3.1 Introduction

Computer words are composed of bits; thus, words can be represented as binary numbers. Chapter 2 shows that integers can be represented either in decimal or binary form, but what about the other numbers that commonly occur? For example:

- What about fractions and other real numbers?
- What happens if an operation creates a number bigger than can be represented?
- And underlying these questions is a mystery: How does hardware really multiply or divide numbers?

The goal of this chapter is to unravel these mysteries including representation of real numbers, arithmetic algorithms, hardware that follows these algorithms, and the implications of all this for instruction sets. These insights may explain quirks that you have already encountered with computers. Moreover, we show how to use this knowledge to make arithmetic-intensive programs go much faster.

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

3.2 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.

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

The 4 bits to the right have all the action; Figure 3.1 shows the sums and carries. The carries are shown in parentheses, with the arrows showing how they are passed.

ANSWER

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

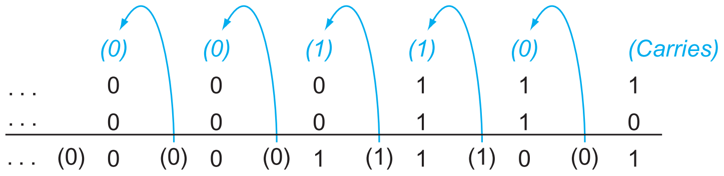

FIGURE 3.1 Binary addition, showing carries from right to left. The rightmost bit adds 1 to 0, resulting in the sum of this bit being 1 and the carry out from this bit being 0. Hence, the operation for the second digit to the right is 0 + 1 + 1. This generates a 0 for this sum bit and a carry out of 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.

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

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

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

Recall 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 $c - a = c + (-a)$ because we subtract by negating the second operand and then add. Therefore, 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.

Knowing when overflow cannot occur in addition and subtraction is all well and good, but how do we detect it when it *does* occur? 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 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. Hence, overflow occurs when adding two positive numbers and the sum is negative, or vice versa. This spurious sum 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. Such a ridiculous result means a borrow occurred from the sign bit. Figure 3.2 shows the combination of operations, operands, and results that indicate an overflow.

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 3.2 Overflow conditions for addition and subtraction.

We have just seen how to detect overflow for two's complement numbers in a computer. What about overflow with 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.

Appendix B describes the hardware that performs addition and subtraction, which is called an **Arithmetic Logic Unit** or **ALU**.

Arithmetic Logic Unit (ALU) Hardware that performs addition, subtraction, and usually logical operations such as AND and OR.

Elaboration: A constant source of confusion for `addiu` is its name and what happens to its immediate field. The u stands for unsigned, which means addition cannot cause an overflow exception. However, the 16-bit immediate field is sign extended to 32 bits, just like `addi`, `slti`, and `sltiu`. Thus, the immediate field is signed, even if the operation is "unsigned."

Hardware/Software Interface

The computer designer must decide how to handle arithmetic overflows. Although some languages like C and Java ignore integer overflow, languages like Ada and Fortran require that the program be notified. The programmer or the programming environment must then decide what to do when overflow occurs.

MIPS detects overflow with an **exception**, also called an **interrupt** on many computers. An exception or interrupt is essentially an unscheduled procedure call. The address of the instruction that overflowed is saved in a register, and the computer jumps to a predefined address to invoke the appropriate routine for that exception. The interrupted address is saved so that in some situations the program can continue after corrective code is executed. (Section 4.9 covers exceptions in

exception Also called **interrupt** on many computers. An unscheduled event that disrupts program execution; used to detect overflow.

more detail; Chapter 5 describes other situations where exceptions and interrupts occur.)

MIPS includes a register called the *exception program counter* (EPC) to contain the address of the instruction that caused the exception. The instruction *move from system control* (`mfc0`) is used to copy EPC into a general-purpose register so that MIPS software has the option of returning to the offending instruction via a jump register instruction.

interrupt An exception that comes from outside of the processor. (Some architectures use the term *interrupt* for all exceptions.)

Summary

A major point of this section is that, independent of the representation, the finite word size of computers means that arithmetic operations can create results that are too large to fit in this fixed word size. It's easy to detect overflow in unsigned numbers, although these are almost always ignored because programs don't want to detect overflow for address arithmetic, the most common use of natural numbers. Two's complement presents a greater challenge, yet some software systems require detection of overflow, so today all computers have a way to detect it.

Check Yourself

Some programming languages allow two's complement integer arithmetic on variables declared byte and half, whereas MIPS only has integer arithmetic operations on full words. As we recall from Chapter 2, MIPS does have data transfer operations for bytes and halfwords. What MIPS instructions should be generated for byte and halfword arithmetic operations?

1. Load with `lbu`, `lhu`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
2. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`; then store using `sb`, `sh`.
3. Load with `lb`, `lh`; arithmetic with `add`, `sub`, `mult`, `div`, using `AND` to mask result to 8 or 16 bits after each operation; then store using `sb`, `sh`.

Elaboration: One feature not generally found in general-purpose microprocessors is *saturating* operations. Saturation means that when a calculation overflows, the result is set to the largest positive number or most negative number, rather than a modulo calculation as in two's complement arithmetic. Saturation is likely what you want for media operations. For example, the volume knob on a radio set would be frustrating if, as you turned it, the volume would get continuously louder for a while and then immediately very soft. A knob with saturation would stop at the highest volume no matter how far you turned it. Multimedia extensions to standard instruction sets often offer saturating arithmetic.

Elaboration: MIPS can trap on overflow, but unlike many other computers, there is no conditional branch to test overflow. A sequence of MIPS instructions can discover overflow. For signed addition, the sequence is the following (see the *Elaboration* on page 89 in Chapter 2 for a description of the `xor` instruction):

```
addu $t0, $t1, $t2 # $t0 = sum, but don't trap
xor  $t3, $t1, $t2 # Check if signs differ
slt  $t3, $t3, $zero # $t3 = 1 if signs differ
bne  $t3, $zero, No_overflow # $t1, $t2 signs ≠,
                             # so no overflow
xor $t3, $t0, $t1 # signs =; sign of sum match too?
                  # $t3 negative if sum sign different
slt $t3, $t3, $zero # $t3 = 1 if sum sign different
bne $t3, $zero, Overflow # All 3 signs ≠; goto overflow
```

For unsigned addition (`$t0 = $t1 + $t2`), the test is

```
addu $t0, $t1, $t2     # $t0 = sum
nor $t3, $t1, $zero    # $t3 = NOT $t1
                       # (2's comp - 1: 2^32 - $t1 - 1)
sltu $t3, $t3, $t2     # (2^32 - $t1 - 1) < $t2
                       # ⇒ 2^32 - 1 < $t1 + $t2
bne $t3,$zero,Overflow # if(2^32-1<$t1+$t2) goto overflow
```

Elaboration: In the preceding text, we said that you copy EPC into a register via `mfc0` and then return to the interrupted code via jump register. This directive leads to an interesting question: since you must first transfer EPC to a register to use with jump register, how can jump register return to the interrupted code *and* restore the original values of *all* registers? Either you restore the old registers first, thereby destroying your return address from EPC, which you placed in a register for use in jump register, or you restore all registers but the one with the return address so that you can jump—meaning an exception would result in changing that one register at any time during program execution! Neither option is satisfactory.

To rescue the hardware from this dilemma, MIPS programmers agreed to reserve registers `$k0` and `$k1` for the operating system; these registers are *not* restored on exceptions. Just as the MIPS compilers avoid using register `$at` so that the assembler can use it as a temporary register (see *Hardware/Software Interface* in Section 2.10), compilers also abstain from using registers `$k0` and `$k1` to make them available for the operating system. Exception routines place the return address in one of these registers and then use jump register to restore the instruction address.

Elaboration: The speed of addition is increased by determining the carry in to the high-order bits sooner. There are a variety of schemes to anticipate the carry so that the worst-case scenario is a function of the $\log_2$ of the number of bits in the adder. These anticipatory signals are faster because they go through fewer gates in sequence, but it takes many more gates to anticipate the proper carry. The most popular is *carry lookahead*, which Section B.6 in **Appendix B** describes.