Packed, Zoned, Binary Math

Posted on [2016-06-06](http://www.askthezoslady.com/packed-zoned-binary-numbers/" \o "02:39) by [TheZosLady](http://www.askthezoslady.com/author/lana-maskellcompuware-com/" \o "View all posts by TheZosLady) — [1 Comment ↓](http://www.askthezoslady.com/packed-zoned-binary-numbers/" \l "comments)

**Mainframe Math: Packed, Zoned, Binary Numbers —**

*Why are there different kinds of numbers?  And how are they different, exactly? Numeric formats on the mainframe (simplified) . . .*

The mainframe can do two basic kinds of math: Decimal and Binary.  Hence the machine recognizes two basic numeric formats: Decimal and Binary.  It has separate machine instructions for each.  Adding two binary integers is a different machine operation from adding two decimal integers.  If you tell your computer program to add a binary number to a packed number, the compiler generates machine code that first converts at least one of the numbers, and then when it has two numbers of the same type it adds them for you.

There is also a displayable, printable type, called Zoned Decimal, which in its unsigned integer form is identical to character format.  You cannot do math with zoned decimal numbers.  If you code a statement in a high level language telling the computer to add two zoned decimal numbers, it might generate an error, but otherwise it works because the compiler generates machine instructions that will first convert the two numbers into another type, and then do the math with the converted copies.

Within Decimal and Binary there are sub-types, such as double-precision and floating point.  These exist mainly to enable representation of, and do math with, very large numbers.  Within displayable numbers there are many possible variations of formatting.  Of course.

For this article, we are going to skip all the variations except the three most common and most basic: Binary (also called Hexadecimal, or Hex), Decimal (called Packed Decimal, or just Packed), and Zoned Decimal (the displayable, printable representation, sometimes also called “Picture”).  To start with we’ll focus on integers.

Generally, packed decimal integers used for mathematical operations can have up to 31 significant digits, but there are limitations: a multiplier or divisor is limited to 15 digits, and, when doing division, the sum of the lengths of the quotient and remainder cannot exceed 31 digits. For practical business purposes, these limits are generally adequate in most countries.

Binary (hex) numbers come in two basic sizes, 4 bytes (a full word, sometimes called a long integer), and 2 bytes (a half word, sometimes called a short integer).

A signed binary integer in a 4 byte field can hold a value up to 2,147,483,647.

The leftmost bit of the leftmost byte is the sign bit.  If the sign bit is zero that means the number is positive.  If the sign bit is one the number is negative.  This is why a full word signed binary integer is sometimes called a 31-bit integer.  Four bytes with 8 bits each should be 32 bits, right?  But no, the number part is only 31 bits, because one of the bits is used for the sign.

A 2-byte (half word) integer can hold a value up to 32,767 if it is defined as a signed integer, or 65,535 if unsigned.

The sign bit in a 2-byte binary integer is still the leftmost bit, but since there are only two bytes, the sign bit is the leftmost bit of the left-hand byte.

Consider the case where you are not using integers, but rather you have an implied decimal point; For example, you’re dealing with dollars and cents. Now you have two positions to the right of an implied (imaginary) decimal point. With two digits used for cents, the maximum number of dollars you can represent is divided by a hundred: $32,767 is no longer possible in a half word; the new limit becomes $327.67, so half word binary math won’t be much use if you’re doing accounting.  $21,474,836.47, that is, twenty-one million and some, would be the limit for full word binary.  Such a choice might be considered to demonstrate pessimism or lack of foresight, or both.  You probably want to choose decimal representations for accounting programs, because decimal representation lets you use larger fields, and hence bigger numbers.

Half word integers are often used for things like loop counters and other smallish arithmetic, because the machine instructions that do half word arithmetic run pretty fast compared to other math.  For the most part binary arithmetic is easier and runs faster than decimal arithmetic.  Also, machine addresses (pointers) are full word binary (hex) integers, so any operation that calculates an offset from an address is quicker if the offset is also a binary integer.  Plus you can fit a bigger number into a smaller field using binary.  However, If you need to do calculations that use reasonably large numbers, for example hundreds of billions in accounting calculations, then you want to use decimal variables and do decimal math.

How are these different types of numbers represented internally – what do they look like?

An unsigned packed decimal number is composed entirely of the hex digits zero through nine.  Two such digits fit in one byte.  So a byte containing hex’12’ would represent unsigned packed decimal twelve.  Two bytes containing hex’9999’ would represent unsigned packed decimal nine thousand nine hundred and ninety-nine.

How is binary different?  You don’t stop counting at nine.  You get to use A for ten, B for eleven, C for twelve, D for thirteen, E for fourteen, and F for fifteen.  It’s base sixteen math (rather than the base ten math that we grew up with).  So in base ten math, when you run out of digits at 9, and you expand leftward into a two-digit number, you write 10 and call it ten.  With base sixteen math, you don’t run out of available digits until you hit F for fifteen; so then when you expand leftward into a two-digit number, you write 10 and call it sixteen.  By the time you get to x’FF’ you’ve got the equivalent of 255 in your two-digit byte, rather than 99.

Why, you may ask, did they do this; In fact, why, having done it, did they stop at F?  Actually it’s pretty simple.  Remember that word binary – it really means you’re dealing in bits, and bits can only be zero (off) or one (on).  On IBM mainframe type machines, there happen to be 8 bits in a byte.  Each half byte, then – each of our digits – has 4 bits in it.  That’s just how the hardware is made.

Yes, a half byte is also named a nibble, but I've never heard even one person actually call it that.  People I've known in reality either say "half byte", or they say "one hex digit".

So we can all see that 0000 should represent zero, and 0001 should be one.  Then what?  Well, 0010 means two; The highest digit you have is 1, and then you have to expand toward the left.  You have to "carry", just like in regular math, except you hit the maximum at 1 rather than 9.  This is base 2 math.

To get three you add one+two, giving you 0011 for three.  Hey, all we have at this point is bit switches, kind of like doing math by turning light switches off and on. (Base 2 math.)  10 isn't ten here, and 10 isn't fifteen; 10 here is two.  So, if 011 is three, and you have to move leftward to count higher, that means 0100 is obviously four.  Eventually you add one (001) and two (010) to four (0100) and you get up to the giddy height of seven (0111).  Lesser machines might stop there, call the bit string 111 seven, and be satisfied with having achieved base 8 math.  Base 8 is called Octal, lesser machines did in fact use it, and personally I found it to be no fun at all.  The word excruciating comes to mind.  Anyway, with the IBM machine people were blessed with a fourth bit to expand into. 1000 became eight, 1001 was nine, 1010 (eight plus two) became A, and so on until all the bits were used up, culminating in 1111 being called F and meaning fifteen.  Base sixteen math, and we call it hexadecimal, affectionately known as hex.  It was pretty easy to see that hex was better than octal, and it was also pretty easy to see that we didn’t need to go any higher — base 16 is quite adequate for ordinary human minds.  So there it is.  It also explains why the word hex is so often used almost interchangeably with the word binary.

And Zoned Decimal?  A byte containing the digit 1 in zoned decimal is represented by hex ‘F1’, which is exactly the same as what it would be as part of a text string (“Our number 1 choice.”)  Think of a printable byte as one character position. The digit 9, when represented as a printable (zoned) decimal number, is hex’F9’.  A number like 123456, if it is unsigned, is hex’F1F2F3F4F5F6’.  (If it has a sign, the sign might be separate, but if the sign is part of the string then the F in the last byte might be some other code to represent the sign.  Conveniently, F is one of the signs for plus, and it also means unsigned.)  You cannot do math with zoned decimal numbers.  If you code a statement in a high level language telling the computer to add two zoned decimal numbers, and if it does not generate an error, the compiler generates machine instructions that will first convert the two numbers into another type and then do the math with the converted copies.

You may be thinking that the left half of each byte is wasted in zoned decimal format.  Well, not wasted exactly: Any printable character will use up one byte; a half byte containing F is no bigger than the same half byte containing zero.  Still, if you are not actually printing the digit at the moment, could you save half the memory by eliminating the F on the left?  Pretty much.

You scrunch the number together, squeezing out the F half-bytes, and you have unsigned packed decimal.  You just need to add a representation of the plus or minus sign to get standard (signed) packed decimal format. The standard is to use the last half byte at the end for the sign, the farthest right position.  This is why decimal numbers are usually set up to allow for an odd number of digits – because memory is allocated in units of bytes, there are two digits in a byte, and the last byte has to contain the sign as the last digit position.

How is the packed decimal sign represented in hex?  The last position is usually a C for plus or a D for minus.  F also means plus, but usually carries the nuance of meaning that the number is defined as unsigned.  Naturally there are some offbeat representations where plus can be F, A, C, or E, like the open spaces on a Treble clef in music, and minus can be either B or D (the two remaining hex letters after F,A,C,E are taken) – hence giving meaning to all the non-decimal digits.  Mostly, when produced by ordinary processes, it’s C for plus, F for unsigned and hence also plus, or D for minus.

So if you have the digit 1 in zoned decimal, as hex’F1’, then after it is fully converted to signed packed decimal the packed decimal byte will be hex’1C’.  Zoned decimal Nine (hex ‘F9’) would convert to packed decimal hex ‘9C’, and zero (hex ‘F0’) becomes hex’0C’.  Minus nine becomes hex ‘9D’, and yes, you can have minus zero as hex ‘0D’.

The mathematical meaning of minus zero is arguable, but some compilers allow it, and in fact the IEEE standard for floating point requires it.  Some machine instructions can also produce it as a result of math involving a negative number and/or overflow.  You care about negative zero mainly because in some operations (which you might easily never encounter), hex’0D’, the minus zero, might give different results from ordinary zero.  A minus zero normally compares equal to ordinary plus zero when doing ordinary decimal comparisons.  Okay, moving on … Zoned decimal 123, hex ‘F1F2F3’, when converted to signed packed decimal will become hex’123C’, and Zoned decimal 4567, hex ‘F4F5F6F7’, when converted to signed packed decimal will become hex’04567C’, with a leading zero added because you need an even number of digits; half bytes have to be paired up so they fill out entire bytes.

Wait, you say, how did the plus or minus look in zoned decimal?

The answer is that there are various formats.

It is possible for the rightmost Zoned Decimal digit to contain the sign in place of that digit’s lefthand “F” (its “zone”), and that is the format generated when a Zoned Decimal number is produced by the UNPACK machine instruction.

The most popular format, for users of high level languages, seems to be when the sign is kept separate and placed at the beginning of the number (the farthest left position). COBOL calls this “SIGN IS LEADING SEPARATE”.

However, many print formats are possible, and you can delve into this topic further by looking at IBM’s Language Reference Manual for whatever language you’re using.  Zoned decimal is essentially a print (or display) format.  High Level Computer Languages facilitate many elaborate editing niceties such as leading blanks or zeroes, insertion of commas and decimal points, currency symbols, and stuff that may never even have occurred to you (or me).

In COBOL, a packed decimal variable is traditionally defined with USAGE IS COMPUTATIONAL-3, Or COMP-3, but it can also be called PACKED-DECIMAL.  A zoned decimal variable is defined with a PICTURE format having USAGE IS DISPLAY.  A binary variable is just called COMP, but it can also be called COMP-4 or BINARY.

In C, a variable that will contain a packed decimal number is just called decimal.  If you are going to use decimal numbers in your C program, the header <decimal.h> should be #included.  A variable that will hold a four byte binary number is called int, or long.  A two byte binary integer is called short.  Typically a number is converted into printable zoned decimal by using some function like sprintf with the %d formatting code.  Input zoned decimal can be treated as character.

PL/I refers to packed decimal numbers as FIXED DECIMAL.  Zoned decimal numbers are defined as having PICTURE values containing nines, e.g. P’99999’ in a simple case.  Binary numbers are called FIXED BINARY, or FIXED BIN, with a four byte binary number being FIXED BINARY(31) and a two byte binary number being called FIXED BINARY(15).

What if you want to use non-integers, that is, you want decimal positions to the right of the decimal?  Dollars and cents, for example?

In most high level languages, you define the number of decimal positions you want when you declare the variable, and for binary numbers and packed decimal numbers, that number of decimal positions is considered to be implied; it just remembers where to put the decimal for you, but the decimal position is not visible when you look at the memory location in a dump or similar display.  For zoned decimal numbers, you can declare the variable (or the print format) in a way that both the implied decimal and a visible decimal occur in the same position.  For example, if (in your chosen language) 999V99 creates an implied decimal position wherever the V is, then you would define an equivalent displayable decimal point as 999V.99, in effect telling the compiler that you want a decimal point to be printed at the same location as the implied decimal.  As previously noted, the limits on the numbers of digits that can be represented or manipulated apply to all the digits in use on both sides of the implied decimal point.

You may have noticed that abends are a bit more common when using packed decimal arithmetic, as compared with binary math.  There are two common ways that decimal arithmetic abends where binary would not.  One occurs when fields are not initialized.  If an uninitialized field contains hex zeroes, and it is defined as binary, that’s valid and some might say lucky.  If the same field of hex zeroes is defined as signed packed decimal, mathematical operations will fail because of the missing sign in the last half byte.  This is a common cause of an 0C7 abend failure in formatted dumps (such as a PL/I program containing an “ON ERROR” unit with a “PUT DATA;” statement).  When the uninitialized fields contain hex zeroes, it might seem that the person using binary variables is lucky, but sometimes uninitialized fields contain leftover data from something else, essentially random trash that happens to be in memory.  In that case decimal instructions usually still abend, and binary mathematical operations do not – they just come up with wrong results, because absolutely any hex value is a valid binary number.  The abend doesn’t look like such bad luck in that situation.  The other common cause for the same problem, besides uninitialized fields, is similar insofar as it means picking up unintended data.  When something goes wrong in a program – maybe a memory overlay, maybe a bad address pointer – an instruction may try to execute using the wrong data.  Again, there is a good chance that decimal arithmetic will fail in such a situation, because of the absence of the sign perhaps, or perhaps because the data contains values other than the digits zero through nine plus the sign.  Binary arithmetic may carry on happily producing wrong answers based on the bogus data values.  Even if you recognize that the output is wrong, it can be difficult to track back to the cause of the problem.  With an immediate 0C7 or other decimal arithmetic abend, you have a better chance of finding the underlying problem with less difficulty.

So there you have it.  Basic mainframe computer math, simplified.  Sort of.