binary_bytes(-1,1);
Bytes:
       255
┌─────────────────┐
│1                │
│2 6 3 1          │
│8 4 2 6 8 4 2 1  │
├─┬─┬─┬─┬─┬─┬─┬─┤
│1│1│1│1│1│1│1│1│
└─┴─┴─┴─┴─┴─┴─┴─┘
```

Now the binary value of “val2” is logically AND’d with the binary value of TRUE:
A Logical AND of 2 bits is 1 if both bits are 1, else the result is 0:

```
1 1 1 1 1 1 1 1  TRUE

0 0 0 0 0 0 1 0  VAL2
-----------------
0 0 0 0 0 0 1 0  VAL2 AND TRUE
```

So, now when the result of “VAL2 AND (A=B)” (or VAL2 AND TRUE) is added to our intial VAL value, we would get 1+2 giving a result of 3.

Scenario 2: Where a <> b:

Val = 1
Val2 = 2
A = 3
B = 4

Because A does not equal B, the result of the comparison is FALSE, so that is stored as 0 (I won’t show what a zero value looks like as a byte, I’m sure you know).

Now the binary value of “val2” is logically AND’d with the binary value of FALSE:
A Logical AND of 2 bits is 1 if both bits are 1, else the result is 0:

```
0 0 0 0 0 0 0 0  FALSE

0 0 0 0 0 0 1 0  VAL2
-----------------
0 0 0 0 0 0 0 0  VAL2 AND TRUE
```

So, now when the result of “VAL2 AND (A=B)” (or VAL2 AND FALSE) is added to our intial VAL value, we would get 1+0 giving a result of 1.

As you can see from this, by storing TRUE as -1 and FALSE as 0, it allowed us to apply some pretty cool logic to conditionally apply values within our code, simply because of the binary AND logic we apply.

It was a nice trick, but unfortunately, as many languages no longer work at this low level of binary, it’s something we can’t use and don’t see used much any more.

B) 2's Complement for Binary Mathematics

The other, and main, reason for two's complement is if we perform mathematics on our binary values. Let's just go back to our original positive and negative values we were looking at, 123 and -123…

```
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |  123

| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |  -123
```

Now lets' add those together. Just like in decimal addition, we add the two digits starting at the right hand side, and if they exceed the upper value for the digits, we '**carry**' over to the next. (You'll be seeing a lot more of "carry")

```
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |  123

| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |  -123
-----------------

|   |   |   |   |   |   |   | 0 |  bit1 = 1+1 = 0 carry 1

|   |   |   |   |   |   | 0 | 0 |  bit2 = 1+0 + carried 1 = 0 carry 1

|   |   |   |   |   | 0 | 0 | 0 |  bit3 = 0+1 + carried 1 = 0 carry 1

|   |   |   |   | 0 | 0 | 0 | 0 |  bit4 = 1+0 + carried 1 = 0 carry 1

|   |   |   | 0 | 0 | 0 | 0 | 0 |  bit5 = 1+0 + carried 1 = 0 carry 1

|   |   | 0 | 0 | 0 | 0 | 0 | 0 |  bit6 = 1+0 + carried 1 = 0 carry 1

|   | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  bit7 = 1+0 + carried 1 = 0 carry 1

| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  bit8 = 0+1 + carried 1 = 0 carry 1
```

Our result of adding these two single byte values is zero. And sure enough 123 added to -123 is zero.
You can surely imagine what would happen -123 was the same as 123 but just with the top bit set. If you're not sure, have a go add adding:

```
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |  123

| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |  123 with top bit set
```

And you should see you get:

```
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
```

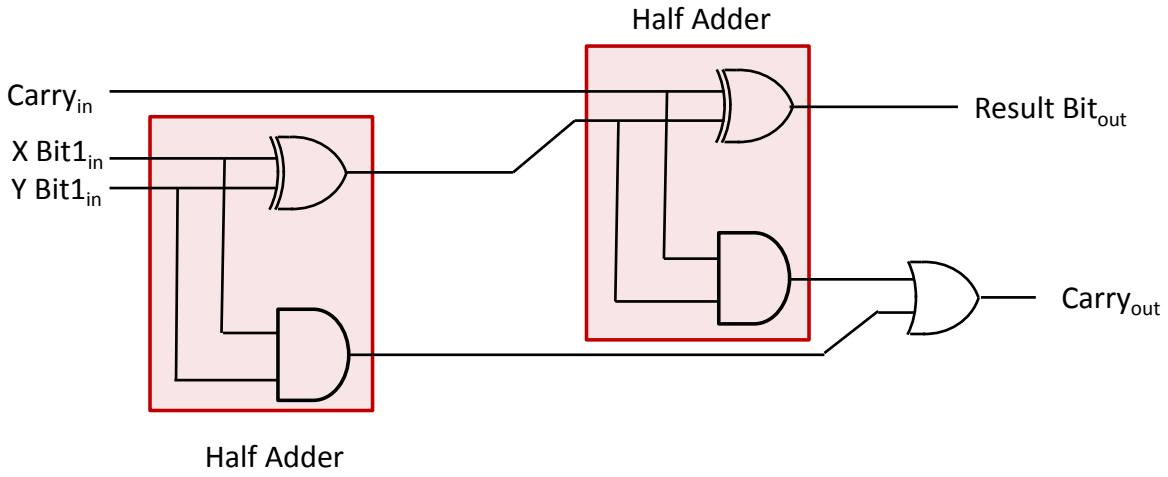
Which certainly isn't zero.

BINARY MATHEMATICS

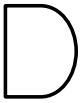
Adding Binary Numbers

Ok, so you've seen an example of adding two binary numbers above. Bit by bit you just add to two bits together. If the result is 2 you have a zero value and you are carrying 1; if the result is 3 (because you are adding in the carry value too) then the result is 1 and you are carrying 1. This is no different to adding decimal numbers, except that you are dealing with a base of 2 rather than a base of 10.

Internally, the addition of two bits, and a carry bit, is done using logic gates. I won't cover all the details (you can look them up), but the following diagram covers, what is known as The Full Adder. This takes a Carry bit + Bit1 + Bit 2 and gives a Result Bit + Carry bit:



AND



X	Y	Result
0	0	0
0	1	0
1	0	0
1	1	1

OR

X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	1

XOR
(exclusive OR)

X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

The Full Adder

Try following the logic of this yourself by putting different 0 and 1 values in to the input. You'll see that each "Half Adder" has an XOR logic gate that takes the two bits and does the main part of the adding, and also an AND logic gate that deals with the carrying of a value from that addition. To deal with the need to add 3 bits (the two main bits plus the incoming carry bit), we use two Half Adders, plus an OR logic gate for the final Carry value. These are the main components of the Full Adder. Multiple Full Adders can be combined to deal with all the bits in the byte(s) we are adding, as you'll see a little later.

Subtracting Binary Numbers

Now, I can see you're getting scared, and you're wondering what "magic" logic gates I'm going to use to demonstrate how we subtract two binary numbers. Hang on though. Think back to our earlier example, when we added 123 and -123 together to get 0!

> ***Subtraction is just addition, but using a negative number.***

So, if for example, I wanted to subtract a Y value, 10 from an X value, 123, this is the same as saying I want to add 123 and -10. Therefore it makes sense that all we need to do to our value of 10, would be to create the 2's complement of it, and then add it to 123. And we know by now how to get the 2's complement by inverting all the bits and adding 1.

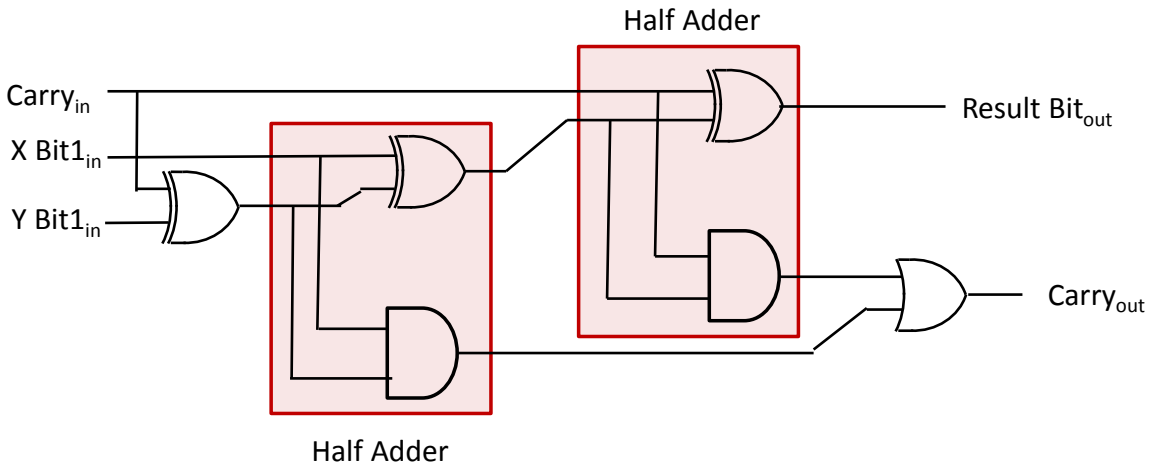
Look at our Full Adder above, we have an input that includes our two bits from our X and Y bytes, and also an input carry bit. When we add numbers normally, we would have a 0 in the initial carry bit, as we're not carrying any value in for the first bit we calculate. But for negative numbers we want to add 1 to the inverted value, so that initial carry bit is ideal for that, we can just set it to 1. To invert all the bits, we could do that for every bit of the Y value by including a NOT logic gate… however…

Let's consider our logic… If the carry bit starts as 0, we are dealing with positive numbers and don't want to invert the bits of our Y value; but if the carry bit starts as 1, we are dealing with a negative Y value and want to invert all the bits of it.

The logic table for that is:

Carry (positive/negative)	Y	Required Y
0	0	0
0	1	1
1	0	1
1	1	0

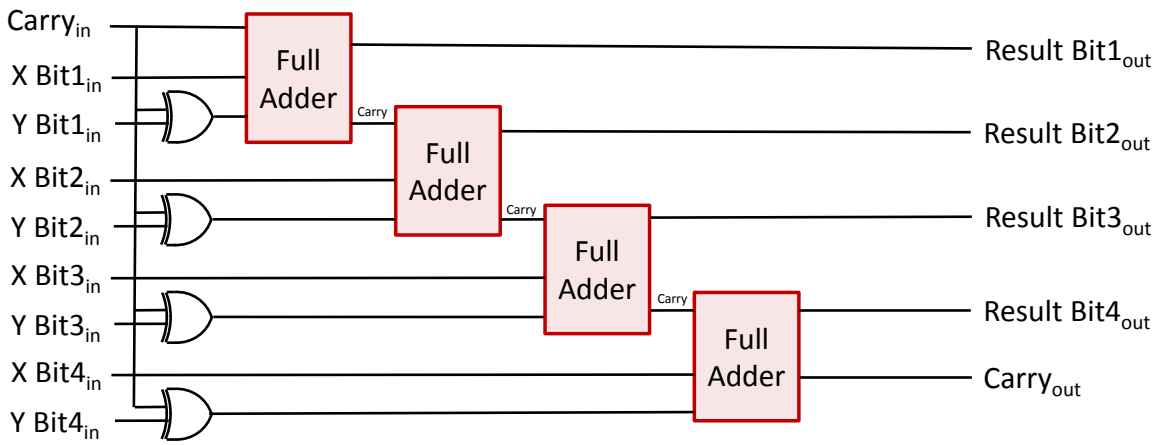
That matches the logic table we expect to see for an Exclusive OR (XOR). *Lightbulb moment*
So, rather than try and use a NOT gate and have something that is only relevant for subtraction, let's use a XOR gate and have something that can work for addition and subtraction, flagged by the initial "Carry" bit we use.



Adding and Subtracting Binary Numbers

By putting an XOR on our Y Bits, with the other input to the XOR being the initial carry value, we can use that initial carry to determine if the results should be an addition (0 carry), or subtraction (1 carry), causing our Y value to become negative and have 1 added (supplying the 2's complement value in the process!)

For two 4 bit numbers (X and Y), that would look something like this:



As you can see from that, the results of one "Full Adder" provides the carry value for the next "Full Adder", essentially providing the add-and-carry method we use for adding numbers together on paper. For an 8 bit number or 16 bit number etc. it would just be an extension of using more "Full Adder" circuits.

So, now we have a circuit of logic gates that are able to add or subtract binary numbers. Let me simulate that logic with some PL/SQL code I've written (I've removed some of the output to just show the essential information and save space):

```
SQL> declare
  2    result pls_integer;
  3  begin
  4    result := bitwize_maths.add_subtract_binary(33,123
  5                                               ,bitwize_maths.addbits
  6                                               ,bitwize_maths.byte
  7                                               );
  8    dbms_output.put_line('Result : '||result);
  9  end;
 10  /
Bytes:
       033
 ---------------
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
 ---------------
Bytes:
       123
 ---------------
| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
 ---------------

Bytes:
       156
 ---------------
| 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
 ---------------
Result : 156
```

The PL/SQL I've written simulates the logic gates, the half adder, the full adder, and the additional XOR needed to deal with adding and subtracting. In the above example, we are adding 33 to 123, giving us a result of 156. The simulation also outputs details of each bit being processed in turn by the Full Adders:

```
Bit: 1 of 8 (value=1)
Full Adder [IN: Bit1(1) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(0) Carry(1)]
Bit: 2 of 8 (value=2)
Full Adder [IN: Bit1(0) Bit2(1) Carry(1)]
Full Adder [OUT: Bit(0) Carry(1)]
Bit: 3 of 8 (value=4)
Full Adder [IN: Bit1(0) Bit2(0) Carry(1)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 4 of 8 (value=8)
Full Adder [IN: Bit1(0) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 5 of 8 (value=16)
Full Adder [IN: Bit1(0) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 6 of 8 (value=32)
Full Adder [IN: Bit1(1) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(0) Carry(1)]
Bit: 7 of 8 (value=64)
Full Adder [IN: Bit1(0) Bit2(1) Carry(1)]
Full Adder [OUT: Bit(0) Carry(1)]
Bit: 8 of 8 (value=128)
Full Adder [IN: Bit1(0) Bit2(0) Carry(1)]
Full Adder [OUT: Bit(1) Carry(0)]
```

If you follow this output, you can see for each bit (starting with the least significant bit) of both values, these are passed INto the Full Adder, along with the current Carry bit value, and the Full Adder (using Half Adders) produces an OUTput giving the result of adding those 3 bits, along with the resultant carry value, which itself will be used in the subsequent bit's calculation. Note, the initial carry bit is set to 0, indicating addition, and the Bit1 and Bit2 values passed into the Full Adder are those bits we can see in our byte values.

Let's do the same with Subtraction:

```
SQL> declare
  2    result pls_integer;
  3  begin
  4    result := bitwize_maths.add_subtract_binary(33,123
  5                                               ,bitwize_maths.subtractbits
  6                                               ,bitwize_maths.byte
  7                                               );
  8    dbms_output.put_line('Result : '||result);
  9  end;
 10  /
Bytes:
       033

| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |

Bytes:
       123

| 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |


Bytes:
       166

| 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Result : 166
```

So, it's telling us that 33-123 = **166**. Is that right?

Let's just check the output from the process:

```
Bit: 1 of 8 (value=1)
Full Adder [IN: Bit1(1) Bit2(0) Carry(1)]
Full Adder [OUT: Bit(0) Carry(1)]
Bit: 2 of 8 (value=2)
Full Adder [IN: Bit1(0) Bit2(0) Carry(1)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 3 of 8 (value=4)
Full Adder [IN: Bit1(0) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 4 of 8 (value=8)
Full Adder [IN: Bit1(0) Bit2(0) Carry(0)]
Full Adder [OUT: Bit(0) Carry(0)]
Bit: 5 of 8 (value=16)
Full Adder [IN: Bit1(0) Bit2(0) Carry(0)]
Full Adder [OUT: Bit(0) Carry(0)]
Bit: 6 of 8 (value=32)
Full Adder [IN: Bit1(1) Bit2(0) Carry(0)]
Full Adder [OUT: Bit(1) Carry(0)]
Bit: 7 of 8 (value=64)
Full Adder [IN: Bit1(0) Bit2(0) Carry(0)]
Full Adder [OUT: Bit(0) Carry(0)]
Bit: 8 of 8 (value=128)
Full Adder [IN: Bit1(0) Bit2(1) Carry(0)]
Full Adder [OUT: Bit(1) Carry(0)]
```

As you can see (and I've highlighted), the initial carry bit is set to 1, indicating subtraction, which is what we wanted, and all the Bit2 values being passed in to the Full Adder are the inverse of the bits in our second value (123). So the logic appears to have worked correctly.

Of course, if the result is negative, remember it's going to be represented, as we've already covered, as the 2's complement value.

Let's just check what -90 is as a signed byte:

```
SQL> exec bytewize.show_binary_bytes(-90,1);
Bytes:
       166
 ┌─────────────┐
 │1            │
 │2 6 3 1      │
 │8 4 2 6 8 4 2 1
 ├─┬─┬─┬─┬─┬─┬─┬─┤
 │1│0│1│0│0│1│1│0│
 └─┴─┴─┴─┴─┴─┴─┴─┘
```

Sure enough, it's 166; as a signed byte, the value of 166 is -90. So our logic gate "Full Adder" simulation is working correctly.

Now, I could go on to demonstrate this working with multiple bytes, but it would start taking up too much of this article, so you'll find the code for my two packages in an Appendix and you can have a go yourself.

Well, that's addition and subtraction covered nice and neatly in one process… but… "what about Multiplication or Division?" I hear you ask.

Well… if you insist… get ready for it… (maybe make yourself a cup of coffee at this point!)

Multiplication

Ok, let's think about multiplication for a minute. What do we mean when we say, for example, 3 x 4?

3 x 4 can also be seen as the addition of 3, 4 times: 3+3+3+3

Therefore, we could have some logic that goes along these lines (pseudo code):

```
result = 0
x = 3
y = 4
while y > 0
{
        result = result + x
        y = y - 1
}
```

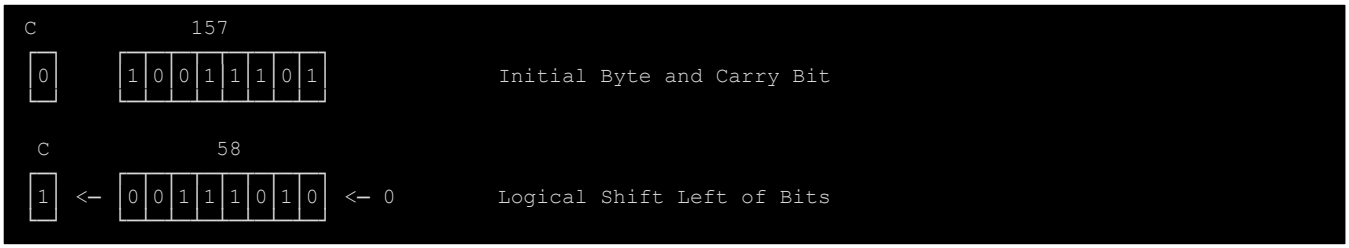
Sure, that would work. But now consider if we want to multiply by a number such as 8,765. Do we really want to loop around that many times adding our number to the result over and over again? There must be a better way?

Firstly, I'm going to side step and have a look at some other functionality that the processor can do on bytes…

As well as being able to add and subtract bits (and hence bytes), it can also shift bits in the bytes left or right. In summary there are 4 main "shift" methods that it uses (there can be others depending on the processor, but we'll just concern ourselves with these core ones).

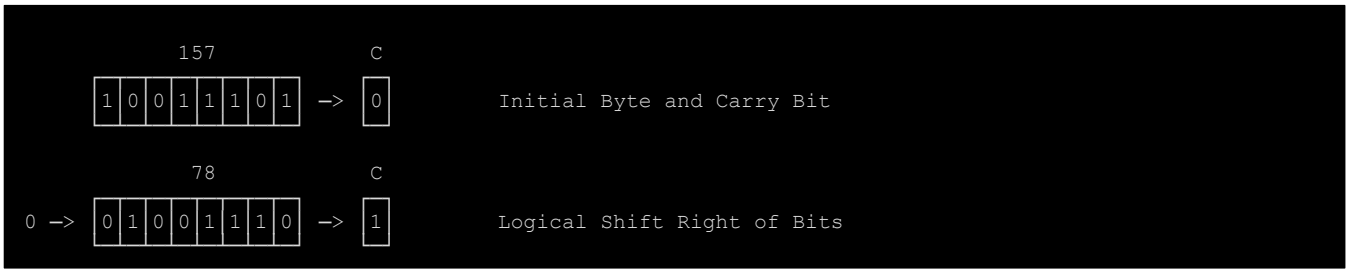
Logical Shift Left

A Logical Shift Left (LSL) moves all the bits in a byte one place to the left, and the topmost bit is moved into the 'Carry' bit, with the leastmost bit of the byte taking a value of 0



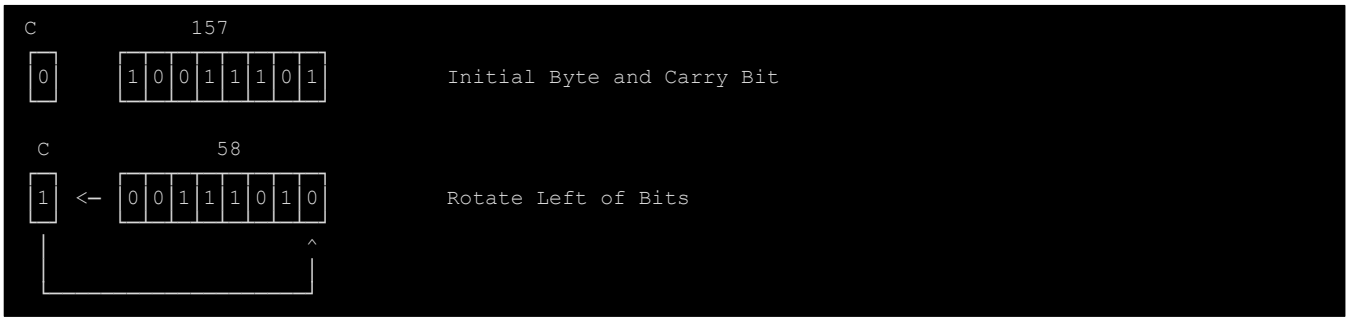
Logical Shift Right

A Logical Shift Right (LSR) is just the opposite of the Logical shift left, all the bits in a byte move one place to the right, and the leastmost bit is moved into the 'Carry' bit, with the topmost bit of the byte taking a value of 0



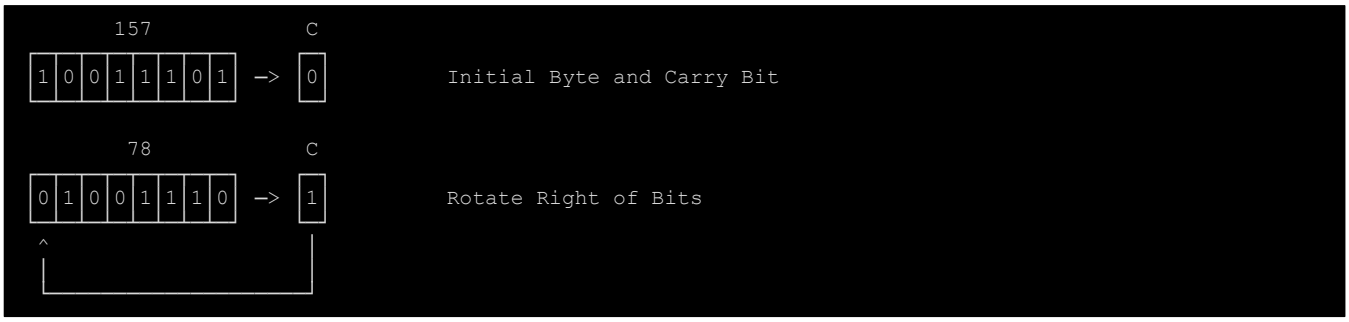
Rotate Left

A Rotate Left (ROL) is similar to the LSR, but the leastmost bit is populated with the current value of the carry bit, rather than a zero value.



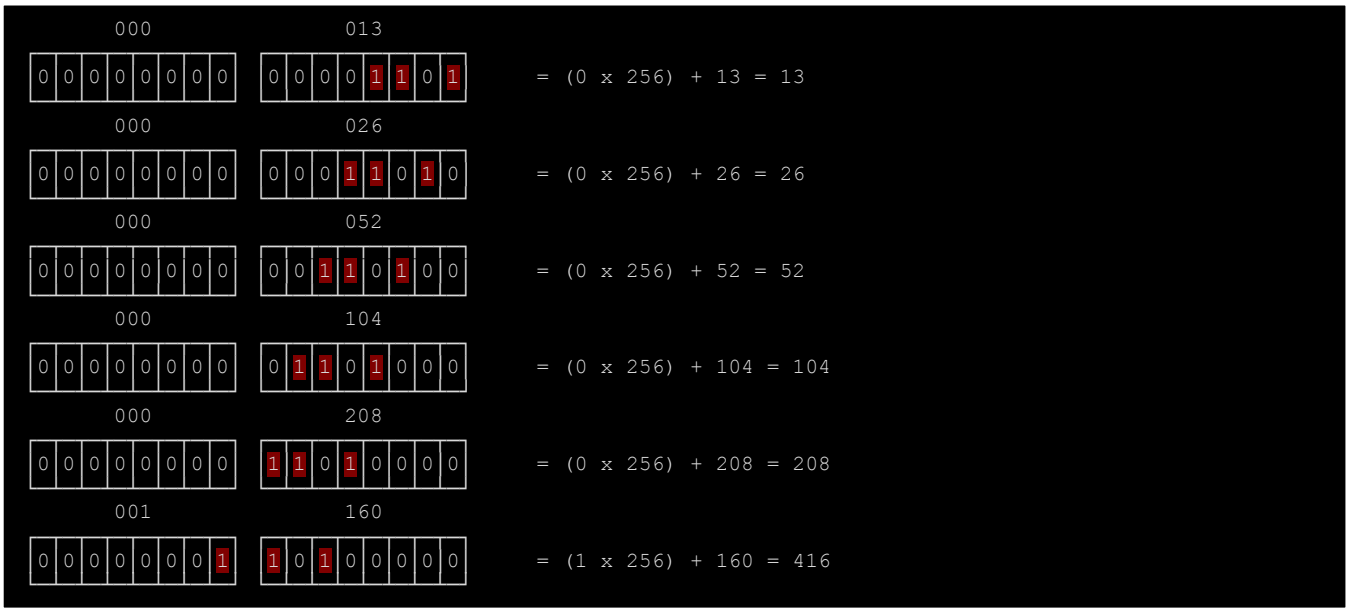
Rotate Right

And finally, the Rotate Right (ROR) is the opposite of the ROL.



So, that's fairly straightforward.

Now, it may not be immediately obvious from this, but when we shift bits to the left, we are in fact multiplying by 2, and likewise when we shift bits to the right we are dividing by 2 (after all, that's binary!). Let's look at a 2 byte number that doesn't put a value into the carry bit, and shift it to the left:



Clearly you can see the bits of the number shifting to the left and the binary value of the number multiplying by 2 each time.

Well, that would be great if all we ever wanted to do was multiply or divide by 2, or some power of 2 (4,8,16 etc.), but we want something that can multiply or divide by any number. Let's just go back to the basics of what multiplication is, using another example… 4 x 7. Now we know that that is the same as:

```
4 + 4 + 4 + 4 + 4 + 4 + 4    (4 added together 7 times)
```

But that's the long way. Can we abbreviate it somehow? Yes we can:

```
4 + (4 x 6)
```

That's got rid of the awkward odd number 7 (I never liked pesky odd numbers!), but why would we want to do that?
We know that when we multiply two numbers, it's possible to have other values that multiply to the same result, for example:

```
4 x 6   = 8 x 3
```

So, what did we do there?
The left hand side of our multiplication (let's call it "x") we multiplied by 2 (4 became 8). The right hand side of the multiplication (let's call it "y"), we divided by 2 (6 became 3). The result is still the same (both equal 24). In short we can show a multiplication as:

$$nx \times \frac{y}{n} \ (where\ n <> 0)$$

You can see from this, if n = 1 then we just have $x \times y$, but if n > 1, for example 2, then we would have $2x \times \frac{y}{2}$

We're dealing with integer numbers, so we don't use a value of n, where the division of "y" results in a non-integer. Thus, in our above example, we wouldn't further change 8 x 3 to 16 x 1.5. So, back to our calculation... Instead of `4 + (4 x 6)`, we can now represent that as:

```
4 + (8 x 3)
```

Now we've got one of those awkward odd numbers again for our "y" value. Look at what we did at the start to abbreviate it, and we can do the same thing, extract a single "x" value out of it to make our odd number even again:

```
4 + 8 + (8 x 2)
```

Again refactor our multiplication, now that we have an even "y" number, and it becomes:

```
4 + 8 + (16 x 1)
```

And as "y" is now odd, we extract the value of "x" out again:

```
4 + 8 + 16 + (16 x 0)
```

As our "y" is now zero, we can eliminate the multiplication part to leave us with:

```
4 + 8 + 16    which is 28.
```

Look at that. We've turned our multiplication, into addition, simply by a process of refactoring the multiplications with an "n" value of 2 each time. (our "x" multiplying by 2, and our "y" dividing by 2), just extracting a single "x" value when "y" was an odd number, so we could make "y" even again.

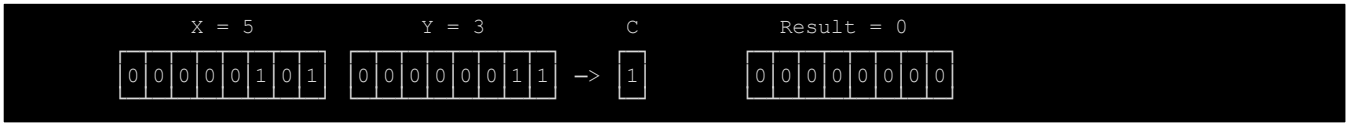
But hang on! What was it I said about shifting bits left and right?

"that would be great if all we ever wanted to do was multiply or divide by 2"

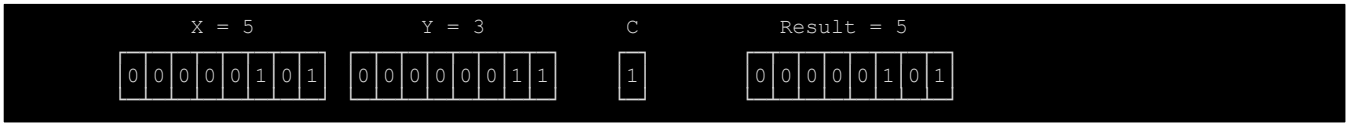
You've just seen the mathematical logic that takes two numbers, and by just using multiplication by 2 and division by 2 and simple addition, we've got the result we wanted. That means we CAN use our bit shifting technique after all:

Let's just take a couple of small numbers, say 5 and 7, and see what we can do with them using shifting:

Logically shift the Y value to the right, and we can tell if bit1 is set (it's was an "odd" number) (carry becomes 1):

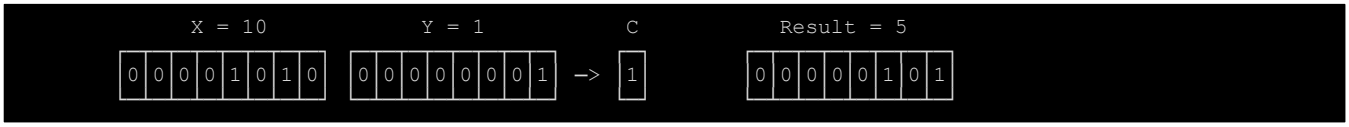


So, there's definitely 1 of our value X needed, let's add X to the result (we already know how to add):

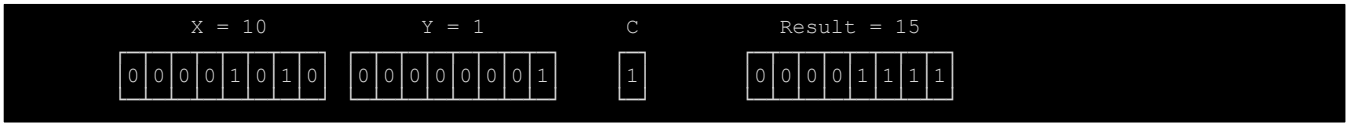


In determining Y was odd, by shifting the bit1 into the carry bit, we divided Y by 2 at the same time. Then as it was odd, we extracted our X value and added it to the result, and now we want to multiply our X value by 2; so shift it to the left:

Now if we shift Y to the right again (divide it by 2 and also determine if it was odd):

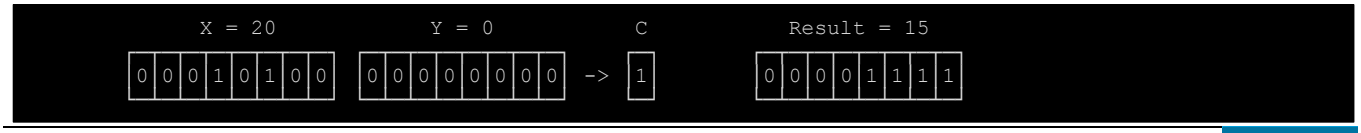


The carry bit is set to 1, so again we add X to the result:

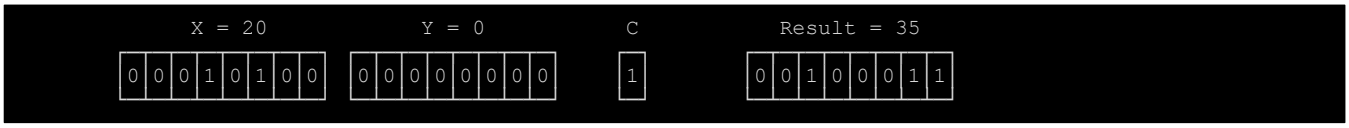


Again, shift X to the left:

And Y to the right:



Once again our carry bit is 1, so add X to the result:



Because we have no more Y left (Y=0), we can stop, and we have our result. 5 x 7 = 35.

On larger numbers, it is still just a case of shifting the bits left and right until our Y value reaches 7, so we could be multiplying by 8,765 and only require us to loop through the shifting process of X and Y, 14 times. That's pretty nifty.

> ***Note: The number of bits for X and the Result, must be at least twice the number of bits in Y, to account for the multiplied result, and the shifting of X to the left (in which case the shifting of the topmost bit will go into higher order bytes of the value)***

The pseudo code for the Multiplication operation of two numbers X and Y is as follows:

```
result = 0
while y > 0
{
        Logical Shift Right y
        if carry = 1 then result = result + x
        Logical Shift Left x
}
```

As it turned out, Multiplication wasn't as complicated as we may have expected. Let's see this simulated in our PL/SQL package (output manually formatted to save on space):

```
SQL> declare
  2    result pls_integer;
  3  begin
  4    result := bitwize_maths.multiply(123,5);
  5    dbms_output.put_line('Result : '||result);
  6  end;
  7  /
X = 123                          Y = 5             Carry   Result = 0
       000             123              005                      000             000
 0 0 0 0 0 0 0 0  0 1 1 1 1 0 1 1  0 0 0 0 0 1 0 1  [0]  0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0
---------- Iteration ---------
X = 246                          Y = 2             Carry   Result = 123
       000             246              002                      000             123
 0 0 0 0 0 0 0 0  1 1 1 1 0 1 1 0  0 0 0 0 0 0 1 0  [1]  0 0 0 0 0 0 0 0  0 1 1 1 1 0 1 1
---------- Iteration ---------
X = 492                          Y = 1             Carry   Result = 123
       001             236              001                      000             123
 0 0 0 0 0 0 0 1  1 1 1 0 1 1 0 0  0 0 0 0 0 0 0 1  [0]  0 0 0 0 0 0 0 0  0 1 1 1 1 0 1 1
---------- Iteration ---------
X = 984                          Y = 0             Carry   Result = 615
       003             216              000                      002             103
 0 0 0 0 0 0 1 1  1 1 0 1 1 0 0 0  0 0 0 0 0 0 0 0  [1]  0 0 0 0 0 0 1 0  0 1 1 0 0 1 1 1

Result : 615
```

Sure enough 123 x 5 = 615!!

Division

Ok, Multiplication was a bit of a lengthy topic, but I hope you can see that the principle of using simple bitwise manipulation, shifting bits left and right, allows our processor to multiply to integer numbers very quickly.

Let's finish off this area of Binary Mathematics, by demonstrating Division.

It probably won't surprise you by now to find that the binary processor also has a trick up its sleeve for division. Sure, it could go through the lengthy process of Long Division (as most of us learn in school), just applying that to binary values instead of decimal, but why do that when we already have a hint that the processor can divide by 2, just by shifting bits to the right.

Now, if you're like me (and I know I am!), you hated all that stuff at school about quotients, dividends, and divisors, so rather than bore you with the mathematical principle applying those to binary numbers, I leave it as an exercise for yourselves to search the web for "binary division" (avoiding those results that just use long division) to get a flavour of the why and how binary division works using shifting. You're probably already brain dead from all that stuff I've talked about with addition, subtraction and multiplication anyway, and if you're still reading this I'm impressed!

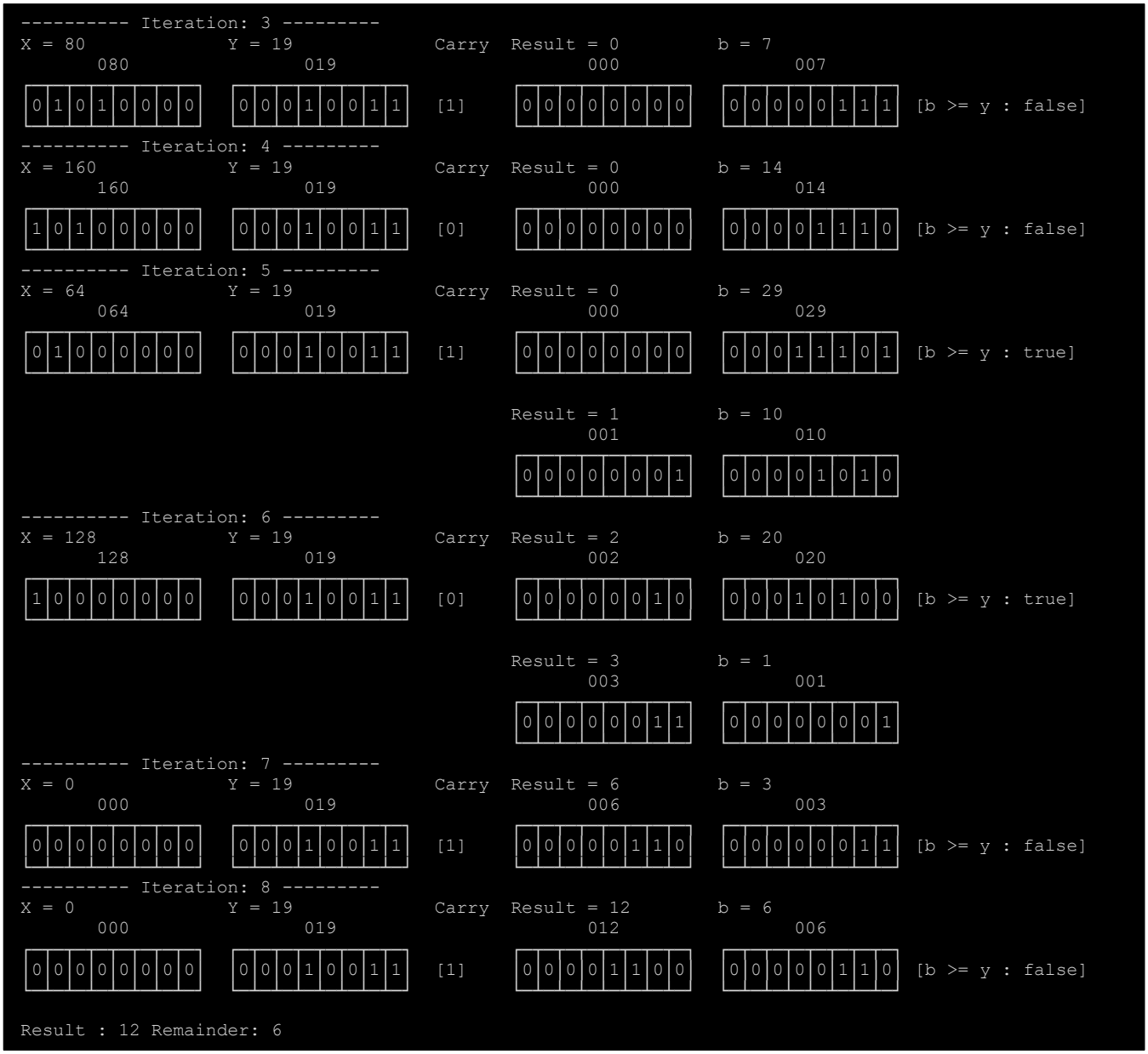
Instead, I'm going to provide you with the pseudo code, and a demonstration of binary division at work, so you can see just how easily binary division works (it's a bit like doing binary multiplication, but in reverse) using shifting of bits. So, first the pseudo code, assuming Result = x/y and we'll have some remainder that will end up in a variable "b"

```
result = 0
b = 0
loop from 1 to N                    (where N is the number of bits in "y" e.g. 8 bits for a byte)
{
        Logical Shift Left result
        Logical Shift Left x        (remember: top bit into carry)
        Rotate Left b               (remember: carry bit into low bit)
        if b >= y then
        {
                result = result + 1
                b = b - y
        }
}
```

And using our PL/SQL simulation of that (again the output has been manually adjusted to save space):

```
SQL> declare
  2    result    pls_integer;
  3    remainder pls_integer;
  4  begin
  5    result := bitwise_maths.divide(234,19,bitwise_maths.byte,remainder);
  6    dbms_output.put_line('Result : '||result||' Remainder: '||remainder);
  7  end;
  8  /
X = 234           Y = 19            Carry  Result = 0        b = 0
       234               019                    000              000
 1 1 1 0 1 0 1 0   0 0 0 1 0 0 1 1  [0]   0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 0
---------- Iteration: 1 ---------
X = 212           Y = 19            Carry  Result = 0        b = 1
       212               019                    000              001
 1 1 0 1 0 1 0 0   0 0 0 1 0 0 1 1  [1]   0 0 0 0 0 0 0 0  0 0 0 0 0 0 0 1  [b >= y : false]
---------- Iteration: 2 ---------
X = 168           Y = 19            Carry  Result = 0        b = 3
       168               019                    000              003
 1 0 1 0 1 0 0 0   0 0 0 1 0 0 1 1  [1]   0 0 0 0 0 0 0 0  0 0 0 0 0 0 1 1  [b >= y : false]
```

(cont…)



Look at that! $\frac{234}{19} = 12\ remainder\ 6$

SUMMARY

Well, it's been a lengthy article, and I expect this last section on Binary numbers has been a bit of an adventure for some people (and provided some surprisingly simple methods), and you've got through lots of cups of coffee (or just got sick of it and submitted it to the recycle bin)

You can, hopefully, now see the importance of using appropriate number datatypes, whether it's the Oracle NUMBER datatype with its decimal precision or the BINARY datatypes with their nifty internal logic. The Oracle NUMBER datatype uses a proprietary library written by Oracle to maintain the decimal precision we require for floating point numbers, whilst offering the best performance possible in achieving that precision. Conversely the BINARY datatypes offer a little less precision for floating points, but use the internal Arithmetic Logic Unit (ALU) of the computer's processor to give us outstanding performance for hard number crunching, especially with INTEGER values (signed or unsigned). When you consider that a single ADD or SUBtract operation of binary numbers within the processor typically takes a single clock cycle, then a single processor core running at 3.0GHz can theoretically[1] do 3 billion additions or subtractions per second. Multiplication or Division takes just a few clock cycles more.

The main thing to remember though, which was the initial point of this article, is that when you want to use numbers in your code or store numbers on your Oracle database, for the sake of everybody, please, please, PLEASE! use an appropriate numeric datatype. Numeric values simply do not belong in something like a VARCHAR2 string.

I've provided, as Appendices, the packages I created to help do the demonstrations in this article. I offer no warranty that they are in any way perfect, but please feel free to play with them if you find it helps you understand what I've been talking about. Just don't copy them or redistribute them, or pass them off as your own code, as you know it's not right; far better to understand the principles and have a go at writing your own, it's more fun too! That's how I learnt all this over the years (and a lot of the binary and logic gates stuff is things I did over 20 years ago at University; it tends to stick with you!)

[1] Assuming the code is written in low level assembly language compiled to native machine code, and that every operation is an ADD or SUBTRACT. Of course, in reality there is surrounding code dealing with the results, or higher level code, that will add their own overhead to the processing.

APPENDIX A – BYTEWIZE PACKAGE

> ***NOTE: Code should ideally be compiled and executed through SQL*Plus. Other environments or character sets may affect the box lines used, so you may have to tinker with it to suit your environment.***

```
set serverout on format wrapped
set linesize 255

create or replace package bytewize as
  /* (c) BluShadow @ community.oracle.com
         This code is provided for personal learning purposes only.
         It may not be copied, distributed, reproduced or modified without the
         express permission of the author, except for personal learning purposes.
         Do not breach copyright - you know it is wrong!
  */

  -- procedures are overloaded for different datatypes
  -- 'simple' flag indicates if the dbms_output should show basic byte details or
  -- more full header details.
  procedure show_bytes(val in number,        simple in boolean := false);
  procedure show_bytes(val in varchar2,      simple in boolean := false);
  procedure show_bytes(val in binary_float,  simple in boolean := false);
  procedure show_bytes(val in binary_double, simple in boolean := false);
  procedure show_bytes(val in date,          simple in boolean := false);
  procedure show_bytes(val in timestamp,     simple in boolean := false);

  -- Show binary values - implemented for numbers only
  procedure show_binary_bytes(val    in number
                             ,bytes  in number   := 1
                             ,simple in boolean  := false
                             ,lbl    in varchar2 := 'Bytes:'
                             );
end;
/

create or replace package body bytewize as
  /* (c) BluShadow @ community.oracle.com
         This code is provided for personal learning purposes only.
         It may not be copied, distributed, reproduced or modified without the
         express permission of the author, except for personal learning purposes.
         Do not breach copyright - you know it is wrong!
  */
  procedure output_bytes(str in varchar2, simple in boolean := false, lbl in varchar2 := 'Bytes:') is
    cursor cur_bits is
      with t as (select str as bytes from dual)
          ,byte as (select level as bytenum, regexp_substr(bytes, '[^,]+', 1, level) as byte
                    from   t
                    connect by level <= regexp_count(bytes,',')+1
                   )
          ,bits as (
            select bytenum, byte
                  ,sign(bitand(byte, 128)) as b128
                  ,sign(bitand(byte, 64)) as b64
                  ,sign(bitand(byte, 32)) as b32
                  ,sign(bitand(byte, 16)) as b16
                  ,sign(bitand(byte, 8)) as b8
                  ,sign(bitand(byte, 4)) as b4
                  ,sign(bitand(byte, 2)) as b2
                  ,sign(bitand(byte, 1)) as b1
            from   byte
            )
       select listagg('       '||to_char(byte,'fm009')||'       ', ' ') within group (order by bytenum)
as byte
             ,listagg('┌─────────────────┐', ' ') within group (order by bytenum) as byte_line
             ,listagg('│1                │', ' ') within group (order by bytenum) as head1
             ,listagg('│2 6 3 1          │', ' ') within group (order by bytenum) as head2
             ,listagg('│8 4 2 6 8 4 2 1  │', ' ') within group (order by bytenum) as head3
             ,listagg('├─┴─┴─┴─┴─┴─┴─┴─┤', ' ') within group (order by bytenum) as top
             ,listagg('│ │ │ │ │ │ │ │ │', ' ') within group (order by bytenum) as simple_top
             ,listagg('└─┴─┴─┴─┴─┴─┴─┴─┘', ' ') within group (order by bytenum) as bottom
```

```
           ,listagg('|'||b128||'|'||b64||'|'||b32||'|'||b16||'|'||b8||'|'||b4||'|'||b2||'|'||b1||'|',
' ') within group (order by bytenum) as bitstring
      from   bits;
  begin
    for i in cur_bits
    loop
      if simple then
        if lbl != 'Bytes:' then
          dbms_output.put_line(lbl);
        end if;
        dbms_output.put_line(i.byte);
        dbms_output.put_line(i.simple_top);
        dbms_output.put_line(i.bitstring);
        dbms_output.put_line(i.bottom);
      else
        dbms_output.put_line(lbl);
        dbms_output.put_line(i.byte);
        dbms_output.put_line(i.byte_line);
        dbms_output.put_line(i.head1);
        dbms_output.put_line(i.head2);
        dbms_output.put_line(i.head3);
        dbms_output.put_line(i.top);
        dbms_output.put_line(i.bitstring);
        dbms_output.put_line(i.bottom);
      end if;
    end loop;
  end;

  procedure show_bytes(val in number, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_bytes(val in varchar2, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_bytes(val in binary_float, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_bytes(val in binary_double, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_bytes(val in date, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_bytes(val in timestamp, simple in boolean := false) is
    str varchar2(2000);
  begin
    select regexp_substr(dump(val),'[^ ]+',1,3)
```

```
    into   str
    from dual;
    output_bytes(str, simple);
  end;

  procedure show_binary_bytes(val    in number
                             ,bytes  in number   := 1
                             ,simple in boolean  := false
                             ,lbl    in varchar2 := 'Bytes:'
                             ) is
    str varchar2(2000);
  begin
    if val > 0 then
      with t (val, bytes) as (select show_binary_bytes.val, show_binary_bytes.bytes from dual)
          ,x (val, res, bytes) as
             (select val, cast(null as varchar2(100)) as res, bytes from t
              union all
              select floor(x.val/256) as val
                    ,to_char(mod(x.val,256))||nvl2(x.res,','||x.res,x.res) as res
                    ,x.bytes-1
              from   x
              where  x.bytes > 0
             )
      select res
      into   str
      from   x
      where  bytes = 0;
    elsif val < 0 then
      with   t   (val,   bytes)   as   (select   power(2,8*show_binary_bytes.bytes)+show_binary_bytes.val,
show_binary_bytes.bytes from dual)
          ,x (val, res, bytes) as
             (select val, cast(null as varchar2(100)) as res, bytes from t
              union all
              select floor(x.val/256) as val
                    ,to_char(mod(x.val,256))||nvl2(x.res,','||x.res,x.res) as res
                    ,x.bytes-1
              from   x
              where  x.bytes > 0
             )
      select res
      into   str
      from   x
      where  bytes = 0;
    else
      str := lpad('0',(bytes*2)-1,'0,');
    end if;
    output_bytes(str, simple, lbl);
  end;
end;
/
```

APPENDIX B – BITWIZE PACKAGE

```
create or replace package bitwize_maths as
  /* (c) BluShadow @ community.oracle.com
         This code is provided for personal learning purposes only.
         It may not be copied, distributed, reproduced or modified without the
         express permission of the author, except for personal learning purposes.
         Do not breach copyright - you know it is wrong!
  */

  /* Add or Subtract Flag */
  addbits      constant PLS_INTEGER := 0;
  subtractbits constant PLS_INTEGER := 1;

  /* Bit Size Values */
  nibble       constant PLS_INTEGER := 4;
  byte         constant PLS_INTEGER := 8;
  word         constant PLS_INTEGER := 16;
  double_word  constant PLS_INTEGER := 32;

  /* API */
  -- Call this to add or subtract two numbers
  function add_subtract_binary(binary1      in PLS_INTEGER
                              ,binary2      in PLS_INTEGER
                              ,add_subtract in PLS_INTEGER := bitwize_maths.addbits
                              ,bits         in PLS_INTEGER := bitwize_maths.byte
                              ) return PLS_INTEGER;

  -- Call this to multiply two numbers
  function multiply(binary1 in PLS_INTEGER
                   ,binary2 in PLS_INTEGER
                   ,bits    in PLS_INTEGER := bitwize_maths.byte
                   ) return PLS_INTEGER;

  -- Call this to divide binary1 by binary2
  function divide(binary1   in PLS_INTEGER
                 ,binary2   in PLS_INTEGER
                 ,bits      in PLS_INTEGER := bitwize_maths.byte
                 ,remainder in out PLS_INTEGER
                 ) return PLS_INTEGER;


  /* Demo calls of LSL, LSR, ROL and ROR */
  procedure demoLSL(x PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte);
  procedure demoLSR(x PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte);
  procedure demoROL(x PLS_INTEGER, carry_in in PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte);
  procedure demoROR(x PLS_INTEGER, carry_in in PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte);
end;
/

create or replace package body bitwize_maths as
  /* (c) BluShadow @ community.oracle.com
         This code is provided for personal learning purposes only.
         It may not be copied, distributed, reproduced or modified without the
         express permission of the author, except for personal learning purposes.
         Do not breach copyright - you know it is wrong!
  */

  /*
     LOGIC FUNCTIONS

     BitAND is already a built in function, so no need to implement
     BitOR and BitXOR are implemented here.
  */

  -- Implementation of logical OR
  function bitOR (x in PLS_INTEGER
                 ,y in PLS_INTEGER
                 ) return PLS_INTEGER is
  begin
    return (x + y) - bitand(x, y);
  end bitor;

  -- Implementation of logical XOR
  function bitXOR (x in PLS_INTEGER
                  ,y in PLS_INTEGER
```

```
                ) return PLS_INTEGER is
begin
  return bitOR(x, y) - bitand(x, y);
end bitxor;

-- Logical Shift Left (LSL)
function LSL (x    in     PLS_INTEGER
             ,bits in     PLS_INTEGER
             ,carry in out PLS_INTEGER
             ) return PLS_INTEGER is
  p PLS_INTEGER := power(2,bits-1);
begin
  carry := bitAND(x, p)/p;
  return bitAND(x, p-1)*2;
end;

-- Logical Shift Right (LSR)
function LSR (x    in     PLS_INTEGER
             ,bits in     PLS_INTEGER
             ,carry in out PLS_INTEGER
             ) return PLS_INTEGER is
  p PLS_INTEGER := power(2,bits);
begin
  carry := bitAND(x, 1);
  return bitAND(x, p-2)/2;
end;

-- Rotate Left (ROL)
function ROL (x    in     PLS_INTEGER
             ,bits in     PLS_INTEGER
             ,carry in out PLS_INTEGER
             ) return PLS_INTEGER is
  p PLS_INTEGER := power(2,bits-1);
  c PLS_INTEGER := carry;
begin
  carry := bitAND(x, p)/p;
  return (bitAND(x, p-1)*2)+c;
end;

-- Rotate Right (ROR)
function ROR (x    in     PLS_INTEGER
             ,bits in     PLS_INTEGER
             ,carry in out PLS_INTEGER
             ) return PLS_INTEGER is
  p PLS_INTEGER := power(2,bits);
  c PLS_INTEGER := carry;
begin
  carry := bitAND(x, 1);
  return (bitAND(x, p-2)/2)+(c*power(2,bits-1));
end;

/*
   Half Adder
   Accepts two binary bit values (each 0 or 1)
   adds them together to return a result
   and supplying the carry bit back as an OUT parameter
*/
function half_adder(bit1      in  PLS_INTEGER
                   ,bit2      in  PLS_INTEGER
                   ,carry_out out PLS_INTEGER
                   ) return PLS_INTEGER is
  outval PLS_INTEGER;
begin
  outval := bitXOR(bit1,bit2);
  carry_out := bitAND(bit1,bit2);
  return outval;
end;

/*
   Full Adder
   Accepts two binary bit values (each 0 or 1) and an
   incoming carry bit.
   adds the 3 bits together to return a result
   and supplying the carry bit back as an OUT parameter
*/
function full_adder(bit1        in  PLS_INTEGER
                   ,bit2        in  PLS_INTEGER
```

```
                    ,carry_inout in out PLS_INTEGER
                    ) return PLS_INTEGER is
    outval     PLS_INTEGER;
    carry_out1 PLS_INTEGER;
    carry_out2 PLS_INTEGER;
  begin
    dbms_output.put_line('Full        Adder        [IN:        Bit1('||bit1||')        Bit2('||bit2||')
Carry('||carry_inout||')]');
    outval := half_adder(bit1, bit2, carry_out1);
    outval := half_adder(carry_inout, outval, carry_out2);
    carry_inout := bitOR(carry_out1, carry_out2);
    dbms_output.put_line('Full Adder [OUT: Bit('||outval||') Carry('||carry_inout||')]');
    return outval;
  end;

  /*
     Add or Subtract Two Binary Numbers
     Simulates part of Arithmetic Logic Unit (ALU) of processor
     Accepts two binary values, a flag (0 or 1) to indicate whether to add or subtract
     and an indication of the number of bits to process (which also determines the bytes
     that are displayed as output on dbms_output buffer)
  */
  function add_subtract_binary(binary1      in PLS_INTEGER
                              ,binary2      in PLS_INTEGER
                              ,add_subtract in PLS_INTEGER := bitwize_maths.addbits
                              ,bits         in PLS_INTEGER := bitwize_maths.byte
                              ) return PLS_INTEGER is
    result PLS_INTEGER := 0;
    bit1   PLS_INTEGER;
    bit2   PLS_INTEGER;
    carry  PLS_INTEGER := add_subtract;
    p      PLS_INTEGER;
    outval PLS_INTEGER;
  begin
    -- Show the initial bytes
    bytewize.show_binary_bytes(binary1,ceil(bits/8));
    bytewize.show_binary_bytes(binary2,ceil(bits/8));

    for bit in 0 .. bits-1 -- bits are numbered from 0 to 7 etc.
    loop
      p := power(2,bit);                 -- bit value e.g. 1,2,4,8 etc.
      dbms_output.put_line('Bit: '||(bit+1)||' of '||bits||' (value='||p||')');
      bit1 := bitAND(binary1,p)/p;       -- bit value of 0 or 1
      bit2 := bitAND(binary2,p)/p;       -- bit value of 0 or 1
      bit2 := bitXOR(add_subtract,bit2); -- invert bit if subtraction
      outval := full_adder(bit1, bit2, carry);

      -- combine the individual bit results using BITOR to give a final result
      result := bitOR(result, p*outval);
    end loop;
    bytewize.show_binary_bytes(result,ceil(bits/8));
    return result;
  end;

  /*
     Multiply Two Binary Numbers
     Simulates part of Arithmetic Logic Unit (ALU) of processor
     Accepts two binary values, and an indication of the number of bits to process
     (which also determines the bytes that are displayed as output on dbms_output buffer)
  */
  function multiply(binary1 in PLS_INTEGER
                   ,binary2 in PLS_INTEGER
                   ,bits    in PLS_INTEGER := bitwize_maths.byte
                   ) return PLS_INTEGER is
    c PLS_INTEGER := 0; -- carry bit
    r PLS_INTEGER := 0; -- result
    x PLS_INTEGER := binary1;
    y PLS_INTEGER := binary2;
  begin
    bytewize.show_binary_bytes(x, ceil(bits/8)*2, true, 'X = '||x);
    bytewize.show_binary_bytes(y, ceil(bits/8), true, 'Y = '||y);
    dbms_output.put_line('Carry = ['||c||']');
    bytewize.show_binary_bytes(r, ceil(bits/8)*2, true, 'Result = '||r);
    while y > 0
    loop
      dbms_output.put_line('---------- Iteration ---------');
      y := LSR(y, bits, c);
```

```
    bytewize.show_binary_bytes(y, ceil(bits/8), true, 'Y = '||y);
    dbms_output.put_line('Carry = ['||c||']');
    if c = 1 then
      r := r + x; -- we could use our add_subtract function for this
    end if;
    bytewize.show_binary_bytes(r, ceil(bits/8)*2, true, 'Result = '||r);
    x := LSL(x, bits*2, c); -- allow twice as many bits for X
    bytewize.show_binary_bytes(x, ceil(bits/8)*2, true, 'X = '||x);
  end loop;
  return r;
end;

/*
   Divide Two Binary Numbers (binary1 / binary2)
   Simulates part of Arithmetic Logic Unit (ALU) of processor
   Accepts two binary values, and an indication of the number of bits to process
   (which also determines the bytes that are displayed as output on dbms_output buffer)
   Also requires an IN OUT parameter to provide back the remainder of the division
*/
function divide(binary1   in PLS_INTEGER
               ,binary2   in PLS_INTEGER
               ,bits      in PLS_INTEGER := bitwize_maths.byte
               ,remainder in out PLS_INTEGER
               ) return PLS_INTEGER is
  c PLS_INTEGER := 0; -- carry bit
  r PLS_INTEGER := 0; -- result
  b PLS_INTEGER := 0; -- remainder
  x PLS_INTEGER := binary1;
  y PLS_INTEGER := binary2;
begin
  bytewize.show_binary_bytes(x, ceil(bits/8), true, 'X = '||x);
  bytewize.show_binary_bytes(y, ceil(bits/8), true, 'Y = '||y);
  dbms_output.put_line('Carry = ['||c||']');
  bytewize.show_binary_bytes(r, ceil(bits/8), true, 'Result = '||r);
  bytewize.show_binary_bytes(b, ceil(bits/8), true, 'b = '||b);
  for i in 1 .. bits
  loop
    dbms_output.put_line('---------- Iteration: '||i||' ---------');
    r := LSL(r, bits, c);
    bytewize.show_binary_bytes(r, ceil(bits/8), true, 'Result = '||r);
    x := LSL(x, bits, c);
    bytewize.show_binary_bytes(x, ceil(bits/8), true, 'X = '||x);
    dbms_output.put_line('Carry = ['||c||']');
    b := ROL(b, bits, c);
    bytewize.show_binary_bytes(b, ceil(bits/8), true, 'b = '||b);
    dbms_output.put_line('Carry = ['||c||']');
    if b >= y then
      dbms_output.put_line('[b >= y : true]');
      r := r + 1;
      bytewize.show_binary_bytes(r, ceil(bits/8), true, 'Result = '||r);
      b := b - y;
      bytewize.show_binary_bytes(b, ceil(bits/8), true, 'b = '||b);
    else
      dbms_output.put_line('[b >= y : false]');
    end if;
  end loop;
  remainder := b;
  return r;
end;

/* Some simple demo outputs of single LSL, LSR, ROL and ROR */
procedure demoLSL(x PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte) is
  v PLS_INTEGER := x;
  c PLS_INTEGER := 0;
begin
  bytewize.show_binary_bytes(v,ceil(bits/8),true);
  v := LSL(v, bits, c);
  bytewize.show_binary_bytes(v,ceil(bits/8),true);
  dbms_output.put_line('Carry: ['||c||']');
end;

procedure demoLSR(x PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte) is
  v PLS_INTEGER := x;
  c PLS_INTEGER := 0;
begin
  bytewize.show_binary_bytes(v,ceil(bits/8),true);
  v := LSR(v, bits, c);
```

```
    bytewize.show_binary_bytes(v,ceil(bits/8),true);
    dbms_output.put_line('Carry: ['||c||']');
  end;

  procedure demoROL(x PLS_INTEGER, carry_in in PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte)
is
    v PLS_INTEGER := x;
    c PLS_INTEGER := carry_in;
  begin
    bytewize.show_binary_bytes(v,ceil(bits/8),true);
    v := ROL(v, bits, c);
    bytewize.show_binary_bytes(v,ceil(bits/8),true);
    dbms_output.put_line('Carry: ['||c||']');
  end;

  procedure demoROR(x PLS_INTEGER, carry_in in PLS_INTEGER, bits in PLS_INTEGER := bitwize_maths.byte)
is
    v PLS_INTEGER := x;
    c PLS_INTEGER := carry_in;
  begin
    bytewize.show_binary_bytes(v,ceil(bits/8),true);
    v := ROR(v, bits, c);
    bytewize.show_binary_bytes(v,ceil(bits/8),true);
    dbms_output.put_line('Carry: ['||c||']');
  end;
end;
/
```