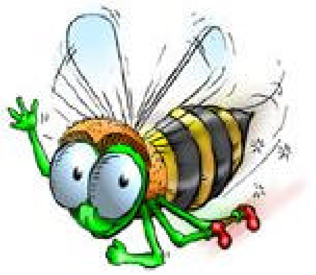
Special Project

PhizzyB COMPUTERS

Understanding Computers

By Clive "Max" Maxfield and Alvin Brown

Bonus Article: Signed and Unsigned Binary Numbers



Welcome to this bonus article, which augments Part 3 of our *PhizzyB Computers* series (Part 3 appeared in the January 1999 issue of *EPE Online*) . These articles are of interest to anyone who wants to know how computers perform their magic, because they use a unique mix of hardware and software to explain how computers work in a fun and interesting way.

This series doesn't assume any great technical knowledge, although an understanding of fundamental electronic concepts would certainly be an advantage. You do need, though, to have had some experience at assembling components onto a printed circuit board. You should also be moderately familiar with using a PC-compatible computer.

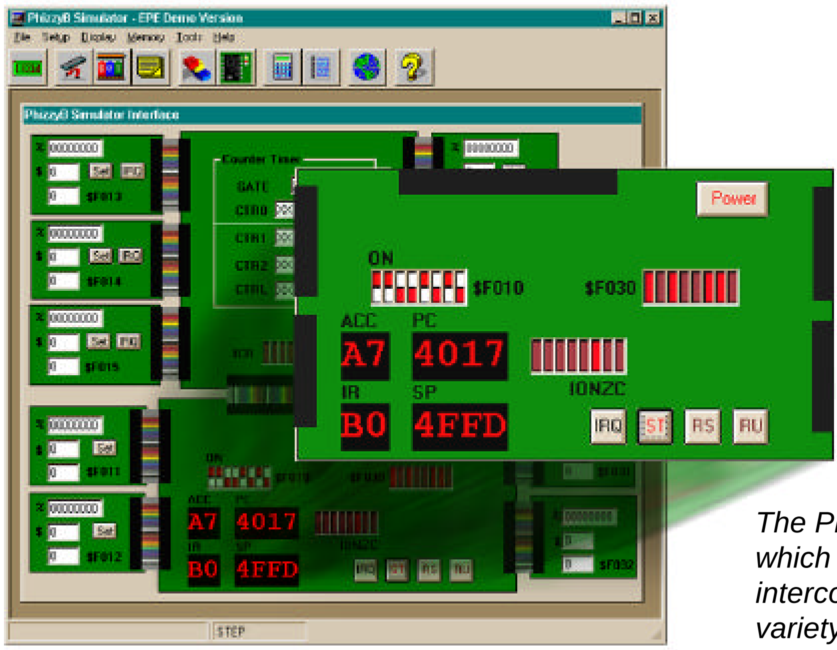
BITS AND TRITS

As we know, today's digital computers are composed of large numbers of simple logic gates. As fate would have it, it's relatively easy to create logic gates that can generate, and respond to, two distinct voltage levels. For this reason, digital computers internally use the binary (base-2) number system.

The term "binary digit" is usually abbreviated to "bit." As we discussed in earlier articles, the *PhizzyB's* databus is 8-bits wide. For a number or reasons, groupings of 8-bits are common in computers, so they are given the special name of "byte," while 4-bit groups are given the name "nybble" (or "nibble"). Thus, two nybbles make a byte, which shows that computer engineers do have a sense of humor (just not a very sophisticated one).

In the early days of computing, everyone pretty much made up their own definitions and standards, so different companies felt free to use the term byte to refer to 6-bit, 7-bit, 8-bit, and even 9-bit "chunks" of data. More recently, however, "byte" has come to be universally accepted as referring only to an 8-bit value.

Also for your interest, some experimental work has been performed with tertiary logic, which refers to logic gates that can generate, and respond to, three distinct voltage levels. The digits used in the tertiary (base-3) number system are called "trits." Working with tertiary systems makes one's brain ache, so you can be thankful that we're not going to discuss these concepts any further here.


PhizzyB Simulator - EPE Demo Version
File Setup Display Memory Tools Help
PhizzyB Simulator Interface
Power
ON
$F010
$F030
ACC
PC
A7
4017
IR
SP
B0
4FFD
IONZC
IRQ
ST
RS
RU
STEP

The PhizzyB Simulator interface, which simulates a real PhizzyB interconnected by ribbon cables to a variety of expansion boards.

Suffice it to say that all of today's existing systems are based on binary logic and, for a number of reasons, we aren't going to see commercial systems based on tertiary logic any time soon, for which we can all be truly thankful. (Logic gates, alternative number systems, and a rather good seafood gumbo recipe are presented in detail in our book *Bebop to the Boolean Boogie (An Unconventional Guide to Electronics)*, which is available from the ***EPE Online Store***.

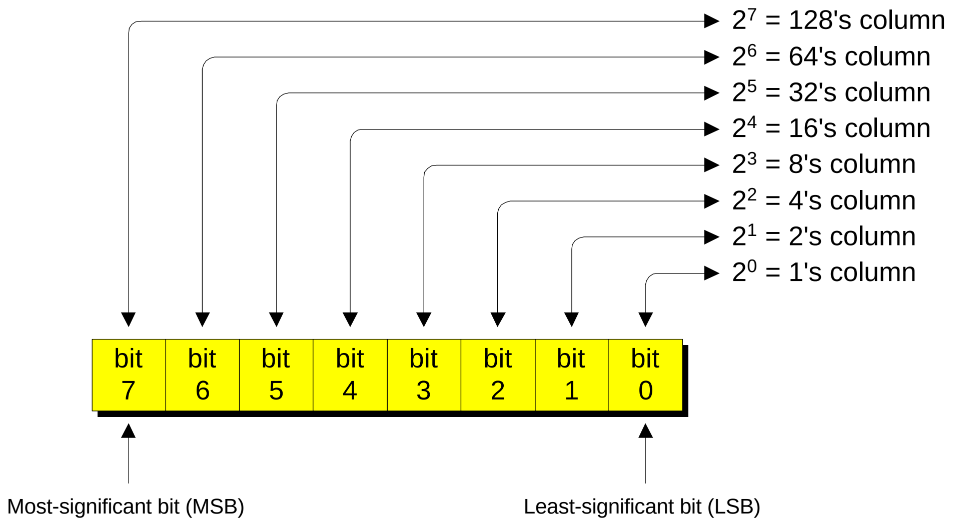


Fig.1. The PhizzyB's 8-bit bus can be used to represent unsigned binary numbers.

UNSIGNED BINARY NUMBERS

A single bit can be used to represent two values: 0 and 1. Two bits can be used to represent 2^2 = 2 x 2 = 4 values: 00, 01, 10, and 11. Three bits can be used to represent 2^3 = 2 x 2 x 2 = 8 values, and so forth. Thus, the *PhizzyB's* 8-bit databus can be used to represent 2^8 = 256 different values.

Many people find working with binary numbers a little difficult at first, but it's really rather easy. For example, when we were young we all had to learn our "twelve times tables," which took quite a lot of effort when you come to look back on it. In the case of unsigned binary numbers, the multiplication table is as follows:

$0 \times 0 = 0$
$0 \times 1 = 0$
$1 \times 0 = 0$
$1 \times 1 = 1$

And that's it. You've essentially learned the entire binary multiplication table. Phew! Perhaps you'd better have a cup of hot, sweet tea to steady your nerves!

One interesting point to ponder is that these values are really just different patterns of 0s and 1s, and we can use these patterns to represent anything we wish, including letters of the alphabet, colors of the spectrum, musical notes, and so forth.

Of course we often want to use these patterns to represent numbers. In this case there are a variety of schemes we might use, and one common technique is to use our 256 patterns of 0s and 1s to represent positive values in the range 0 to 255:

Binary	Decimal
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
00000110	6
00000111	7
00001000	8
:	:
11111011	251
11111100	252
11111101	253
11111110	254
11111111	255

Due to the fact that these values are all positive, we refer to this form of representation as "unsigned binary numbers." Another way to view an unsigned binary number is that each column has a different "weight" associated with it, where these "weights" are different powers of two. So bit 0 represents the one's column, bit 1 represents the two's column, bit 2 represents the four's column, and so forth (Fig.1).

Thus, it's easy to convert a binary value into its decimal equivalent by simply combining each bit with its associated column weight. For example, an unsigned binary value of 11001010 equates to $(1 \times 128) + (1 \times 64) + (0 \times 32) + (0 \times 16) + (1 \times 8) + (0 \times 4) + (1 \times 2) + (0 \times 1)$ = 202 in decimal. Note that the left-most bit is referred to as the *most-significant bit (MSB)*, because it represents the largest (most significant) value. Similarly, the right-most bit is referred to as the *least-significant bit (LSB)*, because ...... well we'll leave you to work this one out for yourself.

ADDING BINARY NUMBERS

Before we tackle binary, let's first remind ourselves as to how we add decimal numbers together, for example:

```
 635
+272
----
=907
```

We commence with the least-significant digits (the right-most column) and add the 5 and 2 to give 7, so no major surprises here.

Next we add the 3 and 7 from the next column, which equals 10, and which we think of as a result of "0" with a "1" to carry forward to the next column.

So when we get to the third column we add the 6 and 2 *and* the 1 carried from the previous column to give a result of 9.

Well things work almost exactly the same in binary, in that we start with the right-most column and work our way to the left. The only difference is that we have fewer digits to play with. To see how this goes, let's look at a simple example:

```
 BBBBBBBB
 iiiiiiii
 tttttttt
 76543210

 00110010  (50)
+00110100  (52)
---------
=01100110 (102)
```

First we add 0 + 0 in the bit 0 position (the right-most column) to give 0. Next we add 1 + 0 in the bit 1 column to give 1, followed by 0 + 1 in the bit 2 column to give 1, followed in turn by 0 + 0 in the bit 3 column to give 0 again. Things start to get a little more interesting when we add 1 + 1 in the bit 4 column, because this gives us 10 in binary (2 in decimal), which we can think of as a result of "0" with a "1" to carry forward to the next column. Similarly, when we come to the bit 5 column, we now have 1 + 1 + 1 (where the last 1 was the carry from the previous column). This gives us 11 in binary (3 in decimal), which we can think of as a result of "0" with a "1" to carry forward to the bit 6 column.

Although this may seem tricky the first time you see it, the thing to remember is that binary mathematics is really quite simple, and with just a little practice you'll be a "black belt."

EXPERIMENT 1:

Additions and the carry flag

Start up your *PhizzyB Simulator*, activate the assembler, and enter the program shown in listing 1.

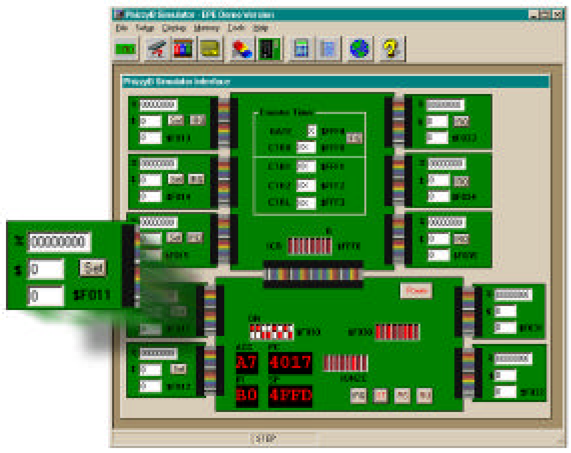
As we see, this is really a rather simple program. Following our constant label declarations, the first thing we do is to load the accumulator with whatever value is on the 8-bit switch device connected to the input port at address $F010. Next we add the whatever value is being presented to the external input port at address $F011 to the current contents of the accumulator. Then we store the result to the output port driving the 8-bit LED device at address $F030 and also to the external output port at address $F031. Finally, we jump back to the label LOOP and do the whole thing all over again.

Use the assembler's *File -> Save As* command to save this file as *baexp1.asm*, then use the *File -> Assemble* command to generate the corresponding

Listing 1

```
INPORT0:    .EQU  $F010      #Assign a label to input port $F010
INPORT1:    .EQU  $F011      #Assign a label to input port $F011
OUTPORT0:   .EQU  $F030      #Assign a label to output port $F030
OUTPORT1:   .EQU  $F031      #Assign a label to output port $F031

            .ORG  $4000      #Set start address to $4000
LOOP:        LDA [INPORT0]   #Load accumulator from I/P port 0
             ADD [INPORT1]   #Add value from I/P port 1
             STA [OUTPORT0]  #Store result to O/P port 0
             STA [OUTPORT1]  #Store same value to O/P port 1
             JMP [LOOP]      #Jump to LOOP and do it all again
            .END
```



Generic input device at port $F011

baexp1.ram machine code file. Now click the simulator's *Power* button to power everything up, use the *Memory -> Load RAM* command to load *baexp1.ram* into the simulator's memory, and click the *Ru* ("Run") button to set the program running.

Your program is now racing around doing its funky thing. The reason nothing appears to be happening is that you're currently adding 0 to 0 to give 0, which is intrinsically somewhat boring.

Click the right-most switch on the 8-bit switch device. The resulting pattern of 00000001 on the switches corresponds to a value of 1 in decimal. (If the red part of the switch is towards the bottom of the screen, this corresponds to a logic 0 or *off*. Similarly, if the red part of the switch is towards the top of the screen, this corresponds to a logic 1 or *on*.)

Your program is now adding this value of 1 from the switch device to a value of 0 from the external input port at address $F011, so the answer displayed on the 8-bit LED and the external output port is, not surprisingly, 00000001 in binary or 1 in decimal.

Use your mouse to select the binary field on the external input device (the binary field is the one at the top, and is indicated by the % character).

Enter the binary value 00000001 and click the device's *Set* button, which formally presents this value to the simulator's input port.

So your program is now adding the 00000001 value on the switches to the 00000001 value on the external port, giving a grand result of 00000010 (or two in decimal).

Feel free to experiment with other values, but for the moment only apply values in the range 00000000 to 01111111 (0 to 127 in decimal) to both the 8-bit switch and the external input device. Adding these values together will return results in the range 00000000 to 11111110 (0 to 254 in decimal).

Now set the 8-bit switch to 11101010 (234 in decimal) and set the binary field of the external input port to 10100001 (161 in decimal) – don't forget to click the port's *Set* button. Eeeeek! Look at the result displayed on the 8-bit LEDs and the external output port: 10001011 (139 in decimal).

How can this be? Surely adding 234 to 161 should result in 395, not the 139 indicated by the simulator!

The problem is that our 8-bit accumulator can only represent unsigned binary values in the range 00000000 to 11111111 (0 to 255 in decimal), so it simply can't hold a value of 395. But all is not lost, because the CPU has a status flag called the C ("Carry") flag, which is used to indicate just such an eventuality.

Click the simulator's *St* ("Step") button to drop down into step mode, and note the values on the status register (the one annotated with "IONCZ"). Observe that both the N ("Negative") and C ("Carry") flags are active (lit).

From last month's article we know that the N flag is simply a copy of the most-significant bit (bit 7) in the accumulator (more on this later). But it's the C flag that's of interest here, because the fact that it is lit reflects the fact that a carry condition has occurred.

What does this mean? Well, in the case of an ADD instruction, the carry flag can be considered to be an extra 1-bit storage location that is attached to the end of the accumulator (Fig.2).

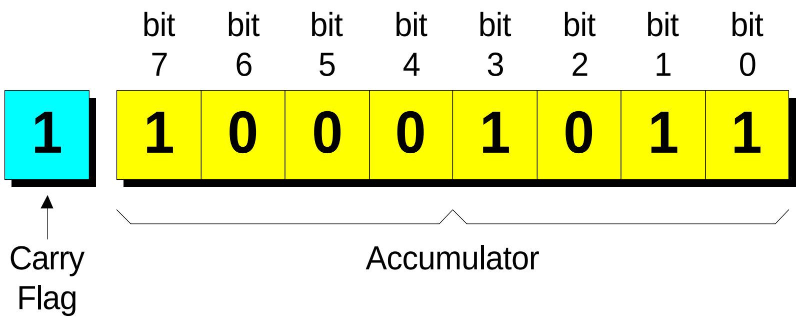


Fig.2. In the case of an ADD, the carry flag can be viewed as an extra bit attached to the end of the accumulator.

One way to visualize this is that the carry flag represents a 2^8 = 256's column, in which case the 9-bit unsigned binary value 110001011 does indeed equate to 395 in decimal.

Another way to view this is that the carry flag represents an error condition. That is, a 0 in the carry flag following an ADD instruction indicates that the ADD was performed without any problems, while a 1 in the carry flag indicates that the result from the ADD was too big to fit in the accumulator.

In the same way that we have conditional jump instructions for the N and Z flags, we also have the JC ("*jump if carry*") and JNC ("*jump if not carry*") instructions, which can be used to test the state of the carry flag and then jump (or not) accordingly.

In addition to the ADD instruction, which adds a new value to whatever value is currently stored in the accumulator, we also have an ADDC ("*add with carry*"). Like the ADD, the ADDC also adds a new value to whatever value is currently stored in the accumulator, but it also adds whatever value is currently held in the carry flag (the value in the carry flag is then overwritten with the result from the ADDC). The ADDC instruction is of use in *multi-byte additions* (see also the *Further Reading* section at the end of this article).

If you have built a real *PhizzyB* (as described in the November 1998 issue), and also the simple input and output devices discussed by Alan Winstanley in the December 1998 issue, then this would be a good time to click the *Rs* ("Reset") button on the simulator, download your *baexp1.ram* machine code file to the *PhizzyB*, and verify that this experiment (and the carry flag) work the same way in the real world.

VERY IMPORTANT

Once you've finished playing with the real *PhizzyB*, it's very important that you reset it and then close down the *PBLink* utility. This is because the *PBLink* spends a lot of time bouncing signals back and forth to the *PhizzyB*, which can significantly impact the speed of the *PhizzyB Simulator*.

EXPERIMENT 2:

Subtractions and the borrow flag

The good news is that – like every other computer on the face of the planet -- the *PhizzyB* doesn't actually have a borrow flag. The bad news is that the carry flag doubles as a borrow in the case of subtractions.

Use the assembler to open your original *baexp1.asm* file, save this out as *baexp2.asm*, change the ADD instruction to a SUB (don't forget to change the comment), and assemble this program to generate a *baexp2.ram* machine code file. It should come as no great surprise to learn that our program now loads the accumulator with whatever value is on the 8-bit switches and then subtracts whatever value is on the external input port.

Load the contents of *baexp2.ram* into the simulator's memory and then set this program running. Now set the 8-bit switches to 01101010 (106 in decimal) and set the binary field on the external input device to 00100001 (33 in decimal) — don't forget to click the device's *Set* button. Note that the resulting values shown on the 8-bit LED display and the external output port are 01001001 (73 in decimal). This is obviously what we'd expect when subtracting 33 from 106, so there are aren't any surprises here, or are there ......?

Click the simulator's *St* ("Step") button, which drops into the step mode and activates all of the displays. Observe that the status register's C ("Carry") flag is lit indicating a 1 value. Doesn't this indicate a problem? Well actually it doesn't, because the carry flag is used to represent "borrow" conditions in the case of subtraction operations, and borrows work in the opposite way to carries. Thus, a borrow=1 indicates that our SUB instruction was performed without any problems.

Now click the simulator's *Ru* ("Run") button to return us to the run mode, then exchange the values on the 8-bit switches and the external input device. That is, set the 8-bit switches to a value of 00100001 (33 in decimal) and the external input device to a value of 01101010 (106 in decimal). So our program is now attempting to subtract 106 from 33. What do you expect the result to be? Well, in the real world we might hope for something vaguely in the region of -73, but instead our 8-bit LED display and external output device are unashamedly displaying 10110111 (183 in decimal).

So what's the problem? Well earlier we stated that unsigned binary numbers can only be used to represent positive values. But we're currently trying to subtract a "bigger" value from a "smaller" one, and this generates a negative result that we simply can't represent using our unsigned format. Just to prove that this is the problem, click the simulator's *St* ("Step") button to return us to the step mode, and observe that the carry flag (acting as a borrow) is off, thereby indicating that there was a problem with the result from this subtraction. (Once again we'll ignore the N flag for the nonce.) When you're ready to proceed, click the simulator's Rs ("Reset") button.

As before, if you have a real *PhizzyB*, now would be a good time to repeat this experiment in the real world.

The PhizzyB supports both SUB and SUBC instructions. In addition to subtracting a byte from the current contents in the accumulator, the SUBC also subtracts the current value in the carry flag.

Note that as the carry flag is acting as a borrow in the case of a subtraction, the SUBC should more properly be called SUBB. Different microprocessor instruction sets may use either of these mnemonics.

SIGNED BINARY NUMBERS

As we've just discovered to our cost, the fact that unsigned binary numbers can only represent positive values means that we can't subtract a bigger value from a smaller one if we're using this format. This is obviously somewhat limiting, so we need a cunning solution.

When we're using decimal numbers, we can indicate positive or negative values by simply prefixing the number with a sign; for example, +33 versus -33 (by convention, a number without a sign is assumed to be positive). This is known as a "sign-magnitude" representation, because it comprised both a sign and a magnitude (value).

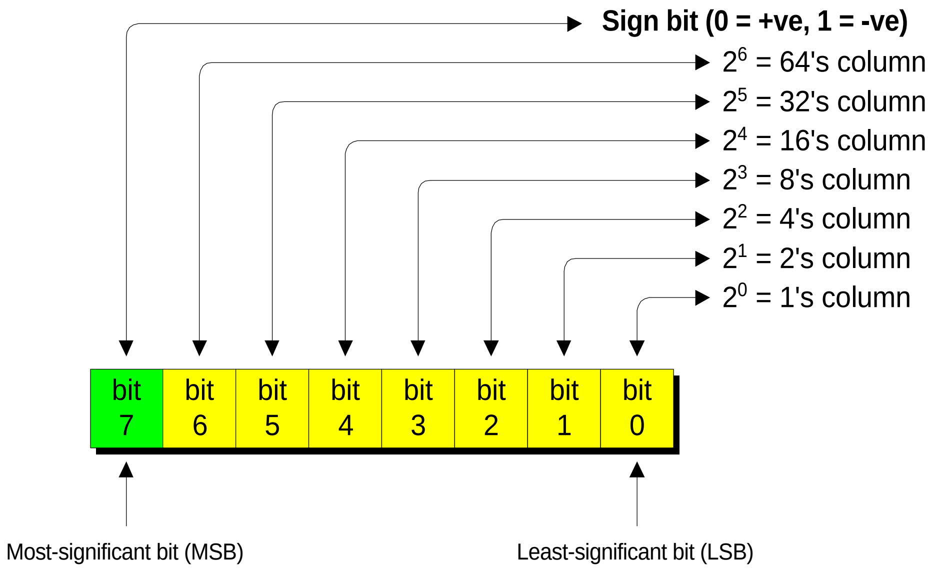


Fig.3. In a sign-magnitude representation, the MSB could be used to represent the sign of the number

So one possibility would be to use one of our bits to indicate the sign of a number. In the case of the PhizzyB's 8-bit databus, for example, we could use bit 7 to indicate the sign of a value (Fig.3).

NOTE that we DO NOT use this scheme — this portion of our discussion is intended only to explore different possibilities. In this case, the most-significant bit would be called the "sign bit," while the remaining bits could be used to represent values in the range 0000000 to 1111111 (0 to 127 in decimal). So if we were to decide that a 0 in the sign bit indicated positive values and a 1 indicated negative values, then 00000001 would represent +1, while 10000001 would represent -1.

The main advantage of this sign-magnitude format is that it mirrors the way we work with decimal numbers, which makes it intrinsically easy to understand. But there are significant disadvantages, including the fact that this format supports both +0 and -0 (which can be a real pain), and also that we would have to treat sign bits in a different manner to the other bits when performing calculations.

Suffice it to say that some clever computing rapscallions put their heads together and came up with another scheme that we call … wait for it, wait for it … *"signed binary numbers."* In this case, the most significant bit of the *PhizzyB's* 8-bit databus doesn't simply represent a sign, but instead it represents a negative quantity of –128 (Fig.4).

This means that the least-significant 7 bits are used to represent positive values in the range 0000000 to 1111111 (0 to 127 in decimal).

However, a 0 in the sign bit equates to (0 x –128) = 0, while a 1 in the sign bit equates to (1 x –128) = -128. Thus, the final value of a signed binary

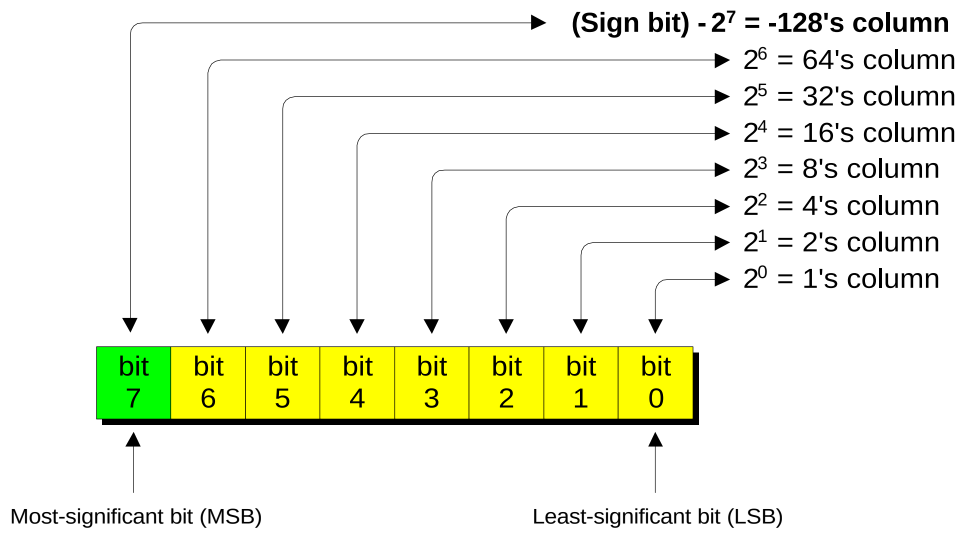


Fig.4. In the case of signed binary numbers, the MSB actually represents a negative quantity!

number is calculated by adding the value represented by the sign bit to the value of the other bits. For example, consider the signed binary value 11100101 shown in Fig.5.

The 1 in the sign bit represents -128, while the remaining bits represent +101, so the end result is -128 + 101 = -27. This means that an 8-bit signed binary number can represent values in the range -128 to +127 as follows:

```
00000000      (0)
00000001      (1)
00000010      (2)
00000011      (3)
00000100      (4)
    :
01111110    (126)
01111111    (127)
10000000   (-128)
10000001   (-127)
10000010   (-126)
10000011   (-125)
    :
11111100     (-4)
11111101     (-3)
11111110     (-2)
11111111     (-1)
```

TWO'S COMPLEMENTS

"Good grief!" you cry, *"What raving drongo came up with this scheme?"* Well admittedly the signed binary representation does seem unduly complex the first time you run into it, but there is reason behind this apparent madness.

First of all, every number system has what is known as its "radix complement" (where "radix" comes from the Latin, meaning "root"). In the case of the binary (base-2) number system, the radix complement is called the *two's complement*. A simple way to generate the two's complement of a binary number is to commence with the least-significant bit, copy every bit up to and including the first 1, and to then invert the remaining bits by exchanging 0s for 1s and vice versa (Fig.6).

As we see, the two's complement of a number is the negative of that number; that is, the two's complement of 01100100 (+100 in decimal) is 10011100 (-100 in decimal). Similarly, if we were to take the two's complement of 10011100 (-100 in decimal), we'd end up back where we started with 01100100 (+100 in decimal).

One huge advantage of signed binary numbers (which are also known as "two's complement numbers") is that if we wish to add two values together, we can simply perform a standard binary addition, irrespective of whether or not the values represent positive or negative quantities.

For example, consider the following four permutations of adding positive and negative values together:

```
   00111001   (+57)
 + 00011110   (+30)
 -----------
 = 01010111   (+87)

   00111001   (+57)
 + 11100010   (-30)
 -----------
 = 00011011   (+27)
```

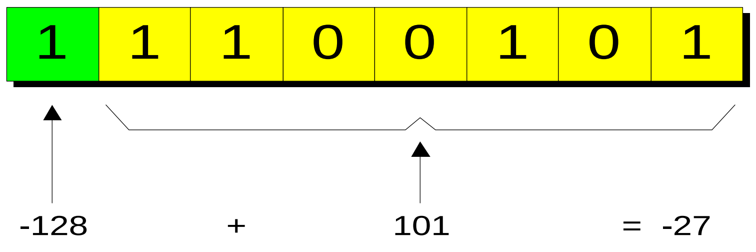


Fig.5. The final value of a signed binary number is calculated by adding the negative (-ve) value of the sign bit to the positive (+ve) value of the other bits.

```
  11000111  (-57)
+ 00011110  (+30)
-----------
= 11100101  (-27)
```

```
  11000111  (-57)
+ 11100010  (-30)
-----------
= 10101001  (-87)
```

In fact life gets even sweeter, because using this signed binary format means that we never actually have to perform a subtraction. For example, we know that 10 - 3 = 7 in decimal (or at least it did when I was a younger man), and that another way to write this is 10 + (-3).

Similarly, if we wish to *subtract* a binary value B from another value A, all we actually have to do is to take the two's complement of B and *add* it to A. In fact this is the way the CPU actually performs subtractions in hardware — pretty cool!

Note that radix complements, diminished radix complements, and complement techniques in general are discussed in greater detail in our book *Bebop BYTES Back (An Unconventional Guide to Computers)*, which is also available from the *EPE Online Store* at **www.epemag.com**.

EXPERIMENT 3:

The overflow flag

One final advantage of the *signed* binary format is that it allows us to perform addition and subtraction operations in exactly the same way as for *unsigned* binary numbers. This means that at any particular time, we [the programmers] can decide to regard a value as being either signed or unsigned — it all depends what we want to use that value for at the time.

Of course there is one fly in the ointment, but there always is isn't there? As we already know, the carry flag is used to show when the result from an operation exceeds the maximum value that can be stored inside the accumulator. But the way the carry flag works is based on the assumption that we're playing with unsigned values. For example, consider the following operation assuming unsigned values:

```
  01010011  (+83)
+ 00111110  (+62)
-----------
= 10010001 (+145)
```

Obviously the carry flag isn't going to be set in this instance, because an unsigned 8-bit value can represent numbers in the range 0 to 255, which means that the result of +145 is perfectly acceptable. But now consider what happens if we assume signed values:

```
  01010011  (+83)
+ 00111110  (+62)
-----------
= 10010001 (-111)
```

The bit patterns are identical in both cases, but now we have a problem, because it appears that 82 + 62 = -111. The problem of course is that a signed 8-bit value can only represent numbers in the range -128 to +127, and adding 82 and 62 together gives a result that exceeds this range.

The solution used by computer designers is an additional status bit called the O ("Overflow") flag. This flag becomes active if the result from an operation on signed binary values exceeds the permitted range (the overflow flag is generated as an exclusive-OR (XOR) of any carry into and out of the most-significant bit of the result).

The important point to understand is that the CPU doesn't really have a clue what we're doing here. That is, the CPU doesn't know whether we're regarding the values as being signed or unsigned. The cunning part of all of this is that the Carry flag assumes we're

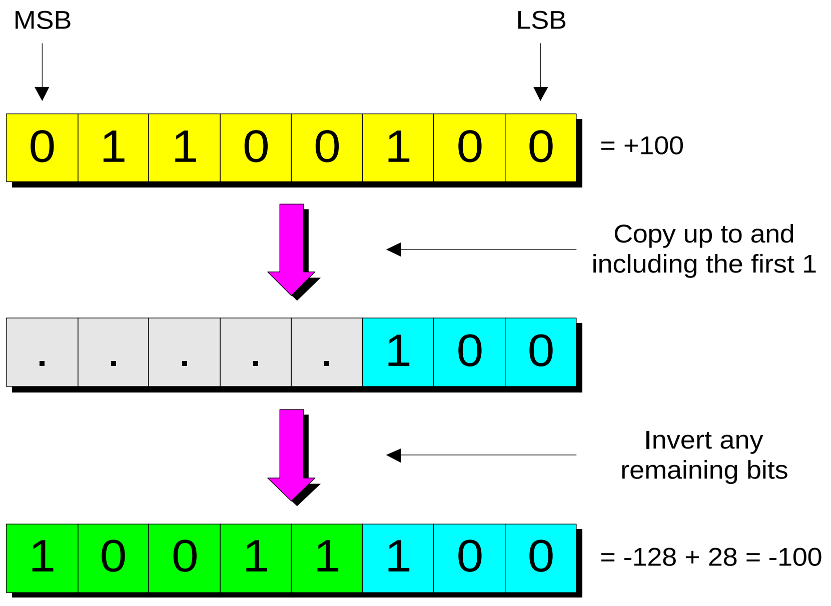


Fig.6. A simple way to generate a two's complement.

using unsigned binary numbers, while the Overflow flag assumes that the numbers are signed. Thus, it's up to us [the programmers] to decide how we wish to treat the result from an operation. Speaking of which, we also have JO ("*jump if overflow*") and JNO ("*jump if not overflow*") conditional jump instructions available to us.

In order to see the Overflow flag work, use the assembler to open the *baexp1.asm* file we created earlier, save this file as *baexp3.asm*, and assemble it to generate the corresponding *baexp3.ram* file. As you will recall, this program simply added the value on the 8-bit switches to the value on the external input port at address $F011.

Load the *baexp3.ram* file into the simulator's memory and set this program running. Now set the 8-bit switches to a value of 01010011 (83 in decimal) and the external input device to a value of 00111110 (62 in decimal) — don't forget to click the device's *Set* button.

Observe that the 8-bit LEDs and the external output device display a value of 10010001 -- note specially that the decimal field of the output display shows a value of 145, because these displays always assume unsigned values (we had to choose one or the other, and we decided to use unsigned values). However, if you now click the St ("Step") button to drop the simulator into its step mode, you'll see that the O ("Overflow") status flag is lit up, thereby indicating that if we're considering these values to be signed, then an overflow condition has occurred.

Also note that the N ("Negative") flag is lit. It now becomes clear why we call this flag the negative flag, because it's a copy of the most-significant bit in the accumulator. Thus, if we're considering the value in the accumulator to be signed, then the N flag is a copy of the sign bit, and a 1 in the sign bit (which causes the N flag to light up) indicates a negative value. Phew!

OK, before we move on, reset the simulator, download this program to your *PhizzyB* (if you have one), and repeat this experiment in the real world (don't forget to reset your *PhizzyB* when you've finished).

THE SHIFT INSTRUCTIONS

In the January 1999 issue of *EPE Online* we introduced the *PhizzyB's* rotate and shift instructions. At that time, we noted that the *PhizzyB* supports two shift instructions called SHL ("*shift left*") and SHR ("*shift right*").

First let's consider the SHL, which shifts the contents of the accumulator one bit to the left and shifts a 0 into the LSB of the accumulator (Figure 7a).

In our January article we noted that as binary is a base-2 number system, shifting a value one bit to the left has the same effect as multiplying it by two. For example, if we load the accumulator with 00001111 (15 in decimal) and execute a SHL, the accumulator ends up containing 00011110 (30 in decimal). One thing we didn't mention, however, is that this works for both signed and unsigned binary numbers. Also, in the case of signed numbers, it works for both positive and negative values. For example (assuming that we're playing with signed binary numbers), if we were to load the accumulator with say 11100110 (-26 in decimal), then shifting this value one bit to the left would give 11001100 (-52 in decimal). Note that this only works until the value in the sign bit changes from a 1 to a 0 or vice versa, at which point we've just exceeded the range of signed binary values that this 8-bit field can represent.

Now turn your attention to the SHR instruction, which shifts the contents of the accumulator one bit to the right (Fig.7b). Once again, due to the fact

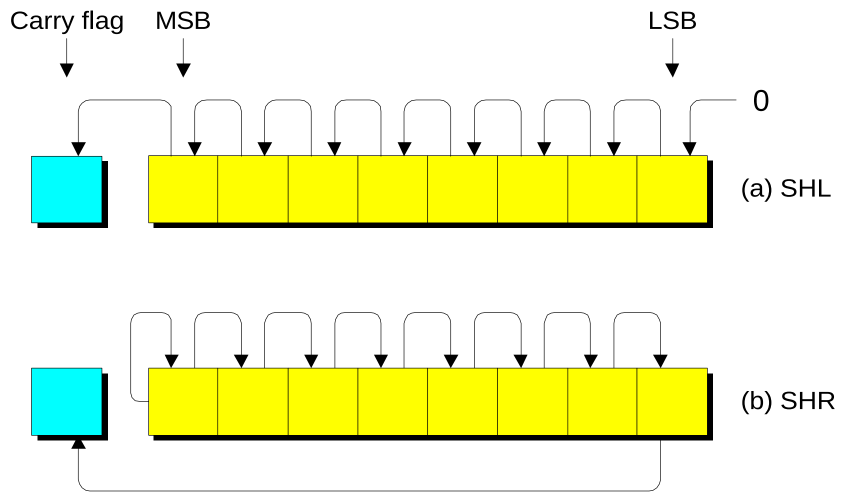


Fig.7. The (a) SHL and (b) SHR instructions.

that binary is a base-2 number system, shifting a value one bit to the right has the same effect as dividing it by two.

For example, if we were to load the accumulator with 00011110 (30 in decimal), shifting it one bit to the right would result in it containing 00001111 (15 in decimal). Note that a further shift to the right would result in 00000111 (7 in decimal).

Obviously 15/2 equals 7.5 in the real world, but we don't have any way to represent the remainder from this division using the unsigned and signed representations we've discussed thus far, so the result is automatically truncated to the integer portion only. (However, the carry flag will now contain the 1 that "fell off the end," and we could use this to detect that this condition had occurred.)

However, the really interesting point about our SHR instruction is the fact that we shift a copy of the MSB of the accumulator back into itself. This is known as an "arithmetic shift right," because it's intended to support arithmetic operations on signed binary numbers. For example, suppose that the accumulator originally contained 11001100 (-52 in decimal). In this case, shifting the accumulator one bit to the right using the *PhizzyB's* SHR instruction will result in 11100110 (-26 in decimal), and this only works because the accumulator's MSB is copied back into itself.

Some CPUs support another type of shift right called a "logical shift right," which means that the shift causes the MSB to be loaded with a 0 (some CPUs supports both logical and arithmetic shifts). In the case of the *PhizzyB* we wanted to keep the instruction set relatively small and easy to understand, so we decided to offer an arithmetic shift right only. Note, however, that it's easy to replicate the effects of a logical shift right if you so desire, because all you have to do is to follow the arithmetic shift right with an AND %01111111 instruction. This "masking operation" will force the MSB of the accumulator to contain 0 (while leaving the remaining bits in the accumulator untouched).

SHR BUG!

Sad to relate, reader Don McBrien in Ireland has rooted out a bug in the *PhizzyB Simulator*. Much to our embarrassment, Don discovered that instead of performing an "arithmetic shift" (in which the MSB is copied back into itself), the simulator's **SHR** actually performs a "logical shift," which means that a 0 is shifted into the MSB.

Happily, the real *PhizzyB* does perform an arithmetic shift as documented above, but this doesn't alter the fact that the actions of the simulator and the *PhizzyB* differ in this regard. However all is not lost, because we have a cunning plan (we'll call this "Plan A" so that no one gets confused), which is documented at the *PhizzyB* web site:

www.maxmon.com/phizzyb

FURTHER READING

Earlier in this article we mentioned *"multi-byte additions."* What did we mean by this? Well one of the problems we face is that the range of numbers we can represent with the *PhizzyB's* 8-bit ACC is somewhat restricted (0 to 255 for unsigned numbers or –128 to +127 for signed numbers).

One solution to this is to use multiple bytes to represent each value. For example, if we decided to use two bytes to represent a number, we now have 16 bits which can represent 2^{16} different patterns of 0s and 1s. Thus, our 2-byte field could represent values in the range 0 to 65,535 for unsigned numbers or –32,768 to +32,767 for signed numbers.

However, we now have a new problem, which is that the *PhizzyB's* data bus, accumulator, and instruction set are geared up to process 1-byte chunks of data. Thus, we have to perform multi-byte additions byte-by-byte. Similarly for subtractions and other mathematical operations.

If you use the *File -> Open* command on the *PhizzyB's* assembler, you'll see a set of files called *add16.asm*, *sub16.asm*, and so forth. These files contain the source code for multi-byte arithmetic operations.

In fact, we'll be using one or two of these subroutines in our *PhizzyB* article in the February 1999 issue of *EPE Online*. In the meantime, if you want to discover more about these little rascals, you'll find that they are discussed in detail in our book *Bebop BYTES Back (An Unconventional Guide to Computers)*, which can be purchased from the *EPE Online Store* at **www.epemag.com**.

COMPACT
disc
Interactive
Maxfield & Montrose Interactive
The PhizzyB™ Simulator
(c) 1998 Maxfield & Montrose Interactive Inc
All Rights Reserved