3.1 Introduction

Computer words are composed of bits; thus words can be represented as binary numbers. Although the natural numbers 0, 1, 2, and so on can be represented either in decimal or binary form, what about the other numbers that commonly occur? For example:

- How are negative numbers represented?
- What is the largest number that can be represented in a computer word?
- What happens if an operation creates a number bigger than can be represented?
- What about fractions and real numbers?

And underlying all these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel this mystery, including representation of numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may even explain quirks that you have already encountered with computers. (If you are familiar with signed binary numbers, you may wish to skip the next section and go to Section 3.3 on page 170.)

3.2 Signed and Unsigned Numbers

Numbers can be represented in any base; humans prefer base 10 and, as we examined in Chapter 2, base 2 is best for computers. To avoid confusion we subscript decimal numbers with *ten* and binary numbers with *two.*

In any number base, the value of *i*th digit *d* is

$$d \times \text{Base}^i$$

where *i* starts at 0 and increases from right to left. This leads to an obvious way to number the bits in the word: Simply use the power of the base for that bit. For example,

1011_{two}

represents

$$\begin{aligned}
&(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{ten}}\\
&= (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{ten}}\\
&= 8 + 0 + 2 + 1_{\text{ten}}\\
&= 11_{\text{ten}}
\end{aligned}$$

Hence the bits are numbered 0, 1, 2, 3, . . . from *right to left* in a word. The drawing below shows the numbering of bits within a MIPS word and the placement of the number 1011_{two}:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Since words are drawn vertically as well as horizontally, leftmost and rightmost may be unclear. Hence, the phrase **least significant bit** is used to refer to the rightmost bit (bit 0 above) and **most significant bit** to the leftmost bit (bit 31).

least significant bit The rightmost bit in a MIPS word.

most significant bit The leftmost bit in a MIPS word.

The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns. It is natural to let these combinations represent the numbers from 0 to $2^{32} - 1$ ($4{,}294{,}967{,}295_{\text{ten}}$):

$$\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{\text{two}} = 0_{\text{ten}}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{\text{two}} = 1_{\text{ten}}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}\\
&\ldots \qquad\qquad\qquad \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{\text{two}} = 4{,}294{,}967{,}293_{\text{ten}}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = 4{,}294{,}967{,}294_{\text{ten}}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{\text{two}} = 4{,}294{,}967{,}295_{\text{ten}}
\end{aligned}$$

That is, 32-bit binary numbers can be represented in terms of the bit value times a power of 2 (here *xi* means the *i*th bit of *x*):

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^1) + (x0 \times 2^0)$$

Hardware Software Interface

Base 2 is not natural to human beings; we have 10 fingers and so find base 10 natural. Why didn't computers use decimal? In fact, the first commercial computer *did* offer decimal arithmetic. The problem was that the computer still used on and off signals, so a decimal digit was simply represented by several binary digits. Decimal proved so inefficient that subsequent computers reverted to all binary, converting to base 10 only for the relatively infrequent input/output events.

ASCII versus Binary Numbers

EXAMPLE

We could represent numbers as strings of ASCII digits instead of as integers (see Figure 2.21 on page 91). How much does storage increase if the number 1 billion is represented in ASCII versus a 32-bit integer?

ANSWER

One billion is 1 000 000 000, so it would take 10 ASCII digits, each 8 bits long. Thus the storage expansion would be $(10 \times 8)/32$ or 2.5. In addition to the expansion in storage, the hardware to add, subtract, multiply, and divide such numbers is difficult. Such difficulties explain why computing professionals are raised to believe that binary is natural and that the occasional decimal computer is bizarre.

Keep in mind that the binary bit patterns above are simply *representatives* of numbers. Numbers really have an infinite number of digits, with almost all being 0 except for a few of the rightmost digits. We just don't normally show leading 0s.

Hardware can be designed to add, subtract, multiply, and divide these binary bit patterns. If the number that is the proper result of such operations cannot be represented by these rightmost hardware bits, *overflow* is said to have occurred. It's up to the operating system and program to determine what to do if overflow occurs.

Computer programs calculate both positive and negative numbers, so we need a representation that distinguishes the positive from the negative. The most obvious solution is to add a separate sign, which conveniently can be represented in a single bit; the name for this representation is *sign and magnitude*.

Alas, sign and magnitude representation has several shortcomings. First, it's not obvious where to put the sign bit. To the right? To the left? Early computers tried both. Second, adders for sign and magnitude may need an extra step to set the sign because we can't know in advance what the proper sign will be. Finally, a separate sign bit means that sign and magnitude has both a positive and negative zero, which can lead to problems for inattentive programmers. As a result of these shortcomings, sign and magnitude was soon abandoned.

In the search for a more attractive alternative, the question arose as to what would be the result for unsigned numbers if we tried to subtract a large number from a small one. The answer is that it would try to borrow from a string of leading 0s, so the result would have a string of leading 1s.

Given that there was no obvious better alternative, the final solution was to pick the representation that made the hardware simple: leading 0s mean positive, and leading 1s mean negative. This convention for representing signed binary numbers is called *two's complement* representation:

$$
\begin{array}{lcl}
0000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_{\text{two}} & = & 0_{\text{ten}} \\
0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{\text{two}} & = & 1_{\text{ten}} \\
0000\,0000\,0000\,0000\,0000\,0000\,0000\,0010_{\text{two}} & = & 2_{\text{ten}} \\
\ldots & & \ldots \\
0111\,1111\,1111\,1111\,1111\,1111\,1111\,1101_{\text{two}} & = & 2{,}147{,}483{,}645_{\text{ten}} \\
0111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{\text{two}} & = & 2{,}147{,}483{,}646_{\text{ten}} \\
0111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_{\text{two}} & = & 2{,}147{,}483{,}647_{\text{ten}} \\
1000\,0000\,0000\,0000\,0000\,0000\,0000\,0000_{\text{two}} & = & -2{,}147{,}483{,}648_{\text{ten}} \\
1000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{\text{two}} & = & -2{,}147{,}483{,}647_{\text{ten}} \\
1000\,0000\,0000\,0000\,0000\,0000\,0000\,0010_{\text{two}} & = & -2{,}147{,}483{,}646_{\text{ten}} \\
\ldots & & \ldots \\
1111\,1111\,1111\,1111\,1111\,1111\,1111\,1101_{\text{two}} & = & -3_{\text{ten}} \\
1111\,1111\,1111\,1111\,1111\,1111\,1111\,1110_{\text{two}} & = & -2_{\text{ten}} \\
1111\,1111\,1111\,1111\,1111\,1111\,1111\,1111_{\text{two}} & = & -1_{\text{ten}}
\end{array}
$$

The positive half of the numbers, from 0 to $2{,}147{,}483{,}647_{\text{ten}}$ ($2^{31} - 1$), use the same representation as before. The following bit pattern ($1000 \ldots 0000_{\text{two}}$) represents the most negative number $-2{,}147{,}483{,}648_{\text{ten}}$ (-2^{31}). It is followed by a declining set of negative numbers: $-2{,}147{,}483{,}647_{\text{ten}}$ ($1000 \ldots 0001_{\text{two}}$) down to -1_{ten} ($1111 \ldots 1111_{\text{two}}$).

Two's complement does have one negative number, $-2{,}147{,}483{,}648_{\text{ten}}$, that has no corresponding positive number. Such imbalance was a worry to the inattentive programmer, but sign and magnitude had problems for both the programmer *and* the hardware designer. Consequently, every computer today uses two's complement binary representations for signed numbers.

Two's complement representation has the advantage that all negative numbers have a 1 in the most significant bit. Consequently, hardware needs to test only this bit to see if a number is positive or negative (with 0 considered positive). This bit is often called the *sign bit*. By recognizing the role of the sign bit, we can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

The sign bit is multiplied by -2^{31}, and the rest of the bits are then multiplied by positive versions of their respective base values.

EXAMPLE

Binary to Decimal Conversion

What is the decimal value of this 32-bit two's complement number?

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{two}$

ANSWER

Substituting the number's bit values into the formula above:

$$\begin{aligned}&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{2}) + (0 \times 2^{1}) + (0 \times 2^{0})\\ &= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0\\ &= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}\\ &= -4_{ten}\end{aligned}$$

We'll see a shortcut to simplify conversion soon.

Just as an operation on unsigned numbers can overflow the capacity of hardware to represent the result, so can an operation on two's complement numbers. Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left (the sign bit is incorrect): a 0 on the left of the bit pattern when the number is negative or a 1 when the number is positive.

Hardware Software Interface

Signed versus unsigned applies to loads as well as to arithmetic. The *function* of a signed load is to copy the sign repeatedly to fill the rest of the register—called *sign extension*—but its *purpose* is to place a correct representation of the number within that register. Unsigned loads simply fill with 0s to the left of the data, since the number represented by the bit pattern is unsigned.

When loading a 32-bit word into a 32-bit register, the point is moot; signed and unsigned loads are identical. MIPS does offer two flavors of byte loads: *load byte* (`lb`) treats the byte as a signed number and thus sign-extends to fill the 24 leftmost bits of the register, while *load byte unsigned* (`lbu`) works with unsigned integers. Since C programs almost always use bytes to represent characters rather than consider bytes as very short signed integers, `lbu` is used practically exclusively for byte loads. For similar reasons, *load half* (`lh`) treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register, while *load halfword unsigned* (`lhu`) works with unsigned integers.

Hardware Software Interface

Unlike the numbers discussed above, memory addresses naturally start at 0 and continue to the largest address. Put another way, negative addresses make no sense. Thus, programs want to deal sometimes with numbers that can be positive or negative and sometimes with numbers that can be only positive. Some programming languages reflect this distinction. C, for example, names the former *integers* (declared as `int` in the program) and the latter *unsigned integers* (`unsigned int`). Some C style guides even recommend declaring the former as `signed int` to keep the distinction clear.

Comparison instructions must deal with this dichotomy. Sometimes a bit pattern with a 1 in the most significant bit represents a negative number and, of course, is less than any positive number, which must have a 0 in the most significant bit. With unsigned integers, on the other hand, a 1 in the most significant bit represents a number that is *larger* than any that begins with a 0. (We'll take advantage of this dual meaning of the most significant bit to reduce the cost of the array bounds checking in a few pages.)

MIPS offers two versions of the set on less than comparison to handle these alternatives. *Set on less than* (`slt`) and *set on less than immediate* (`slti`) work with signed integers. Unsigned integers are compared using *set on less than unsigned* (`sltu`) and *set on less than immediate unsigned* (`sltiu`).

Signed versus Unsigned Comparison

EXAMPLE

Suppose register `$s0` has the binary number

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two}$

and that register `$s1` has the binary number

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two}$

What are the values of registers `$t0` and `$t1` after these two instructions?

```
slt     $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

ANSWER

The value in register `$s0` represents –1 if it is an integer and $4{,}294{,}967{,}295_{ten}$ if it is an unsigned integer. The value in register `$s1` represents 1 in either case. Then register `$t0` has the value 1, since $-1_{ten} < 1_{ten}$, and register `$t1` has the value 0, since $4{,}294{,}967{,}295_{ten} > 1_{ten}$.

Before going on to addition and subtraction, let's examine a few useful shortcuts when working with two's complement numbers.

The first shortcut is a quick way to negate a two's complement binary number. Simply invert every 0 to 1 and every 1 to 0, then add one to the result. This shortcut is based on the observation that the sum of a number and its inverted representation must be $111 \ldots 111_{two}$, which represents –1. Since $x + \bar{x} \equiv -1$, therefore $x + \bar{x} + 1 = 0$ or $\bar{x} + 1 = -x$.

Negation Shortcut

EXAMPLE

Negate 2_{ten}, and then check the result by negating -2_{ten}.

ANSWER

$2_{ten} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

Negating this number by inverting the bits and adding one,

$$\begin{array}{rr} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$$

is first inverted and then incremented:

$$\begin{array}{rr} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

The second shortcut tells us how to convert a binary number represented in *n* bits to a number represented with more than *n* bits. For example, the immediate field in the load, store, branch, add, and set on less than instructions contains a two's complement 16-bit number, representing $-32{,}768_{\text{ten}}$ (-2^{15}) to $32{,}767_{\text{ten}}$ ($2^{15} - 1$). To add the immediate field to a 32-bit register, the computer must convert that 16-bit number to its 32-bit equivalent. The shortcut is to take the most significant bit from the smaller quantity—the sign bit—and replicate it to fill the new bits of the larger quantity. The old bits are simply copied into the right portion of the new word. This shortcut is commonly called *sign extension.*

Sign Extension Shortcut

EXAMPLE

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

ANSWER

The 16-bit binary version of the number 2 is

$0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{\text{two}} = 2_{\text{ten}}$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$0000\ 0000\ 0000\ 0010_{\text{two}}$

becomes

$$\begin{array}{lr} & 1111\ 1111\ 1111\ 1101_{\text{two}} \\ + & 1_{\text{two}} \\ \hline \end{array}$$

$= 1111\ 1111\ 1111\ 1110_{\text{two}}$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{\text{two}} = -2_{\text{ten}}$

This trick works because positive two's complement numbers really have an infinite number of 0s on the left and those that are negative two's complement numbers have an infinite number of 1s. The binary bit pattern representing a number hides leading bits to fit the width of the hardware; sign extension simply restores some of them.

The third shortcut reduces the cost of checking if $0 \le x < y$, which matches the index out-of-bounds check for arrays. The key is that negative integers in two's complement notation look like large numbers in unsigned notation; that is, the most significant bit is a sign bit in the former notation but a large part of the number in the latter. Thus, an unsigned comparison of $x < y$ also checks if x is negative.

EXAMPLE

Bounds Check Shortcut

Use this shortcut to reduce an index-out-of-bounds check: jump to `IndexOutOfBounds` if `$a1` ≥ `$t2` or if `$a1` is negative.

ANSWER

The checking code just uses `sltu` to do both checks:

```
sltu $t0,$a1,$t2 # Temp reg $t0=0 if k>=length or k<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

Summary

The main point of this section is that we need to represent both positive and negative integers within a computer word, and although there are pros and cons to any option, the overwhelming choice since 1965 has been two's complement. Figure 3.1 shows the additions to the MIPS assembly language revealed in this section. (The MIPS machine language is also illustrated on the back endpapers of this book.)

Check Yourself

Which type of variable that can contain $1{,}000{,}000{,}000_{\text{ten}}$ takes the most memory space?

1. `int` in C
2. `string` in C
3. `string` in Java (which uses Unicode)

MIPS operands

Name	Example	Comments
32 registers	`$s0–$s7, $t0–$t9, $gp, $fp, $zero, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register `$zero` always equals 0. Register `$at` is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`	Three operands
	subtract	`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`	Three operands
	add immediate	`addi $s1,$s2,100`	`$s1 = $s2 + 100`	+ constant
Data transfer	load word	`lw $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Word from memory to register
	store word	`sw $s1,100($s2)`	`Memory[$s2 + 100] = $s1`	Word from register to memory
	load half unsigned	`lhu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Halfword memory to register
	store half	`sh $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Halfword register to memory
	load byte unsigned	`lbu $s1,100($s2)`	`$s1 = Memory[$s2 + 100]`	Byte from memory to register
	store byte	`sb $s1,100($s2)`	Memory[`$s2` + 100] = `$s1`	Byte from register to memory
	load upper immediate	`lui $s1,100`	`$s1 = 100 *` 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1 = $s2 & $s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1 = $s2 | $s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1 = ~ ($s2 |$s3)`	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,100`	`$s1 = $s2 & 100`	Bit-by-bit AND with constant
	or immediate	`ori $s1,$s2,100`	`$s1 = $s2 | 100`	Bit-by-bit OR with constant
	shift left logical	`sll $s1,$s2,10`	`$s1 = $s2 << 10`	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1 = $s2 >> 10`	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1` != `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; two's complement
	set less than immediate	`slti $s1,$s2,100`	if (`$s2` < 100) `$s1 = 1`; else `$s1 = 0`	Compare < constant; two's complement
	set less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2 < $s3) $s1 = 1`; else `$s1 = 0`	Compare less than; unsigned numbers
	set less than immediate unsigned	`sltiu $s1,$s2,100`	if `($s2 < 100) $s1 = 1`; else `$s1 = 0`	Compare < constant; unsigned numbers
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

FIGURE 3.1 MIPS architecture revealed thus far. Color indicates portions from this section added to the MIPS architecture revealed in Chapter 2 (Figure 3.26 on page 228). MIPS machine language is listed in the MIPS summary reference card in the front of this book.

Elaboration: Two's complement gets its name from the rule that the unsigned sum of an n-bit number and its negative is 2^n; hence, the complement or negation of a two's complement number x is $2^n - x$.

A third alternative representation is called *one's complement*. The negative of a one's complement is found by inverting each bit, from 0 to 1 and from 1 to 0, which helps explain its name since the complement of x is $2^n - x - 1$. It was also an attempt to be a better solution than sign and magnitude, and several scientific computers did use the notation. This representation is similar to two's complement except that it also has two 0s: $00 \dots 00_{two}$ is positive 0 and $11 \dots 11_{two}$ is negative 0. The most negative number $10 \dots 000_{two}$ represents $-2{,}147{,}483{,}647_{ten}$, and so the positives and negatives are balanced. One's complement adders did need an extra step to subtract a number, and hence two's complement dominates today.

A final notation, which we will look at when we discuss floating point, is to represent the most negative value by $00 \dots 000_{two}$ and the most positive value represented by $11 \dots 11_{two}$, with 0 typically having the value $10 \dots 00_{two}$. This is called a biased notation, since it biases the number such that the number plus the bias has a nonnegative representation.

biased notation A notation that represents the most negative value by $00 \dots 000_{two}$ and the most positive value by $11 \dots 11_{two}$, with 0 typically having the value $10 \dots 00_{two}$, thereby biasing the number such that the number plus the bias has a nonnegative representation.

Elaboration: For signed decimal numbers we used "–" to represent negative because there are no limits to the size of a decimal number. Given a fixed word size, binary and hexadecimal bit strings can encode the sign, and hence we do not normally use "+" or "–" with binary or hexadecimal notation.

Subtraction: Addition's Tricky Pal

No. 10, Top Ten Courses for Athletes at a Football Factory, David Letterman et al., *Book of Top Ten Lists*, 1990

3.3 Addition and Subtraction

Addition is just what you would expect in computers. Digits are added bit by bit from right to left, with carries passed to the next digit to the left, just as you would do by hand. Subtraction uses addition: The appropriate operand is simply negated before being added.

EXAMPLE

Binary Addition and Subtraction

Let's try adding 6_{ten} to 7_{ten} in binary and then subtracting 6_{ten} from 7_{ten} in binary.

ANSWER

$$
\begin{array}{lll}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\
+ & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} & = 6_{ten} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,1101_{two} & = 13_{ten}
\end{array}
$$

The 4 bits to the right have all the action; Figure 3.2 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

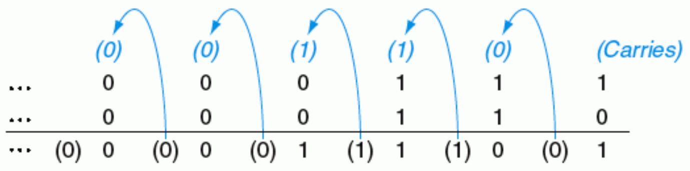

FIGURE 3.2 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 1. The third digit is the sum of 1 + 1 + 1, resulting in a carry out of 1 and a sum bit of 1. The fourth bit is 1 + 0 + 0, yielding a 1 sum and no carry.

Subtracting 6_{ten} from 7_{ten} can be done directly:

$$
\begin{array}{lll}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\
- & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0110_{two} & = 6_{ten} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} & = 1_{ten}
\end{array}
$$

or via addition using the two's complement representation of –6:

$$
\begin{array}{lll}
 & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0111_{two} & = 7_{ten} \\
+ & 1111\,1111\,1111\,1111\,1111\,1111\,1111\,1010_{two} & = -6_{ten} \\
\hline
= & 0000\,0000\,0000\,0000\,0000\,0000\,0000\,0001_{two} & = 1_{ten}
\end{array}
$$

We said earlier that overflow occurs when the result from an operation cannot be represented with the available hardware, in this case a 32-bit word. When can overflow occur in addition? When adding operands with different signs, overflow cannot occur. The reason is the sum must be no larger than one of the operands. For example, –10 + 4 = –6. Since the operands fit in 32 bits and the sum is no larger than an operand, the sum must fit in 32 bits as well. Therefore no overflow can occur when adding positive and negative operands.

There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: When the signs of the operands are the *same*, overflow cannot occur. To see this, remember that $x - y = x + (-y)$ because we subtract by negating the second operand and then add. So, when we subtract operands of the same sign we end up by *adding* operands of *different* signs. From the prior paragraph, we know that overflow cannot occur in this case either.

Having examined when overflow cannot occur in addition and subtraction, we still haven't answered how to detect when it *does* occur. Overflow occurs when adding two positive numbers and the sum is negative, or vice versa. Clearly, adding or subtracting two 32-bit numbers can yield a result that needs 33 bits to be fully expressed. The lack of a 33rd bit means that when overflow occurs the sign bit is being set with the *value* of the result instead of the proper sign of the result. Since we need just one extra bit, only the sign bit can be wrong. This means a carry out occurred into the sign bit.

Overflow occurs in subtraction when we subtract a negative number from a positive number and get a negative result, or when we subtract a positive number from a negative number and get a positive result. This means a borrow occurred from the sign bit. Figure 3.3 shows the combination of operations, operands, and results that indicate an overflow.

We have just seen how to detect overflow for two's complement numbers in a computer. What about unsigned integers? Unsigned integers are commonly used for memory addresses where overflows are ignored.

The computer designer must therefore provide a way to ignore overflow in some cases and to recognize it in others. The MIPS solution is to have two kinds of arithmetic instructions to recognize the two choices:

- Add (`add`), add immediate (`addi`), and subtract (`sub`) cause exceptions on overflow.
- Add unsigned (`addu`), add immediate unsigned (`addiu`), and subtract unsigned (`subu`) do *not* cause exceptions on overflow.

Because C ignores overflows, the MIPS C compilers will always generate the unsigned versions of the arithmetic instructions `addu`, `addiu`, and `subu` no matter what the type of the variables. The MIPS Fortran compilers, however, pick the appropriate arithmetic instructions, depending on the type of the operands.

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

FIGURE 3.3 Overflow conditions for addition and subtraction.