

Math in MIPS

Adding and Subtracting Binary Numbers

Adding two binary numbers together is very similar to the method used with decimal numbers, except simpler. When you add two binary numbers together, you have to carry if the value is greater than 1 (the values would be 10 or 11). This is the same as when two decimal numbers are added and you have to carry if the value is greater than 9 (the values between 10 and 19). Examples of adding decimal and binary numbers are presented below:

Binary		Decimal	
$\begin{array}{r} \text{Carry bit} \\ 11 \\ 10010 \\ + 00111 \\ \hline 11001 \end{array}$	$\begin{array}{r} 11111 \\ 10011 \\ + 11101 \\ \hline 110000 \end{array}$	$\begin{array}{r} \text{Carry digit} \\ 11 \\ 50070 \\ + 00982 \\ \hline 51052 \end{array}$	$\begin{array}{r} 11111 \\ 30096 \\ + 89904 \\ \hline 120000 \end{array}$

Subtracting a binary number from another binary number also bears an uncanny resemblance to the way it's done in decimal.

Binary		Decimal	
$\begin{array}{r} \text{Borrow bit} \\ 1110 \\ 00111 \\ - 00111 \\ \hline 01011 \end{array}$	$\begin{array}{r} -0 \\ 10011 \\ - 11101 \\ \hline -01010 \end{array}$	$\begin{array}{r} \text{Borrow bit} \\ 9910 \\ 00957 \\ - 00957 \\ \hline 09056 \end{array}$	$\begin{array}{r} -6 \\ 60423 \\ - 79103 \\ \hline -19280 \end{array}$

Two's Complement

Two's complement is the representation used in most modern microprocessors for positive and negative numbers (non-floating point). It makes addition and subtraction easy to perform because of it's generally easy to convert from positive to negative numbers and visa versa, as well as add positive and negative numbers together. In fact, subtraction is done by converting the subtrahend to a negative number and then adding them together.

There are two ways of coming up with a two's complement number from a negative binary number. Notice that the end result looks positive, although it can be used as though it represented a negative number. Something similar can also be done using decimal numbers.

Binary		Decimal	
$\begin{array}{r} -00000110 \\ \downarrow \text{Invert} \\ 11111001 \\ + 1 \\ \hline 11111010 \end{array}$	$\begin{array}{r} 10000000 \\ - 00000110 \\ \hline 011111010 \end{array}$	$\begin{array}{r} -00000470 \\ \downarrow \text{Invert} \\ 99999529 \\ + 1 \\ \hline 99999530 \end{array}$	$\begin{array}{r} 10000000 \\ - 0000470 \\ \hline 99999530 \end{array}$

Inverting the numbers is done by subtracting the largest single digit value to each digit in the number and taking the absolute value of it. Thus, for binary 0, we have $|0 - 1| = 1$, and for binary 1, we have $|1 - 1| = 0$. The same can be done for decimal. For decimal 4, we have $|4 - 9| = 5$, and for decimal 7, we have $|7 - 9| = 3$.

The simplest method of obtaining the two's complement is by inverting the negative value and then adding one to it. To go back the other way, you reverse the process by subtracting one and then inverting the value. Another way to find the two's complement value is through understanding that when you subtract a value from 0, yet you have only a limited number of digits (8 in the above example), so the value will loop backwards as though you subtracted from a value with a 1 in the unobtainable position (9 in the above example).

Whenever the computer interprets the value, you can think of it as looking at the value either subtracted by the value with the 1 in the unobtainable position, or subtracting one and inverting the digits. Either way, we can get from a truly negative value to a positive two's complement value quickly. One way to visualize the equivalence of adding two's complement value to each other to get the correct positive or negative equivalent two's complement value is by using the value with the 1 in the unobtainable position.

Binary (positive result)		Binary (negative result)	
$\begin{array}{r} 00010101 \\ - 00000110 \\ \hline 00011111 \end{array}$	$\begin{array}{r} 00000110 \\ - 00010101 \\ \hline -00011111 \end{array}$	$\begin{array}{r} 00010101 \\ - 00000110 \\ \hline 10000000 \\ - 10000000 \\ \hline 00001111 \end{array}$	$\begin{array}{r} 00000110 \\ - 00010101 \\ \hline 10000000 \\ - 10000000 \\ \hline -00011111 \end{array}$
$\begin{array}{r} 00010101 \\ 11111010 \\ - 10000000 \\ \hline 00011111 \end{array}$	$\begin{array}{r} 00010101 \\ 11111010 \\ + 00001111 \\ \hline 00011111 \end{array}$	$\begin{array}{r} 00010101 \\ 11111010 \\ + 11111010 \\ \hline 00011111 \end{array}$	$\begin{array}{r} 00000110 \\ 11101011 \\ - 10000000 \\ \hline -00011111 \end{array}$
Two's complement		Two's complement	

Shifting Digits

Shifting digits in numbers is fairly straightforward. All you do is increase or decrease the position of the digits in the number. For example, if we had a binary point in our MIPS numbers, then shifting would work as follows:

Theoretical Shifting		Actual MIPS Logical Shifting	
00110110.000	Unshifted	00110110	Unshifted
00110110000.	Shift left 3	10110000	Shift left 3
.00110.110000	Shift right 3	00000110	Shift right 3

Because MIPS binary integers have a limited number of digits, when using logical shifting, the digits disappear when they go beyond the largest or smallest digit positions (as shown above). When shifting to the right, there are actually two types of shifts, *logical* and *arithmetic*. *Logical right shift* replaces the most significant bits with 0s. *Arithmetic right shift* must preserve the sign of the number, so if the number is negative, it replaces the most significant bits with 1s (called sign extension), and if the number is positive, it replaces the most significant bits with 0s.

Multiplying Binary Numbers

Multiplying positive binary numbers together is extremely simple using the techniques used with decimal numbers. For '1' in the *multiplier*, you copy the *multiplicand* shifted over the correct number of digits. All these numbers you add together to get the final *product*.

$\begin{array}{r} 1000 \text{ Multiplicand} \\ \times 1001 \text{ Multiplier} \\ \hline 1000 \\ 0000 \\ 0000 \\ +1000 \\ \hline 1001000 \text{ Product} \end{array}$	$\begin{array}{r} 1111 \\ \times 1111 \\ \hline 1111 \\ 1111 \\ 1111 \\ +1111 \\ \hline 11100001 \end{array}$
Normal Example	Largest Number Example

Notice how the largest possible value obtained by multiplying the two largest 4 bit numbers is no more than 8 bits. This is true no matter how many bits are used. So any product of two 32 bit values must need at most 64 bits to hold its value. This fact is used in the MIPS multiply instruction which I'll describe later.

Another fact about multiplying numbers is how it doesn't seem to work as well with negative 2's complement values as does addition. If the values we multiplied together were the 2's complement value of -1, then the value would be 1, not -31! You should note however that if we had sign extended all of the values we multiplied together then we would have arrived at the correct answer as shown (also notice that we only add up the significant bits):

Sign Extension Bits		
$\begin{array}{r} 11111000 \text{ Multiplicand} \\ \times 11111001 \text{ Multiplier} \\ \hline 11111000 \\ 00000000 \\ 00000000 \\ 110000 \Rightarrow \times -8 \\ 1000 \Rightarrow \times -7 \\ 000 \\ 00 \\ +0 \\ \hline 100111000 \text{ Product} \end{array}$	$\begin{array}{r} 11111111 \\ \times 11111111 \\ \hline 11111111 \\ 11111111 \\ 111111 \\ 11111 \Rightarrow \times -1 \\ 1111 \Rightarrow \times -1 \\ 111 \\ 11 \\ +1 \\ \hline 100000001 \end{array}$	$\begin{array}{r} 11111000 \\ \times 0000101 \\ \hline 11111000 \\ 00000000 \\ 111000 \\ 000000 \Rightarrow \times -8 \\ 0000 \Rightarrow \times -5 \\ 0000 \\ 00 \\ +0 \\ \hline 11011000 \end{array}$
	Out of Bounds Bits	

Because the same two numbers can produce different results depending on whether they are signed or unsigned multiplication, it is important to specify the desired type. For this reason, MIPS has both signed and unsigned multiplication. You should also notice the number of additions required to produce the correct answer increased by two times when we decided to handle 2's complement signed values. A more efficient method of multiplying called *Booth's Algorithm* can reduce the number of additions needed in the signed case to the same number needed in the unsigned case¹.

Dividing Binary Numbers

Dividing binary numbers can also be done using the same methods used in decimal division. (Amaze your friends by division using different bases!) It's also significantly easier than the decimal equivalent because you only have to worry about two values, 0 and 1!

$\begin{array}{r} 1001010 \\ \hline 1000 \end{array} = \text{Divisor } 1000 \begin{array}{r} 1001 \\ \hline 1001010 \\ -1000 \\ \hline 1010 \\ -1000 \\ \hline 10 \end{array} \begin{array}{l} \text{Quotient} \\ \text{Dividend} \\ \text{Remainder} \end{array}$
--

To handle signed numbers, the standard method used is to only divide using positive numbers, compare the signs and figure out what sign to make the result. If both signs are equal, the result is positive. If both signs are opposite, the result is negative. Both the Quotient and Remainder always have the same sign. Once again it is to MIPS advantage to make both a signed and unsigned division.

Instructions Causing Overflow

Certain MIPS instructions, such as the signed add and subtract (both immediate and register types) can cause a system interrupt to be triggered if a condition called *overflow* occurs. The definition of an overflow is whenever a mathematical function creates an answer that cannot be represented given the limitations of the destination register. For example, if two positive numbers were added together and a negative number resulted because the most significant bit became '1', that would be an overflow. Similarly, if two negative numbers were added together and a positive number resulted because the most significant bit became '0', that would be an overflow.

$\begin{array}{r} 10000000 \\ + 11111111 \\ \hline 01111111 \end{array} = + -1$	$\begin{array}{r} 01111111 \\ + 00000001 \\ \hline 10000000 \end{array} = + -1$
Negative to Positive Flip	Positive to Negative Flip

Take notice that numbers of opposite signs added together can never cause an overflow.

¹ Although the algorithm is fun to learn about, I don't have the time to go over it in any detail. It's covered in *Computer Organization & Design: The Hardware/Software Interface* by Patterson and Hennessy on pages 259 through 263.

MIPS Math Instructions

The following are the MIPS instructions for addition, subtraction, multiplication, and division. Notice that the multiplication result is a 64 bit number loaded into two special registers called *Hi* and *Lo*. The division result is two 32 bit numbers loaded into the same two special registers *Hi* and *Lo*. These values can be loaded into general purpose registers using the instructions “mfhi” (move from *Hi*) and “mflo” (move from *Lo*).

```
add $dest, $src1, $src2      # $dest = $src1 + $src2; /* overflow */
sub $dest, $src1, $src2      # $dest = $src1 - $src2; /* overflow */
addi $dest, $src, number     # $dest = $src + number; /* sign extended
                             overflow */
addu $dest, $src1, $src2      # $dest = $src1 + $src2; /* no overflow */
subu $dest, $src1, $src2      # $dest = $src1 - $src2; /* no overflow */
addiu $dest, $src, number     # $dest = $src + number; /* sign extended
                             no overflow */

mult $src1, $src2            # (Hi:Lo) = $src1 * $src2; /* signed
                             no overflow */
multu $src1, $src2           # (Hi:Lo) = $src1 * $src2; /* unsigned
                             no overflow */

div $src1, $src2             # Lo = $src1 / $src2 /* signed */
                             # Hi = $src1 % $src2 /* no overflow */
divu $src1, $src2            # Lo = $src1 / $src2 /* unsigned */
                             # Hi = $src1 % $src2 /* no overflow */

mfhi $dest                   # $dest = Hi
mflo $dest                    # $dest = Lo
```

An important feature of “addiu” that you should be aware of is that the 16 bit immediate value is sign extended. This means the instruction can act as an unsigned addition and subtraction. The only real feature that makes this appear unsigned is that the overflow is ignored as though the source register’s value is unsigned.