3.1 Introduction

> “ Numerical precision is the very soul of science.
>
> *Sir D'arcy Wentworth Thompson, On Growth and Form, 1917.*

Computer words are composed of bits; thus, words can be represented as binary numbers. COD Chapter 2 (Instructions: Language of the Computer) 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.

PARTICIPATION ACTIVITY | 3.1.1: Representation of information as bits.

1) Integers can be represented as bits.

○ True

○ False

2) Instructions can be represented as bits.

○ True

○ False

3.2 Addition and subtraction

> 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*

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.

PARTICIPATION ACTIVITY 3.2.1: Example of binary addition (COD Figure 3.1).

```
                                            1 1 0
    0000  0000  0000  0000  0000  0000  0000  0 1 1 1 two   = 7 ten
+   0000  0000  0000  0000  0000  0000  0000  0 1 1 0 two   = 6 ten
    0000  0000  0000  0000  0000  0000  0000  1 1 0 1 two   = 13 ten
```

Animation content:

Static figure: A binary addition of two 32-bit operands and a 32-bit resulting sum. The most significant 28 bits of both operands are zeros. The top operand ends with 0 1 1 1, the bottom operand ends with 0 1 1 0, and the sum ends with 1 1 0 1. The decimal equivalents for the 4-bit binary values are shown, with 0 1 1 1 equivalent to 7 base ten, 0 1 1 0 equivalent to 6 base ten, and 1 1 0 1 equivalent to 13 base ten. The carry bits 1 1 0 are above the fourth, third and second least significant bits of the top operand respectively.

Step 1: 1 plus 0 is 0 1. Carry bit is 0, sum bit is 1.
The least significant bit of the top operand 1 is added to the corresponding bit of the bottom operand 0. The result is 0 1 in binary, with the leftmost bit representing the carry bit 0 and the rightmost bit representing the least significant sum bit 1.

Step 2: 0 plus 1 plus 1 is 1 0. Carry bit is 1, sum bit is 0.
The second least significant bit of the top operand 1 is added to the corresponding bit of the bottom operand 1, plus the carry bit 0 from the previous addition. The result is 1 0 in binary, with the leftmost bit representing the carry bit 1 and the rightmost bit representing the second least significant sum bit 0.

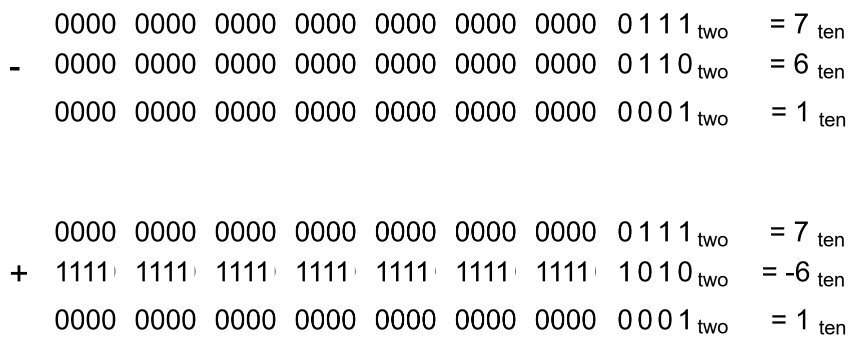
Step 3: 1 plus 1 plus 1 is 1 1. Carry bit is 1, sum bit is 1.
The third least significant bit of the top operand 1 is added to the corresponding bit of the bottom operand 1, plus the carry bit 1 from the previous addition. The result is 1 1 in binary, with the leftmost bit representing the carry bit 1 and the rightmost bit representing the third least significant sum bit 1.

Step 4: 1 plus 0 plus 0 is 0 1. Remaining bits sum to 0's.
The fourth least significant bit of the top operand 0 is added to the corresponding bit of the bottom operand 0, plus the carry bit 1 from the previous addition. The result is 0 1 in binary, with no further carry bit, and the fourth least significant sum bit is 1. The remaining sum bits to the left are all zeros.

Animation captions:

1. 1 + 0 is 01. Carry bit is 0, sum bit is 1.
2. 0 + 1 + 1 is 10. Carry bit is 1, sum bit is 0.
3. 1 + 1 + 1 is 11. Carry bit is 1, sum bit is 1.
4. 1 + 0 + 0 is 01. Remaining bits sum to 0's.

PARTICIPATION ACTIVITY | 3.2.2: Example of binary subtraction.



```
  0000 0000 0000 0000 0000 0000 0000 0111 two  = 7 ten
- 0000 0000 0000 0000 0000 0000 0000 0110 two  = 6 ten
  0000 0000 0000 0000 0000 0000 0000 0001 two  = 1 ten

  0000 0000 0000 0000 0000 0000 0000 0111 two  = 7 ten
+ 1111 1111 1111 1111 1111 1111 1111 1010 two  = -6 ten
  0000 0000 0000 0000 0000 0000 0000 0001 two  = 1 ten
```

Animation content:

Static figure: A binary subtraction operation. The most significant 28 bits of both operands and the resulting difference are zeros and the least significant 4 bits are shown in the operation. The top operand ends with 0 1 1 1, the bottom operand ends with 0 1 1 0 and the difference ends with

0 0 0 1. The decimal equivalents for the 4-bit binary values are shown, with 0 1 1 1 equivalent to 7 base ten, 0 1 1 0 equivalent to 6 base ten, and 0 0 0 1 equivalent to 1 base ten. A separate binary addition of two 32-bit operands and a 32-bit resulting sum. The most significant 28 bits for the first operand and the sum are zeros, the most significant 28 bits for the second operand are ones, and the least significant 4 bits of each are shown in the operation. The top operand ends with 0 1 1 1, the bottom operand ends with 1 0 1 0, and the sum ends with 0 0 0 1. The decimal equivalents for the 4-bit binary values are shown, with 0 1 1 1 equivalent to 7 base ten, 1 0 1 0 equivalent to negative 6 base ten, and 0 0 0 1 equivalent to 1 base ten.

Step 1: Subtraction can be done as normally done by hand.
A binary subtraction operation where 6 is subtracted from 7. Each bit is subtracted from right to left, with the final binary result being 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, representing 1 base ten.

Step 2: Or, subtraction can be done by determining the two's complement representation of negative 6 (6's bits inverted plus 1) ...
The bits of 6 in binary are inverted and 1 is added, resulting in the binary number 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0, which represents negative 6 in two's complement representation.

Step 3: ... and then adding that two's complement representation of -6 with 7.
The resulting binary number 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 0 0 1, or 1 base ten appears.

Animation captions:

1. Subtraction can be done as normally done by hand.
2. Or, subtraction can be done by determining the two's complement representation of -6 (6's bits inverted plus 1) ...
3. ... and then adding that two's complement representation of -6 with 7.

PARTICIPATION ACTIVITY | 3.2.3: Binary addition.

Consider the addition of base ten numbers 3 and 2 using 32-bit binary numbers. Fill in the missing values.

```
          b a
  0 ... 0 0 1 1
+ 0 ... 0 0 1 0
----------------
  0 ... 0 y x w
```

1) w

○ 0

○ 1

2) a

○ 0

○ 1

3) x

○ 0

○ 1

4) b

○ 0

○ 1

5) y

○ 0

○ 1

6) 0...0011 + 0...0010 = ?

○ 0...0100

○ 0...0101

PARTICIPATION ACTIVITY 3.2.4: Binary subtraction.

Consider the subtraction of base ten numbers 5 - 4 using 32-bit binary numbers, and achieved by adding 5 with the two's complement of 4:

```
  0 0 ... 0 1 0 1       0 0 ... 0 1 0 1
- 0 0 ... 0 1 0 0     + 1 1 ... d c b a
-----------------     -----------------
                      s 0 0 ... z y x w
```

1) dcba

○ 1011

○ 1100

2) zyxw

○ 0000

○ 0001

3) If a 33rd sum bit, s, existed on the left, what value would that bit get?

○ 0

○ 1

CHALLENGE ACTIVITY 3.2.1: Add two numbers.

622166.4950548.qx3zqy7

Start

Add 2 and 3. Work from right to left.

			⋯	-	-	-	-			Carry bits
	0	0	⋯	0	0	0	1	0	2	32-bit number
+	0	0	⋯	0	0	0	1	1	3	32-bit number
	0	0	⋯	-	-	-	-	-		32-bit number

1	2	3	4	5

Check Next

Overflow

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.

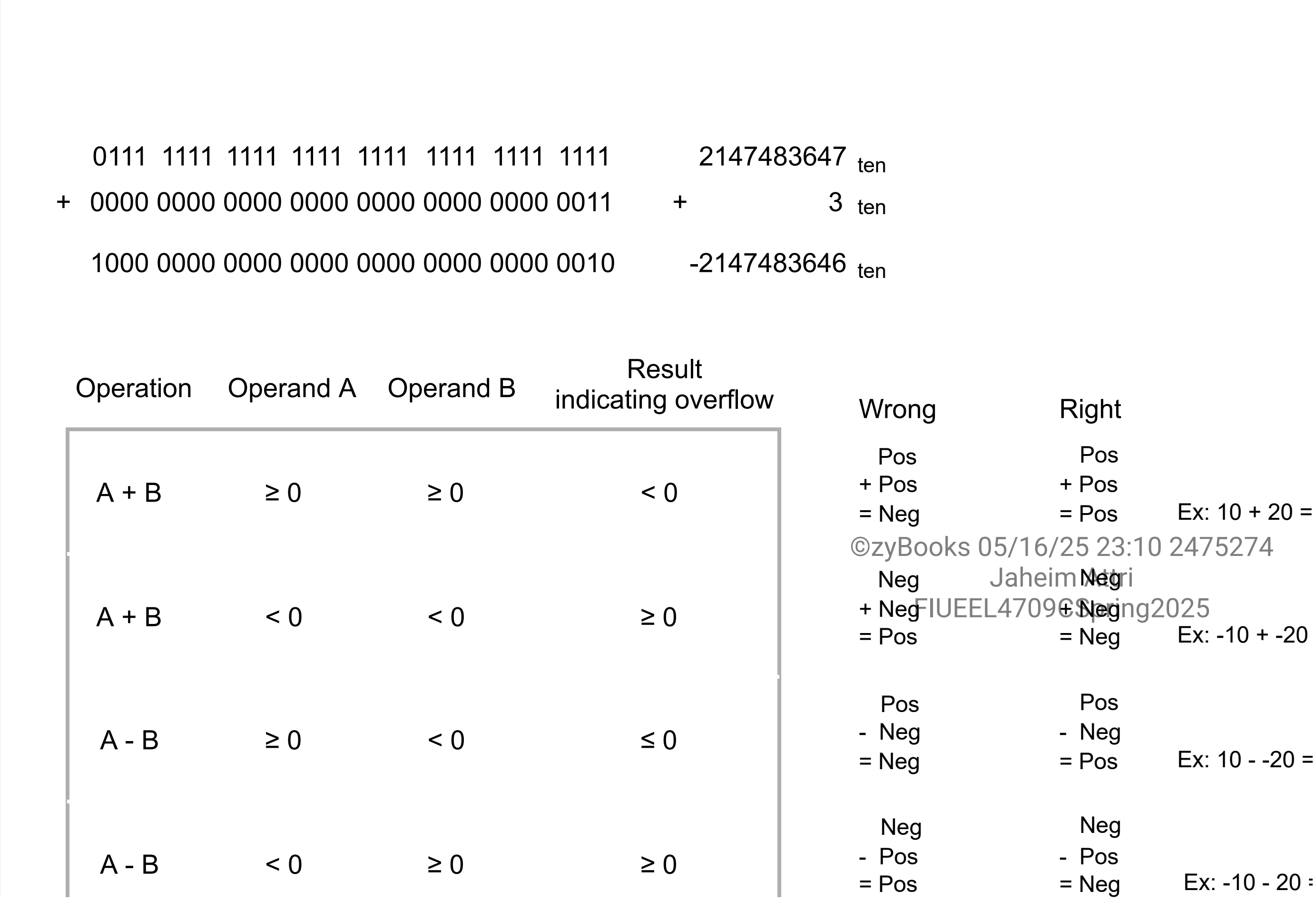
There are similar restrictions to the occurrence of overflow during subtract, but it's just the opposite principle: when the signs of the operands are the *same*, overflow cannot occur. To see this, remember that *c - a = c + (−a)* because we subtract by negating the second operand and then add. Therefore, 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. The figure below shows the combination of operations, operands, and results that indicate an overflow.

PARTICIPATION ACTIVITY 3.2.5: Overflow conditions for addition and subtraction (COD Figure 3.2).



```
  0111  1111  1111  1111  1111  1111  1111  1111        2147483647 ten
+ 0000 0000 0000 0000 0000 0000 0000 0011          +             3 ten

  1000 0000 0000 0000 0000 0000 0000 0010          -2147483646 ten
```

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	
A + B	≥ 0	≥ 0	< 0	Pos + Pos = Neg	Pos + Pos = Pos	Ex: 10 + 20 =
A + B	< 0	< 0	≥ 0	Neg + Neg = Pos	Neg + Neg = Neg	Ex: -10 + -20
A - B	≥ 0	< 0	≤ 0	Pos - Neg = Neg	Pos - Neg = Pos	Ex: 10 - -20 =
A - B	< 0	≥ 0	≥ 0	Neg - Pos = Pos	Neg - Pos = Neg	Ex: -10 - 20 =

Animation content:

Static Figure:
The following binary calculation is displayed:
  0111 1111 1111 1111 1111 1111 1111 1111
+ 0000 0000 0000 0000 0000 0000 0000 0011
= 1000 0000 0000 0000 0000 0000 0000 0010

The equivalent decimal calculation is displayed:
2147483647 + 3 = -2147483646

The following chart is shown:

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	Example
A + B	≥ 0	≥ 0	< 0	Pos + Pos = Neg	Pos + Pos = Pos	10 + 20 = 30
A + B	< 0	< 0	≥ 0	Neg + Neg = Pos	Neg + Neg = Neg	-10 + -20 = -30
A - B	≥ 0	< 0	≤ 0	Pos - Neg = Neg	Pos - Neg = Pos	10 - -20 = 30
A - B	< 0	≥ 0	≥ 0	Neg - Pos = Pos	Neg - Pos = Neg	-10 - 20 = -30

Step 1: Overflow occurs when result exceeds 31 bits, using 32nd bit for the result rather than the sign.

The following binary calculation is displayed:
  0111 1111 1111 1111 1111 1111 1111 1111
+ 0000 0000 0000 0000 0000 0000 0000 0011
= 1000 0000 0000 0000 0000 0000 0000 0010

The equivalent decimal calculation is displayed:

2147483647 + 3 = -2147483646

The leftmost bit 1 in the total binary value is highlighted. The negative sign in the total decimal value is highlighted.

Step 2: If adding two positive numbers and the sum is negative, overflow occurred.

The following chart is shown:

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	Example
A + B	≥ 0	≥ 0	< 0	Pos + Pos = Neg	Pos + Pos = Pos	10 + 20 = 30

In the Result indicating overflow box, < 0 is highlighted. In the Wrong box, =Neg is highlighted.

Step 3: If adding two negative numbers and the sum is positive, overflow occurred.

The following chart is shown:

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	Example
A + B	< 0	< 0	≥ 0	Neg + Neg = Pos	Neg + Neg = Neg	-10 + -20 = -30

Step 4: If subtracting a negative from a positive but the result is negative, overflow occurred.

The following chart is shown:

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	Example
A - B	≥ 0	< 0	≤ 0	Pos - Neg = Neg	Pos - Neg = Pos	10 - -20 = 30

Step 5: If subtracting a positive from a negative but the result is positive, overflow occurred.

The following chart is shown:

Operation	Operand A	Operand B	Result indicating overflow	Wrong	Right	Example

A - B	< 0	≥ 0	≥ 0	Neg - Pos = Pos	Neg - Pos = Neg	-10 - 20 = -30

Animation captions:

1. Overflow occurs when result exceeds 31 bits, using 32nd bit for the result rather than the sign.
2. If adding two positive numbers and the sum is negative, overflow occurred.
3. If adding two negative numbers and the sum is positive, overflow occurred.
4. If subtracting a negative from a positive but the result is negative, overflow occurred.
5. If subtracting a positive from a negative but the result is positive, overflow occurred.

PARTICIPATION ACTIVITY 3.2.6: Overflow.

Indicate if the binary operation resulted in overflow.

1)

```
   0 1 1 0 ...
 + 0 1 1 1 ...
 -------------
   1 1 0 1 ...
```

○ Overflow

○ No overflow

2)

```
   1 1 0 0 ...
 + 1 1 0 1 ...
 -------------
   1 0 0 1 ...
```

○ Overflow

○ No overflow

3)

```
   0 0 0 1 ...
 + 1 0 0 1 ...
 -------------
   1 0 1 0 ...
```

○ Overflow

○ No overflow

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.

COD Appendix B (The Basics of Logic Design) 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. (COD Section 4.9 (Exceptions) covers exceptions in more detail; COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy) describes other situations where exceptions and interrupts occur.) MIPS calls an exception that comes from outside the processor an *interrupt*.

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.

Exception: Also called ***interrupt*** on many computers. An unscheduled event that disrupts program execution; used to detect overflow, for example.

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

PARTICIPATION ACTIVITY 3.2.7: Unsigned addition and subtraction.

1) The MIPS add, addi, and sub instructions may result in an exception.

○ True

○ False

2) The MIPS addu, addiu, and subu

instructions may result in an exception.

○ True

○ False

3) The MIPS addu, addiu, and subu instructions may result in an overflow.

○ True

○ False

4) The MIPS addu, addiu, and subu instructions can only operate on unsigned operands.

○ True

○ False

5) A compiler for the C language never generates add, addi, or sub instructions.

○ True

○ False

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.

PARTICIPATION ACTIVITY 3.2.8: Check yourself: Integer arithmetic for byte and halfword variables.

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 COD Chapter 2 (Instructions: Language of the Computer), MIPS does have data transfer operations for bytes and halfwords.

1) Which MIPS load instructions should be generated for byte and halfword

arithmetic operations?

○ lbu, lhu

○ lb, lh

2) Which MIPS arithmetic instructions should be generated for byte and halfword arithmetic operations?

○ add, sub, mult, div

○ add, sub, mult, div, using AND to mask result to 8 or 16 bits after each operation

3) Which MIPS store instructions should be generated for byte and halfword arithmetic operations?

○ sbu, shu

○ 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 COD Chapter 2 (Instructions: Language of the Computer) 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 overfl
```

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 `$at` so that the assembler can use it as a temporary register (see Hardware/Software Interface in COD Section 2.10 (MIPS Addressing for 32-bit Immediates and Addresses)), compilers also abstain from using registers `$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 log2 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 COD Section B.6 (Faster Addition: Carry Lookahead) in COD Appendix B (The Basics of Logic Design) describes.

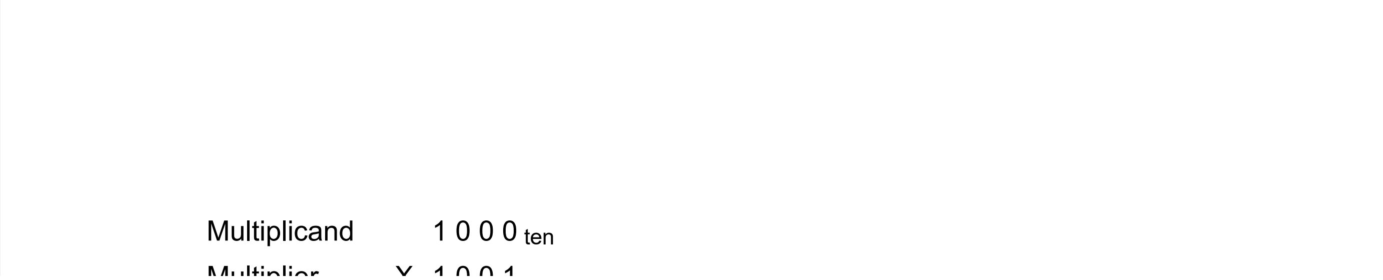
3.3 Multiplication

> “ Multiplication is vexation, Division is as bad; The rule of three doth puzzle me, And practice drives me mad.
> *Anonymous, Elizabethan manuscript, 1570*

Now that we have completed the explanation of addition and subtraction, we are ready to build the more vexing operation of multiplication.

First, let's review the multiplication of decimal numbers in longhand to remind ourselves of the steps of multiplication and the names of the operands. For reasons that will become clear shortly, we limit this decimal example to using only the digits 0 and 1. The following animation illustrates multiplying 1000_{ten} by 1001_{ten}.

PARTICIPATION ACTIVITY 3.3.1: Multiplication example.



Multiplicand $1\,0\,0\,0_{ten}$

Multiplier X 1 0 0 1

Animation content:

Static figure: A long multiplication setup between two decimal numbers. The multiplicand is 1 0 0 0 base ten and the multiplier is 1 0 0 1 base ten. The two numbers are separated by a multiplication symbol "X". The steps between the equation and the product include four decimal numbers placed vertically: 1 0 0 0 aligned with the multiplicand and the multiplier, 0 0 0 0 shifted to the left by one digit, 0 0 0 0 shifted to the left by two digits, and 1 0 0 0 shifted to the left by three digits. The product is 1 0 0 1 0 0 0 base ten.

Step 1: The first operand is called the multiplicand and the second the multiplier.
A long multiplication setup between two decimal numbers. The multiplicand is 1 0 0 0 base ten and the multiplier is 1 0 0 1 base ten. The two numbers are separated by a multiplication symbol "X".

Step 2: Digits are multiplied one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier.
The multiplication starts with the rightmost digit of the multiplier, which is 1. This digit is multiplied by each digit of the multiplicand, starting from the rightmost digit to the leftmost digit. The resulting number is 1 0 0 0.

Step 3: The intermediate products are shifted one digit to the left of the earlier intermediate products.
The second line of the intermediate products results from multiplying the multiplicand by the second rightmost digit of the multiplier, which is 0. The result is then shifted one digit to the left compared to the first line of the intermediate products. Each subsequent line is further shifted to the left as the multiplicand is multiplied by each digit of the multiplier, moving from right to left.

Step 4: Intermediate products are added to determine the final product.
All the rows of intermediate products are summed to find the final product. The final product is 1 0 0 1 0 0 0 base ten.

Animation captions:

1. The first operand is called the multiplicand and the second the multiplier.
2. Digits are multiplied one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier.
3. The intermediate products are shifted one digit to the left of the earlier intermediate products.
4. Intermediate products are added to determine the final product.

The first operand is called the *multiplicand* and the second the *multiplier*. The final result is called the *product*. As you may recall, the algorithm learned in grammar school is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by the single digit of the multiplier, and shifting the intermediate product one digit to the left of the earlier intermediate products.

The first observation is that the number of digits in the product is considerably larger than the number in either the multiplicand or the multiplier. In fact, if we ignore the sign bits, the length of the multiplication of an *n*-bit multiplicand and an *m*-bit multiplier is a product that is *n* + *m* bits long. That is, *n* + *m* bits are required to represent all possible products. Hence, like add, multiply must cope with overflow because we frequently want a 32-bit product as the result of multiplying two 32-bit numbers.

In this example, we restricted the decimal digits to 0 and 1. With only two choices, each step of the multiplication is simple:

1. Just place a copy of the multiplicand (1 × multiplicand) in the proper place if the multiplier digit is a 1, or
2. Place 0 (0 × multiplicand) in the proper place if the digit is 0.

Although the decimal example above happens to use only 0 and 1, multiplication of binary numbers must always use 0 and 1, and thus always offers only these two choices.

PARTICIPATION ACTIVITY 3.3.2: Binary multiplication.

Consider $13_{ten} \times 6_{ten}$, or $1101_{two} \times 0110_{two}$. Fill in the missing values.

```
         1 1 0 1   (Multiplicand)
x        0 1 1 0   (Multiplier)
-----------------
         ? ? ? ?   (Partial product 1)
       ? ? ? ?     (Partial product 2)
     ? ? ? ?       (Partial product 3)
+  ? ? ? ?         (Partial product 4)
```

```
-----------------
   ? ? ? ? ? ? ?  (Product)
```

1) Partial product 1

○ 0000

○ 1101

2) Partial product 2

○ 0000

○ 1101

3) Partial product 3

○ 0000

○ 1101

4) Partial product 4

○ 0000

○ 1101

5) Product

○ 11010

○ 1001110

6) The largest product resulting from a multiplication of a 7-bit multiplicand and a 7-bit multiplier is _____ bits long.

○ 14

○ 35

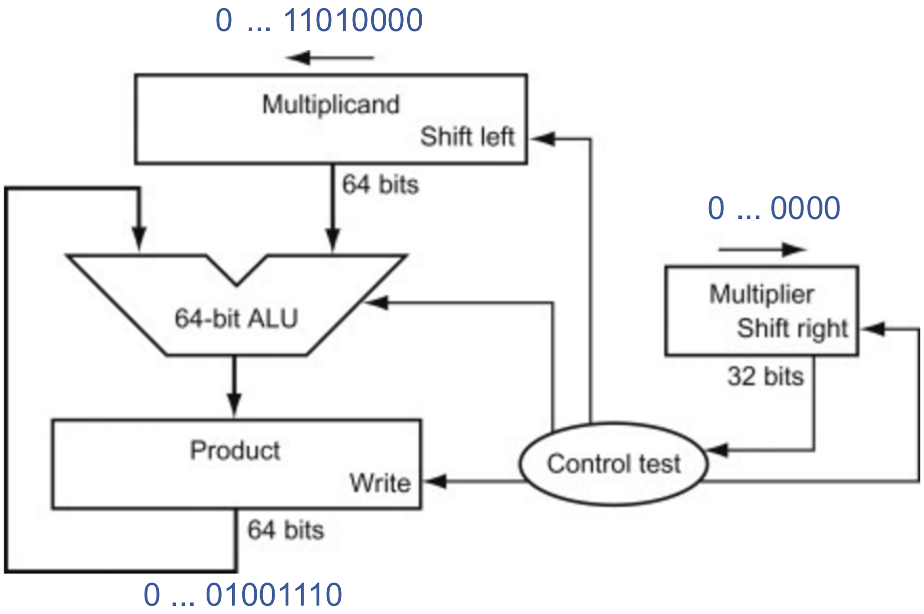
Now that we have reviewed the basics of multiplication, the traditional next step is to provide the highly optimized multiply hardware. We break with tradition in the belief that you will gain a better understanding by seeing the evolution of the multiply hardware and algorithm through multiple generations. For now, let's assume that we are multiplying only positive numbers.

Sequential version of the multiplication algorithm and hardware

This design mimics the algorithm we learned in grammar school; the figure below shows the hardware. We have drawn the hardware so that data flows from top to bottom to resemble more closely the paper-and-pencil method.

The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits. (COD Appendix A, The Basics of Logic Design, describes ALUs.) The 32-bit multiplicand starts in the right half of the Multiplicand register and is shifted left 1 bit on each step. The multiplier is shifted in the opposite direction at each step. The algorithm starts with the product initialized to 0. Control decides when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.

PARTICIPATION ACTIVITY 3.3.3: First version of the multiplication hardware for 32-bit multiplicand and multiplier (COD Figure 3.3).



Animation content:

Static figure: A hardware diagram for a sequential multiplication algorithm. The diagram includes five labeled components: Multiplicand register (64 bits), ALU (64-bit), Product register (64 bits), Multiplier register (32 bits), and Control Test. The number 0 . . . 1 1 0 1 0 0 0 0 is above the Multiplicand with an arrow pointing left. The number 0 . . . 0 0 0 0 is above the Multiplier with an arrow pointing right. The number 0 . . . 0 1 0 0 1 1 1 0 is below the Product. The Multiplicand has an arrow pointing to an input of the ALU. The output of the ALU has an arrow pointing to the Product. The Product has an arrow pointing to the other input of the ALU. The Multiplier has an arrow pointing to the Control Test. The Control Test has arrows pointing to the Multiplicand, ALU, Product, and Multiplier. The Multiplicand register has a shift-left enabler, and the Multiplier register has a shift-right enabler.

Step 1: The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
The Multiplicand register, ALU, and Product register are all highlighted. The Multiplier register is highlighted.

Step 2: The multiplication operation starts with the product initialized to 0. The 32-bit multiplicand starts in the right half of the Multiplicand register.
0 . . . 0 0 0 0 0 0 0 0 is stored in the Product register. 0 . . . 0 0 0 0 1 1 0 1 is stored in the Multiplicand register. 0 . . . 0 1 1 0 is stored in the Multiplier register.

Step 3: Control checks Multiplier(0) to determine when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.
The Control test and the rightmost bit of the number in the Multiplier register are highlighted. The rightmost bit of the number, which is 0, is sent to the Control test.

Step 4: If Multiplier(0) is 0, then Multiplicand register is shifted left 1 bit and the Multiplier is shifted right 1 bit.
The number in the Multiplicand register is shifted 1 bit to the left resulting in 0 . . . 0 0 0 1 1 0 1 0. The number in the Multiplier register is shifted one bit to the right resulting in 0 . . . 0 0 1 1.

Step 5: If Multiplier(0) is 1, then Multiplicand and product registers are added and placed in the Product register.
The Control test and the rightmost bit of the number in the Multiplier register are highlighted. The rightmost bit of the number, which is 1, is sent to the Control test. The 0 . . . 0 0 0 0 0 0 0 0 from the Product register are sent to the input of the ALU, and the 0 . . . 0 0 0 1 1 0 1 0 from the Multiplicand register are sent to the other input of the ALU. The result of the ALU is 0 . . . 0 0 0 1 1 0 1 0 and is sent to the Product register.

Step 6: The Multiplicand register is again shifted left 1 bit and the Multiplier is shifted right 1 bit.
The number in the Multiplicand register is shifted 1 bit to the left resulting in 0 . . . 0 0 1 1 0 1 0 0. The number in the Multiplier register is shifted one bit to the right resulting in 0 . . . 0 0 0 1.

Step 7: Process continues 32 times to obtain the product.
The 1 at the rightmost bit of the number in the Multiplier register is sent to the Control test. The number in the Multiplicand register is shifted 1 bit to the left resulting in 0 . . . 0 1 1 0 1 0 0 0. The number in the Multiplier register is shifted one bit to the right resulting in 0 . . . 0 0 0 0. The 0 at the rightmost bit of the number in the Multiplier register is sent to the Control test. The number in the Multiplicand register is shifted 1 bit to the left resulting in 0 . . . 1 1 0 1 0 0 0 0. The number in the Multiplier register is shifted one bit to the right resulting in 0 . . . 0 0 0 0. The process repeats until all 32 bits of the number in the Multiplier register are processed.

Animation captions:

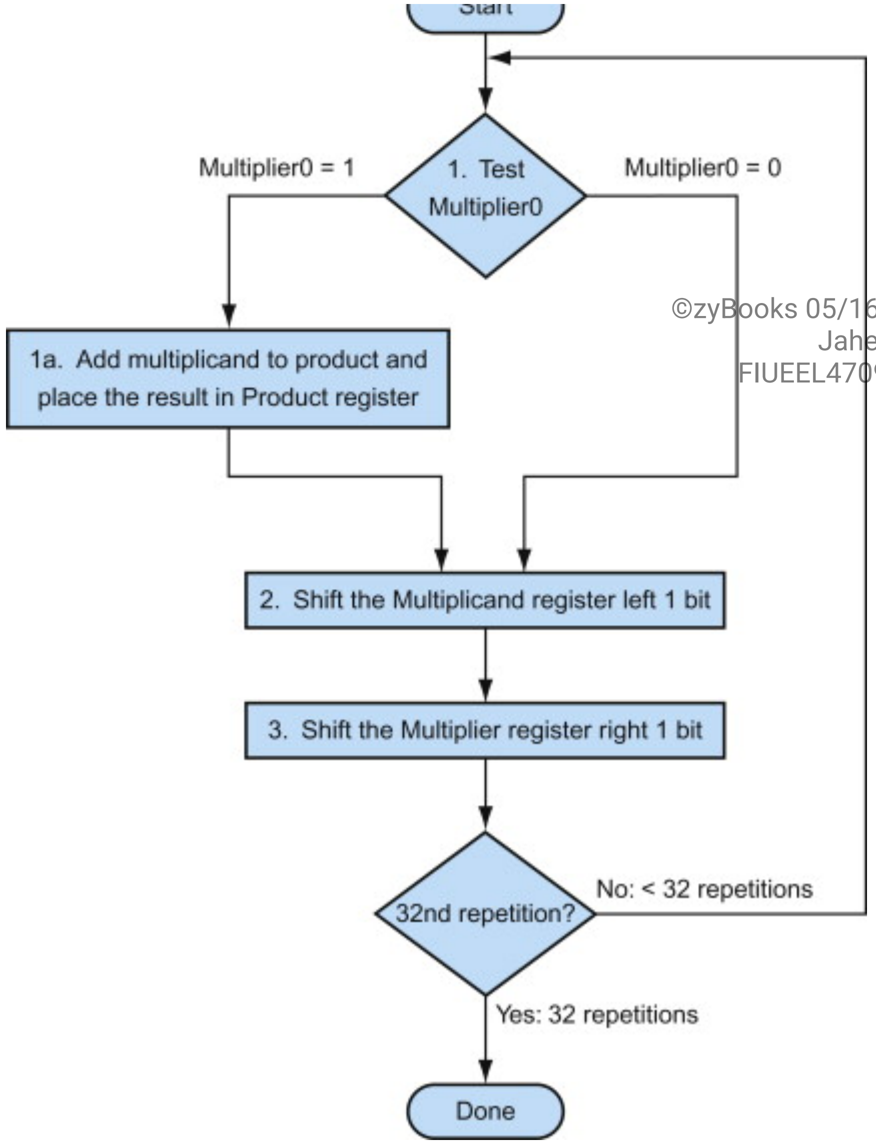
1. The Multiplicand register, ALU, and Product register are all 64 bits wide, with only the Multiplier register containing 32 bits.
2. The multiplication operation starts with the product initialized to 0. The 32-bit multiplicand starts in the right half of the Multiplicand register.
3. Control checks Multiplier0 to determine when to shift the Multiplicand and Multiplier registers and when to write new values into the Product register.
4. If Multiplier0 is 0, then Multiplicand register is shifted left 1 bit and the Multiplier is shifted right 1 bit.
5. If Multiplier0 is 1, then Multiplicand and product registers are added and placed in the Product register.
6. The Multiplicand register is again shifted left 1 bit and the Multiplier is shifted right 1 bit.
7. Process continues 32 times to obtain the product.

Let's assume that the multiplier is in the 32-bit Multiplier register and that the 64-bit Product register is initialized to 0. From the paper-and-pencil example above, it's clear that we will need to move the multiplicand left one digit each step, as it may be added to the intermediate products. Over 32 steps, a 32-bit multiplicand would move 32 bits to the left. Hence, we need a 64-bit Multiplicand register, initialized with the 32-bit multiplicand in the right half and zero in the left half. This register is then shifted left 1 bit each step to align the multiplicand with the sum being accumulated in the 64-bit Product register.

The figure below shows the three basic steps needed for each bit. The least significant bit of the multiplier (Multiplier0) determines whether the multiplicand is added to the Product register. The left shift in step 2 has the effect of moving the intermediate operands to the left, just as when multiplying with paper and pencil. The shift right in step 3 gives us the next bit of the multiplier to examine in the following iteration. These three steps are repeated 32 times to obtain the product. If each step took a clock cycle, this algorithm would require almost 100 clock cycles to multiply two 32-bit numbers. The relative importance of arithmetic operations like multiply varies with the program, but addition and subtraction may be anywhere from 5 to 100 times more popular than multiply. Accordingly, in many applications, multiply can take multiple clock cycles without significantly affecting performance. Yet Amdahl's Law (see COD Section 1.10 (Fallacies and pitfalls)) reminds us that even a moderate frequency for a slow operation can limit performance.

Figure 3.3.1: The first multiplication algorithm, using the hardware shown in the above figure (COD Figure 3.4).

If the least significant bit of the multiplier is 1, add the multiplicand to the product. If not, go to the next step. Shift the multiplicand left and the multiplier right in the next two steps. These three steps are repeated 32 times.


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

PARTICIPATION ACTIVITY 3.3.4: Sequential multiplication algorithm and hardware.

1) Each step of the multiplication algorithm shifts the Multiplier register 1 bit to the _____.

○ right

○ left

2) The Multiplier register is _____-bits wide.

○ 32

○ 64

3) Each step of the multiplication algorithm shifts the Multiplicand register 1 bit to the _____.

   ○ right

   ○ left

4) The Multiplicand register is _____-bits wide.

   ○ 32

   ○ 64

5) The Product register is _____-bits wide.

   ○ 32

   ○ 64

   ○ 128

6) Each iteration of the multiplication algorithm consists of _____ basic steps.

   ○ 3

   ○ 7

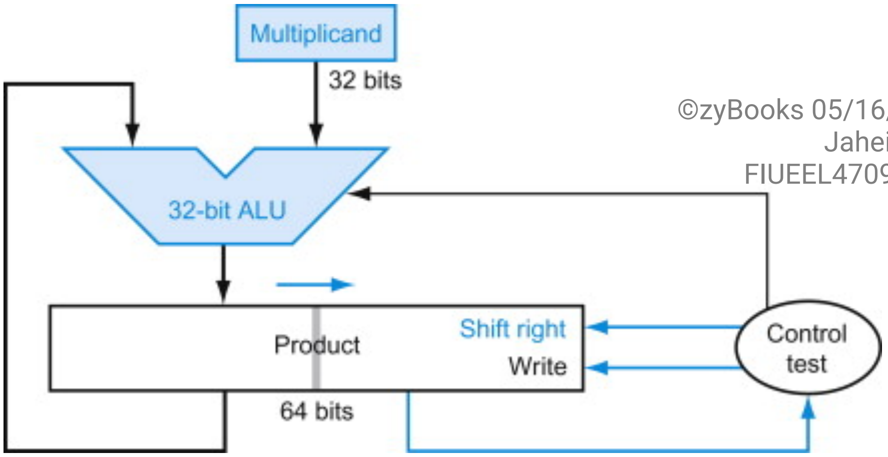
   ○ 32

This algorithm and hardware are easily refined to take one clock cycle per step. The speedup comes from performing the operations in parallel: the multiplier and multiplicand are shifted while the multiplicand is added to the product if the multiplier bit is a 1. The hardware only has to ensure that it tests the right bit of the multiplier and gets the preshifted version of the multiplicand. The hardware is usually further optimized to halve the width of the adder and registers by noticing where there are unused portions of registers and adders. The figure below shows the revised hardware.

Figure 3.3.2: Refined version of the multiplication hardware (COD Figure 3.5).

Compare with the first version in COD Figure 3.3 (First version of the multiplication hardware). The Multiplicand register, ALU, and Multiplier register are all 32 bits wide, with only the Product register left at 64 bits. Now the product is shifted right. The separate Multiplier register also disappeared. The multiplier is placed instead in the right half of the

Product register. These changes are highlighted in color. (The Product register should really be 65 bits to hold the carry out of the adder, but it's shown here as 64 bits to highlight the evolution from COD Figure 3.3 (First version of the multiplication hardware).)



Hardware/Software Interface

Replacing arithmetic by shifts can also occur when multiplying by constants. Some compilers replace multiplies by short constants with a series of shifts and adds. Because one bit to the left represents a number twice as large in base 2, shifting the bits left has the same effect as multiplying by a power of 2. As mentioned in COD Chapter 2 (Instructions: Language of the Computer), almost every compiler will perform the strength reduction optimization of substituting a left shift for a multiply by a power of 2.

PARTICIPATION ACTIVITY 3.3.5: Refined multiplication hardware.

Refer to the refined multiplication hardware figure above (COD Figure 3.5).

1) The refined multiplication hardware halves the width of the Multiplicand register from 64-bits to 32-bits.

○ True

○ False

2) The Multiplier register is removed and placed inside of the _____ register.

○ Product

○ Multiplicand

3) The ALU adds the 64-bit Product and 32-bit Multiplicand, and then stores the result into the Product register.

○ True

○ False

Example 3.3.1: A multiply algorithm.

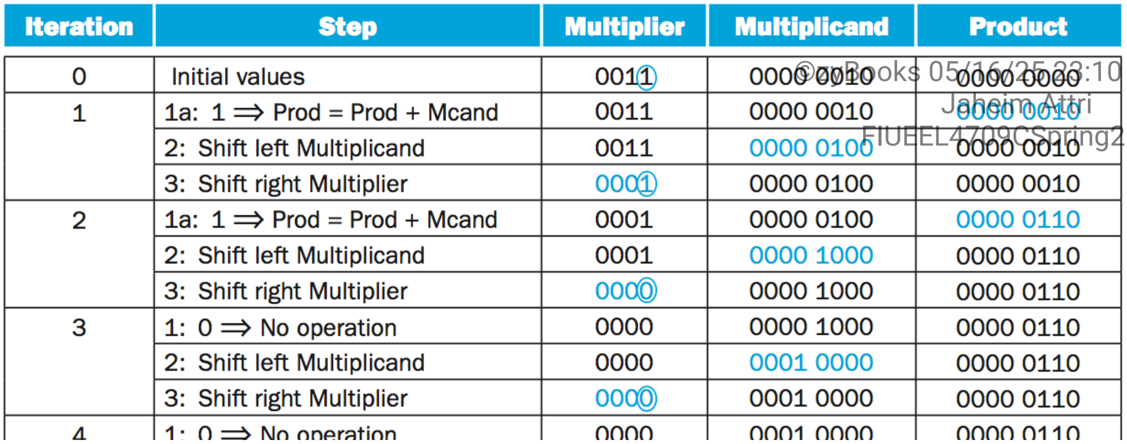
Using 4-bit numbers to save space, multiply 2_{ten} × 3_{ten}, or 0010_{two} × 0011_{two}.

Answer

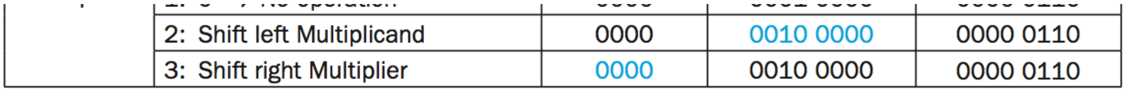
The figure below shows the value of each register for each of the steps labeled according to the figure above (The first multiplication algorithm ...), with the final value of 0000 0110_{two} or 6_{ten}. Color is used to indicate the register values that change on that step, and the bit circled is the one examined to determine the operation of the next step.

Figure 3.3.3: Multiply example using algorithm in COD Figure 3.4 (The first multiplication algorithm) (COD Figure 3.6).

The bit examined to determine the next step is circled in color.



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	001①	0000 0010	0000 0000
1	1a: 1 ⇒ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	000①	0000 0100	0000 0010
2	1a: 1 ⇒ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	000⓪	0000 1000	0000 0110
3	1: 0 ⇒ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	000⓪	0001 0000	0000 0110
4	1: 0 ⇒ No operation	0000	0001 0000	0000 0110



	1: 0 ⇒ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

PARTICIPATION ACTIVITY

3.3.6: Multiply example using the first multiplication algorithm.

Consider the table in the above figure.

1) The initial 8-bit value of Product is _____.

Check Show answer

2) At the end of iteration 1, the 8-bit value of Product is _____.

Check Show answer

3) At the end of iteration 2, the 4-bit value of Multiplier is _____.

Check Show answer

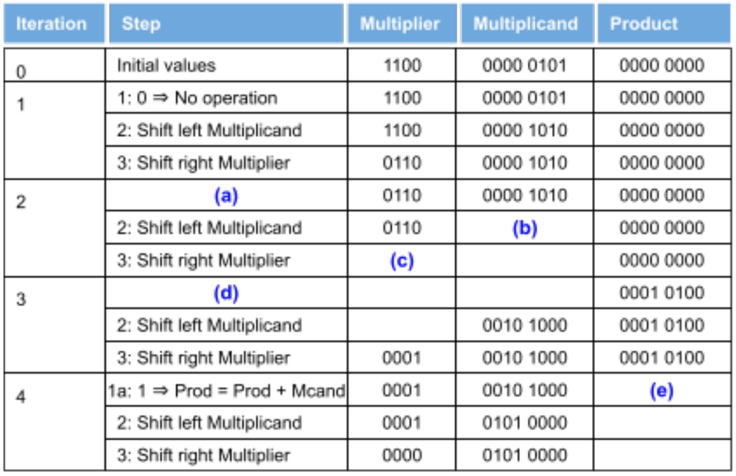
4) At the end of iteration 2, the 8-bit value of Multiplicand is _____.

Check Show answer

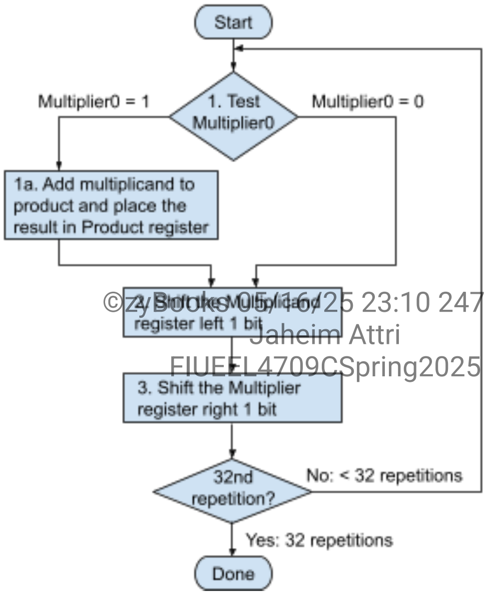
PARTICIPATION ACTIVITY

3.3.7: Multiplication algorithm steps and register values.

Consider the multiplication of $5_{10} \times 12_{10}$, or $0101_2 \times 1100_2$. Fill in the missing values for each of the steps labeled according to the figure above (The first multiplication algorithm …). A copy of the multiplication algorithm figure is shown below to the right.



Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	1100	0000 0101	0000 0000
1	1: 0 ⇒ No operation	1100	0000 0101	0000 0000
	2: Shift left Multiplicand	1100	0000 1010	0000 0000
	3: Shift right Multiplier	0110	0000 1010	0000 0000
2	(a)	0110	0000 1010	0000 0000
	2: Shift left Multiplicand	0110	(b)	0000 0000
	3: Shift right Multiplier	(c)		0000 0000
3	(d)			0001 0100
	2: Shift left Multiplicand		0010 1000	0001 0100
	3: Shift right Multiplier	0001	0010 1000	0001 0100
4	1a: 1 ⇒ Prod = Prod + Mcand	0001	0010 1000	(e)
	2: Shift left Multiplicand	0001	0101 0000	
	3: Shift right Multiplier	0000	0101 0000	


Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

How to use this tool

1a: 1 ⇒ Prod = Prod + Mcand **0011 1100** **0001 0100** **1: 0 ⇒ No operation**

0011

(a)

(b)

(c)

(d)

(e)

Reset

Signed multiplication

So far, we have dealt with positive numbers. The easiest way to understand how to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original signs. The algorithms should then be run for 31 iterations, leaving the signs out of the calculation. As we learned in grammar school, we need negate the product only if the original signs disagree.

It turns out that the last algorithm will work for signed numbers, provided that we remember that we are dealing with numbers that have infinite digits, and we are only representing them with 32 bits. Hence, the shifting steps would need to extend the sign of the product for signed numbers. When the algorithm completes, the lower word would have the 32-bit product.

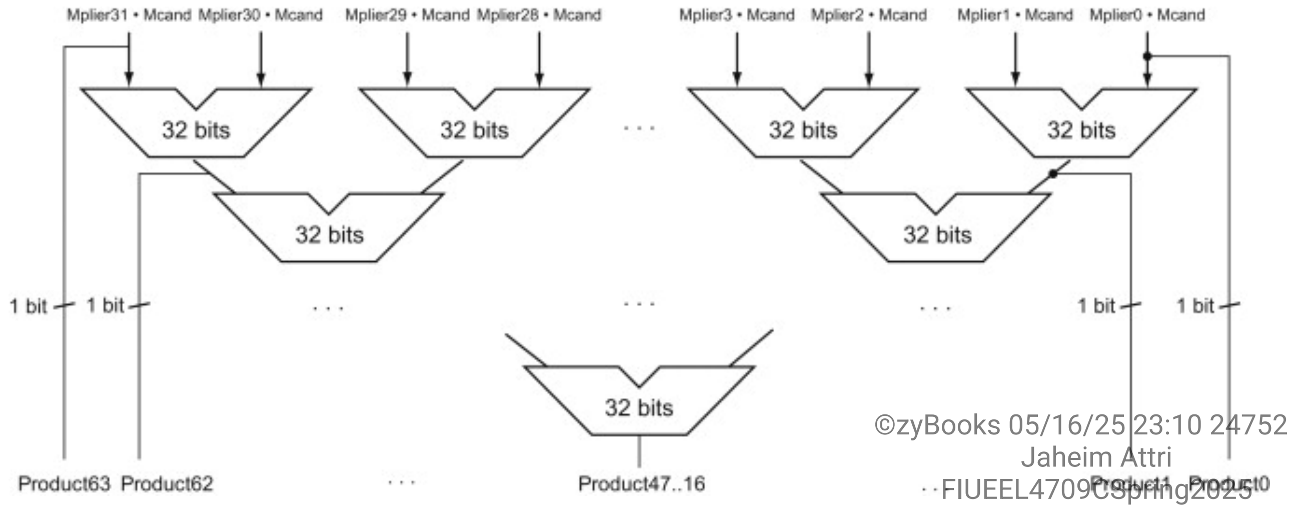
Faster multiplication

Moore's Law has provided so much more in resources that hardware designers can now build much faster multiplication hardware. Whether the multiplicand is to be added or not is known at the beginning of the multiplication by looking at each of the 32 multiplier bits. Faster multiplications are possible by essentially providing one 32-bit adder for each bit of the multiplier: one input is the multiplicand ANDed with a multiplier bit, and the other is the output of a prior adder.

A straightforward approach would be to connect the outputs of adders on the right to the inputs of adders on the left, making a stack of adders 32 high. An alternative way to organize these 32 additions is in a parallel tree, as the figure below shows. Instead of waiting for 32 add times, we wait just the $\log_2$ (32) or five 32-bit add times.

Figure 3.3.4: Fast multiplication hardware (COD Figure 3.7).

Rather than use a single 32-bit adder 31 times, this hardware "unrolls the loop" to use 31 adders and then organizes them to minimize delay.



In fact, multiply can go even faster than five add times because of the use of *carry save adders* (see COD Section B.6 (Faster Addition: Carry Lookahead) in COD Appendix B, (The Basics of Logic Design)) and because it is easy to **pipeline** such a design to be able to support many multiplies simultaneously (see COD Chapter 4

(The Processor)).

Multiply in MIPS

MIPS provides a separate pair of 32-bit registers to contain the 64-bit product, called *Hi* and *Lo*. To produce a properly signed or unsigned product, MIPS has two instructions: multiply (`mult`) and multiply unsigned (`multu`). To fetch the integer 32-bit product, the programmer uses *move from lo* (`mflo`). The MIPS assembler generates a pseudoinstruction for multiply that specifies three general-purpose registers, generating `mflo` and `mfhi` instructions to place the product into registers.

Summary

Multiplication hardware simply shifts and adds, as derived from the paper-and-pencil method learned in grammar school. Compilers even use shift instructions for multiplications by powers of 2. With much more hardware we can do the adds in **parallel**, and do them much faster.

Hardware/Software Interface

Both MIPS multiply instructions ignore overflow, so it is up to the software to check to see if the product is too big to fit in 32 bits. There is no overflow if Hi is 0 for `multu` or the replicated sign of Lo for `mult`. The instruction *move from hi* (`mfhi`) can be used to transfer Hi to a general-purpose register to test for overflow.

PARTICIPATION ACTIVITY | 3.3.8: MIPS multiplication.

1) The multiplication hardware supports signed multiplication.

○ True

○ False

2) The 32-bit registers, called Hi and Lo, combine to form a 64-bit product register.

○ True

○ False

3) The multiply (`mult`) instruction ignores overflow, while the multiply unsigned (`multu`) instruction detects overflow.

○ True

○ False

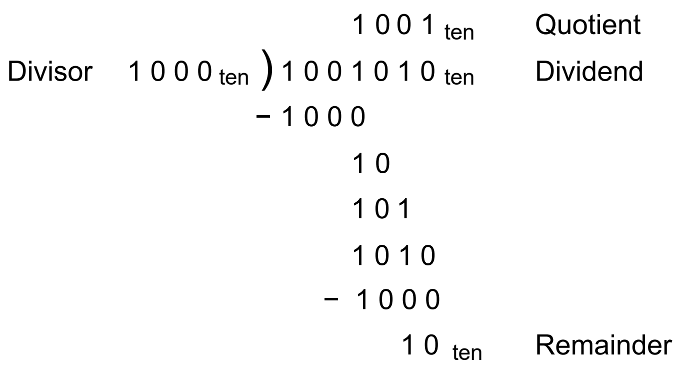
3.4 Division

> Divide et impera.
>
> *Latin for "Divide and rule," ancient political maxim cited by Machiavelli, 1532*

The reciprocal operation of multiply is divide, an operation that is even less frequent and even more quirky. It even offers the opportunity to perform a mathematically invalid operation: dividing by 0.

Let's start with an example of long division using decimal numbers to recall the names of the operands and the grammar school division algorithm. For reasons similar to those in the previous section, we limit the decimal digits to just 0 or 1. The following animation illustrates dividing 1,001,010$_{ten}$ by 1000$_{ten}$.

PARTICIPATION ACTIVITY | 3.4.1: Division example.



Animation content:

Static figure: A long division problem with the divisor 1 0 0 0 base ten, the dividend 1 0 0 1 0 1 0 base ten, the quotient 1 0 0 1 base ten, and remainder 1 0 base ten. The intermediate steps required to complete the problem are also shown.

Step 1: A divide operation's two operands are called the divisor and dividend.
A long division problem with the divisor 1 0 0 0 base ten and the dividend 1 0 0 1 0 1 0 base ten appear.

Step 2: The number of times the divisor goes into a portion of the dividend is determined. The whole number is placed at the top.
The first four digits of the dividend, 1 0 0 1, are highlighted. The whole number 0 0 0 1 appears above the four left-most digits of the dividend.

Step 3: The whole number is multiplied by the divisor, and then subtracted from the corresponding portion of the dividend. Leading zeros are ignored to simplify the division.
The multiplication equation 1 times 1 0 0 0 appears resulting in 1 0 0 0. The product 1 0 0 0 subtracts from the four left-most digits of the dividend, 1 0 0 1, resulting in 1.

Step 4: Subsequent digits are brought down from the dividend until the value is larger than the divisor. Zeros indicate the divisor did not go into the dividend.
The next digit of the dividend, 0, is added as the right-most digit of the difference resulting in 1 0. A 0 appears as the right-most digit of the whole number above the dividend resulting in 1 0. The next digit of the dividend, 1, is added as the right-most digit of the difference resulting in 1 0 1. A 0 appears as the right-most digit of the whole number above the dividend resulting in 1 0 0. The next digit of the dividend, 0, is added as the right-most digit of the difference resulting in 1 0 1 0. A 1 appears as the right-most digit of the whole number above the dividend resulting in 1 0 0 1.

Step 5: The whole number is multiplied by the divisor, and then subtracted from the corresponding portion of the dividend.
The multiplication equation 1 times 1 0 0 0 appears resulting in 1 0 0 0. The product 1 0 0 0 subtracts from the four left-most digits of the dividend, 1 0 1 0, resulting in 1 0.

Step 6: No more digits can be brought down from the dividend, so the division is complete. The result, called the quotient, is accompanied by a second result, called the remainder.
The whole number above the dividend is highlighted and identified as the Quotient. The last difference, 10, is highlighted and identified as the Remainder.

Animation captions:

1. A divide operation's two operands are called the divisor and dividend.
2. The number of times the divisor goes into a portion of the dividend is determined. The whole number is placed at the top.
3. The whole number is multiplied by the divisor, and then subtracted from the corresponding portion of the dividend. Leading zeros are ignored to simplify the division.
4. Subsequent digits are brought down from the dividend until the value is larger than the divisor. Zeros indicate the divisor did not go into the dividend.
5. The whole number is multiplied by the divisor, and then subtracted from the corresponding portion of the dividend.
6. No more digits can be brought down from the dividend, so the division is complete. The result, called the quotient, is accompanied by a second result, called the remainder.

Divide's two operands, called the **dividend** and **divisor**, and the result, called the **quotient**, are accompanied by a second result, called the **remainder**. Here is another way to express the relationship between the components:

$$\text{Dividend} = \text{Quotient} \times \text{Divisor} + \text{Remainder}$$

where the remainder is smaller than the divisor. Infrequently, programs use the divide instruction simply to get the remainder, ignoring the quotient.

Dividend: A number being divided.

Divisor: A number that the dividend is divided by.

Quotient: The primary result of a division; a number that when multiplied by the divisor and added to the remainder produces the dividend.

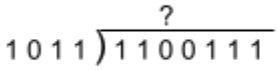
Remainder: The secondary result of a division; a number that when added to the product of the quotient and the divisor produces the dividend.

The basic grammar school division algorithm tries to see how big a number can be subtracted, creating a digit of the quotient on each attempt. Our carefully selected decimal example uses only the numbers 0 and 1, so it's easy to figure out how many times the divisor goes into the portion of the dividend: it's either 0 times or 1 time. Binary numbers contain only 0 or 1, so binary division is restricted to these two choices, thereby simplifying binary division.

PARTICIPATION ACTIVITY | 3.4.2: Binary division.

Consider $103_{ten} \div 11_{ten}$, or $1100111_{two} \div 1011_{two}$. Fill in the missing values.

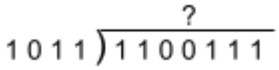
1) The value placed in the quotient is _____.



○ 0

○ 1

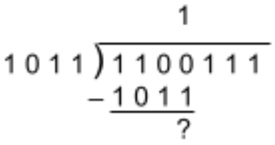
2) The value placed in the quotient is _____.



○ 0

○ 1

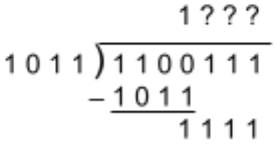
3) The result of the subtraction is _____.



○ 0

○ 1

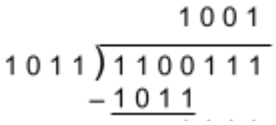
4) The value placed in the quotient is _____.



○ 111

○ 001

5) The remainder is _____.



```
  1 1 1 1
− 1 0 1 1
    ? ? ?
```

○ 000

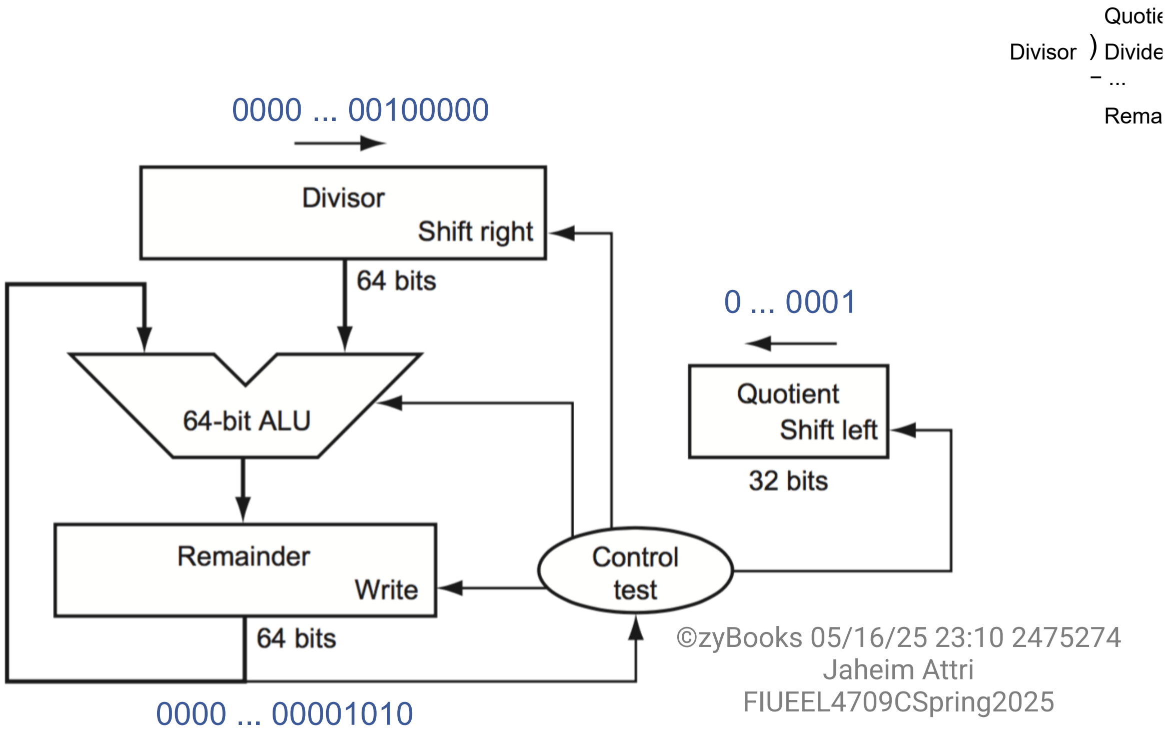
○ 100

Let's assume that both the dividend and the divisor are positive and hence the quotient and the remainder are nonnegative. The division operands and both results are 32-bit values, and we will ignore the sign for now.

A division algorithm and hardware

The figure below shows hardware to mimic our grammar school algorithm. We start with the 32-bit Quotient register set to 0. Each iteration of the algorithm needs to move the divisor to the right one digit, so we start with the divisor placed in the left half of the 64-bit Divisor register and shift it right 1 bit each step to align it with the dividend. The Remainder register is initialized with the dividend.

PARTICIPATION ACTIVITY 3.4.3: First version of the division hardware for 32-bit divisor and dividend (COD Figure 3.8).



Animation content:

Static figure: The five components are the Divisor register (64 bits), Quotient register (32 bits), Remainder register (64 bits), 64-bit ALU, and Control. The inputs to the ALU are the Divisor register and the Remainder register. The output of the ALU is stored in the Remainder register. The Control has one input, test, produced by the Remainder register. The Control has four outputs. The first output controls when the Divisor register is shifted right. The second output controls when the Quotient register is shifted left. The third output controls when the output of the ALU is placed in the Remainder register. The fourth output selects the arithmetic operation of the ALU.

Step 1: The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits.
The Divisor register, ALU, and Remainder register are highlighted. The Quotient register is highlighted.

Step 2: The divisor is placed in the left half of the Divisor register, the dividend is placed in the Remainder, and the Quotient is set to 0.
The number 1 0 0 0 . . . 0 0 0 0 0 0 0 0 appears above the Divisor register. The number 0 0 0 0 . . . 0 1 0 0 1 0 1 0 appears below the Remainder register. The number 0 . . . 0 0 0 0 appears above the Quotient register.

Step 3: The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.
The Divisor register contents and the Remainder register contents are applied to the ALU inputs. The word subtract appears in the ALU. The number 1 1 1 1 . . . 1 0 1 1 0 1 1 0 appears and replaces the previous value stored in the Remainder register.

Step 4: If the result is negative, then the Remainder value is restored by adding the divisor to the Remainder.

Remainder less than 0 appears and moves from the Remainder register to the Control input, test. The Divisor register contents and the Remainder register contents are applied to the ALU inputs. The word add appears in the ALU. The number 1 0 0 0 . . . 0 1 0 0 1 0 1 0 appears and replaces the previous value stored in the Remainder register.

Step 5: A negative Remainder indicates the divisor did not go into the dividend. Control shifts Quotient left 1 bit and places a 0 in the least significant bit.
The number above the Quotient register is shifted left, and a 0 is placed in the least significant bit. The resulting number is 0 . . . 0 0 0 0.

Step 6: Divisor is shifted right 1 bit, and the divide operation repeats.

The number above the Divisor register is shifted right, and a 0 is placed in the most significant bit. The resulting number is 0 1 0 0 . . . 0 0 0 0 0 0 0 0.

Step 7: The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.
The Divisor register contents and the Remainder register contents are applied to the ALU inputs. The word subtract appears in the ALU. The number 0 0 0 0 . . . 0 0 0 0 1 0 1 0 appears and replaces the previous value stored in the Remainder register.

Step 8: If the Remainder is positive, then the divisor was smaller or equal to the dividend. Control shifts Quotient left 1 bit and places a 1 in the least significant bit.

Remainder greater than or equal to 0 appears and moves from the Remainder to the Control input, test. The number above the Quotient register is shifted left, and a 1 is placed in the least significant bit. The resulting number is 0 . . . 0 0 0 1.

Step 9: Divisor is shifted right 1 bit, and the divide operation repeats. The steps repeat a total of 33 times.

The number above the Divisor register is shifted right, and a 0 is placed in the most significant bit. The resulting number is 0 0 0 0 . . . 0 0 1 0 0 0 0 0.

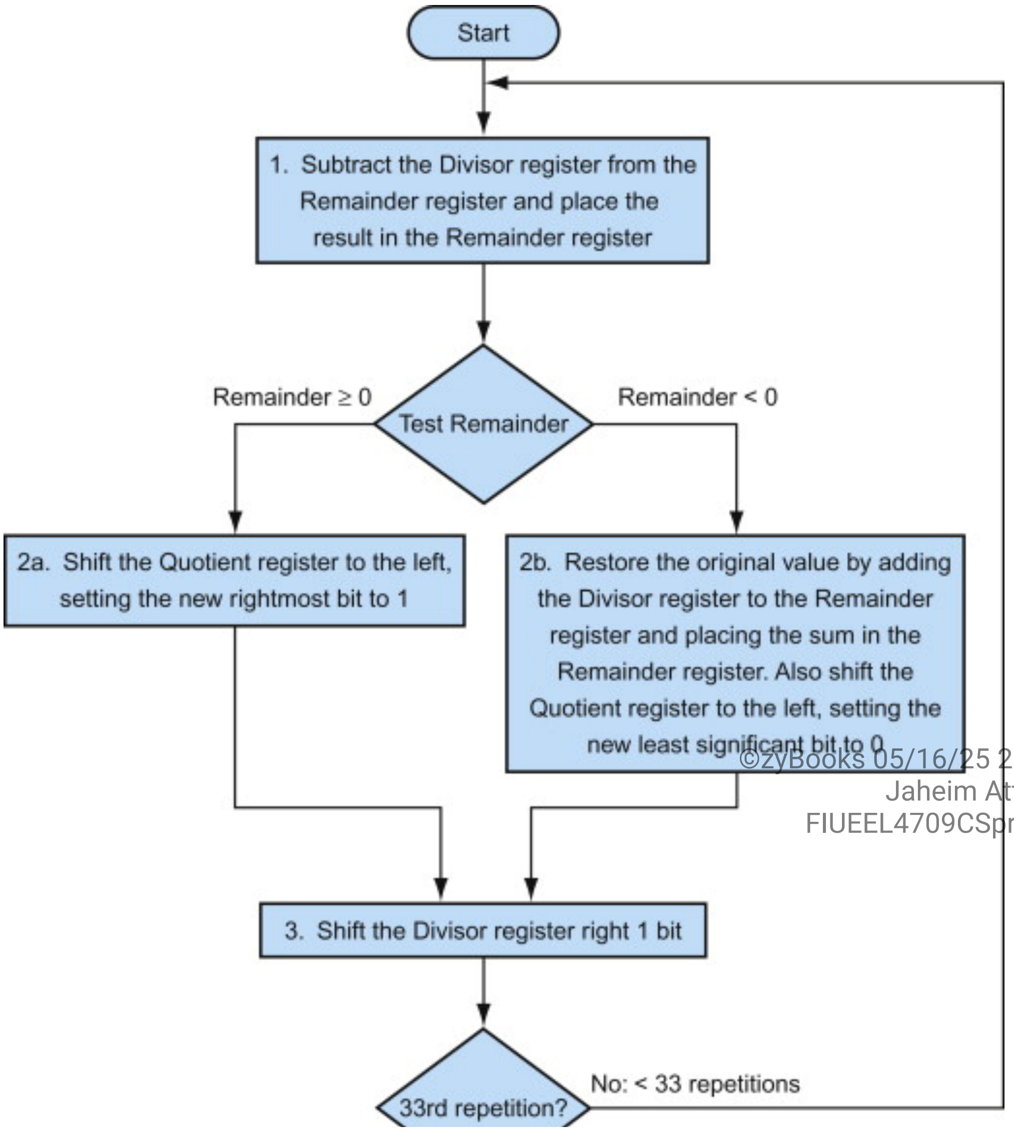
Animation captions:

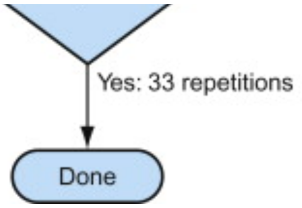
1. The Divisor register, ALU, and Remainder register are all 64 bits wide, with only the Quotient register being 32 bits.
2. The divisor is placed in the left half of Divisor register, the dividend is placed in the Remainder, and the Quotient is set to 0.
3. The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.
4. If the result is negative, then the Remainder value is restored by adding the divisor to the Remainder.
5. A negative Remainder indicates the divisor did not go into the dividend. Control shifts Quotient left 1 bit and places a 0 in the least significant bit.
6. Divisor is shifted right 1 bit and the divide operation repeats.
7. The Divisor is subtracted from the Remainder, then the result is placed into the Remainder.
8. If the Remainder is positive, then the divisor was smaller or equal to the dividend. Control shifts Quotient left 1 bit and places a 1 in the least significant bit.
9. Divisor is shifted right 1 bit and the divide operation repeats. The steps repeat a total of 33 times.

The figure below shows three steps of the first division algorithm. Unlike a human, the computer isn't smart enough to know in advance whether the divisor is smaller than the dividend. It must first subtract the divisor in step 1; remember that this is how we performed the comparison in the set on less than instruction. If the result is positive, the divisor was smaller or equal to the dividend, so we generate a 1 in the quotient (step 2a). If the result is negative, the next step is to restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient (step 2b). The divisor is shifted right, and then we iterate again. The remainder and quotient will be found in their namesake registers after the iterations are complete.

Figure 3.4.1: A division algorithm, using the hardware in the above figure (COD Figure 3.9).

If the remainder is positive, the divisor did go into the dividend, so step 2a generates a 1 in the quotient. A negative remainder after step 1 means that the divisor did not go into the dividend, so step 2b generates a 0 in the quotient and adds the divisor to the remainder, thereby reversing the subtraction of step 1. The final shift, in step 3, aligns the divisor properly, relative to the dividend for the next iteration. These steps are repeated 33 times.


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
No: < 33 repetitions
33rd repetition?



PARTICIPATION ACTIVITY | 3.4.4: Division algorithm and hardware.

1) Each step of the division algorithm shifts the Divisor register 1 bit to the _____.

○ right

○ left

2) If the Remainder is negative, a _____ is shifted into the least significant bit of the Quotient register.

○ 0

○ 1

3) If the Remainder is negative, then the original Remainder value is restored by adding _____ to the Remainder.

○ Control

○ the Divisor

4) Each iteration of the division algorithm consists of _____ basic steps.

○ 3

○ 8

○ 33

Example 3.4.1: A divide algorithm.

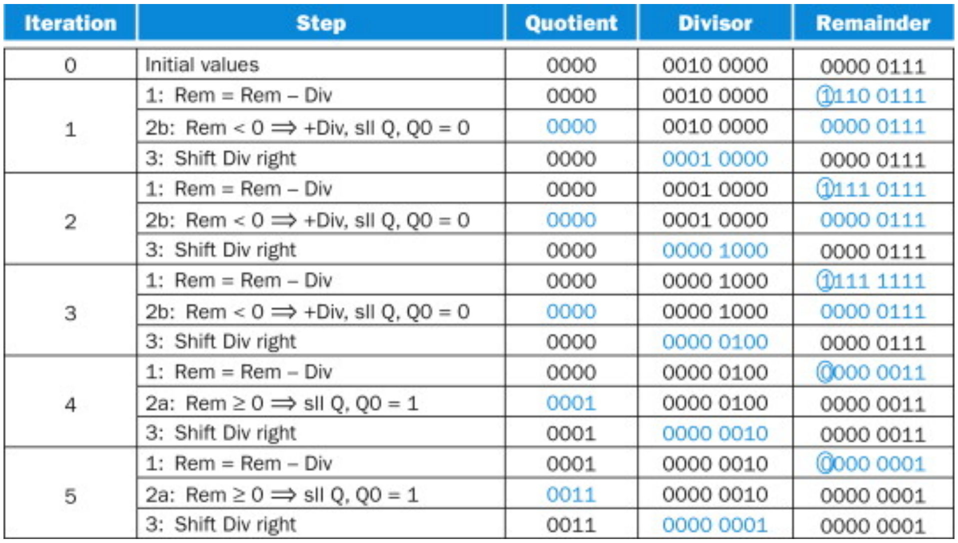
Using a 4-bit version of the algorithm to save pages, let's try dividing 7_{ten} by 2_{ten}, or 0000 0111_{two} by 0010_{two}.

Answer

The figure below shows the value of each register for each of the steps, with the quotient being 3_{ten} and the remainder 1_{ten}. Notice that the test in step 2 of whether the remainder is positive or negative simply tests whether the sign bit of the Remainder register is a 0 or 1. The surprising requirement of this algorithm is that it takes $n + 1$ steps to get the proper quotient and remainder.

Figure 3.4.2: Division example using the algorithm in the above figure (COD Figure 3.10).

The bit examined to determine the next step is circled in color.



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	1110 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	1111 0111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	1111 1111
	2b: Rem < 0 ⟹ +Div, sll Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	0000 0011
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	0000 0001
	2a: Rem ≥ 0 ⟹ sll Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

PARTICIPATION ACTIVITY 3.4.5: Divide example using the division algorithm.

Consider the table in the above figure.

1) The initial 8-bit value of Divisor is _____.

Check Show answer

2) The initial 8-bit value of Remainder is _____.

Check Show answer

3) Iteration 1, step 1 subtracts the Divisor from the Remainder and places the result in the Remainder. The Remainder's updated value is _____.

Check Show answer

4) Iteration 1, step 2b restores the Remainder's value to _____.

Check Show answer

5) At the end of iteration 4, the 4-bit value of Quotient is _____.

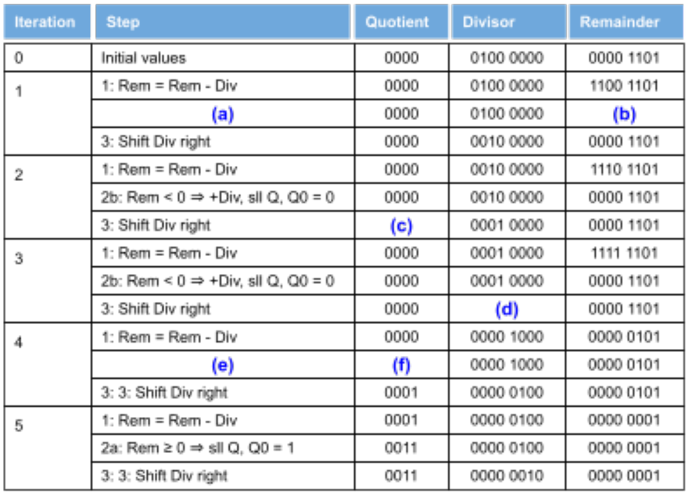
Check Show answer

6) The divide operation results in a 4-bit Quotient of 0011 and an 8-bit Remainder of _____.

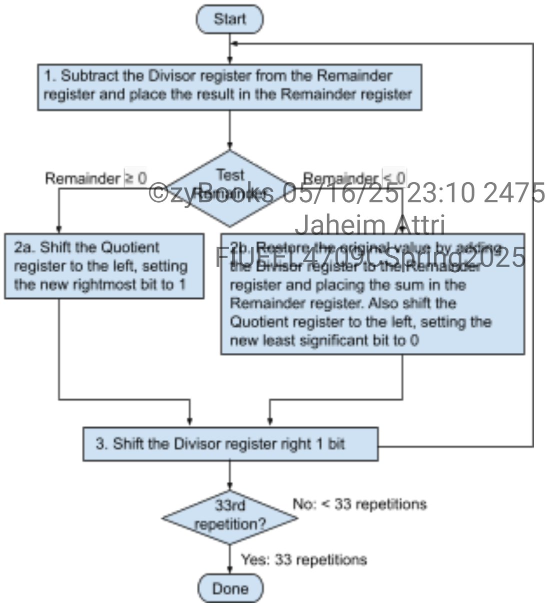
Check Show answer

PARTICIPATION ACTIVITY 3.4.6: Division algorithm steps and register values.

Consider the divison of $13_{10} \div 4_{10}$, or $1101_2 \div 0100_2$. Fill in the missing values for each of the steps labeled according to the figure above (A division algorithm ...). A copy of the division algorithm figure is shown below to the right.



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0100 0000	0000 1101
1	1: Rem = Rem - Div	0000	0100 0000	1100 1101
	(a)	0000	0100 0000	(b)
	3: Shift Div right	0000	0010 0000	0000 1101
2	1: Rem = Rem - Div	0000	0010 0000	1110 1101
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0010 0000	0000 1101
	3: Shift Div right	(c)	0001 0000	0000 1101
3	1: Rem = Rem - Div	0000	0001 0000	1111 1101
	2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0	0000	0001 0000	0000 1101
	3: Shift Div right	0000	(d)	0000 1101
4	1: Rem = Rem - Div	0000	0000 1000	0000 0101
	(e)	(f)	0000 1000	0000 0101
	3: 3: Shift Div right	0001	0000 0100	0000 0101
5	1: Rem = Rem - Div	0001	0000 0100	0000 0001
	2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1	0011	0000 0100	0000 0001
	3: 3: Shift Div right	0011	0000 0010	0000 0001


Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

How to use this tool ⌄

2b: Rem < 0 ⇒ +Div, sll Q, Q0 = 0 **0000 1101** **2a: Rem ≥ 0 ⇒ sll Q, Q0 = 1** **0001**

0000 1000 **0000**

(a)

(b)

(c)

(d)

(e)

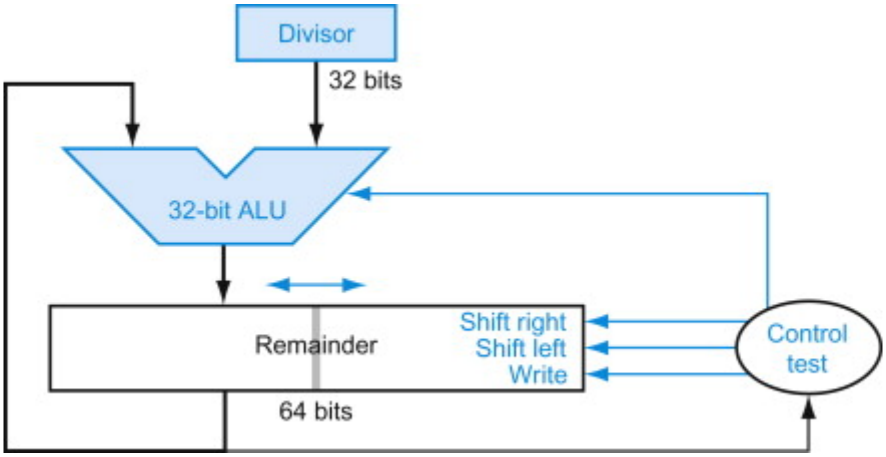
(f)

Reset

This algorithm and hardware can be refined to be faster and cheaper. The speedup comes from shifting the operands and the quotient simultaneously with the subtraction. This refinement halves the width of the adder and registers by noticing where there are unused portions of registers and adders. The figure below shows the revised hardware.

Figure 3.4.3: An improved version of the division hardware (COD Figure 3.11).

The Divisor register, ALU, and Quotient register are all 32 bits wide, with only the Remainder register left at 64 bits. Compared to the figure above (First version of the division hardware), the ALU and Divisor registers are halved and the remainder is shifted left. This version also combines the Quotient register with the right half of the Remainder register. (As in COD Figure 3.5 (Refined version of the multiplication hardware), the Remainder register should really be 65 bits to make sure the carry out of the adder is not lost.)



PARTICIPATION ACTIVITY 3.4.7: Refined division hardware.

Refer to the improved version of the division hardware (figure above) .

1) The improved version of the division hardware does not require a quotient to perform a divide operation.

○ True

○ False

2) The improved version of the division hardware halves the width of the

adder and divisor.

○ True

○ False

3) The speedup comes from reducing the size of the registers and ALU.

○ True

○ False

Signed division

So far, we have ignored signed numbers in division. The simplest solution is to remember the signs of the divisor and dividend and then negate the quotient if the signs disagree.

Elaboration

The one complication of signed division is that we must also set the sign of the remainder. Remember that the following equation must always hold:

$$\mathbf{Dividend = Quotient \times Divisor + Remainder}$$

To understand how to set the sign of the remainder, let's look at the example of dividing all the combinations of ±7$_{ten}$ by ±2$_{ten}$. The first case is easy:

$$\mathbf{+7 \ \div \ +2 : Quotient = +3, \ Remainder = +1}$$

Checking the results:

$$\mathbf{+7 = 3 \ \times \ 2 \ + \ (+1) = 6 \ + \ 1}$$

If we change the sign of the dividend, the quotient must change as well:

$$\mathbf{-7 \ \div \ +2 : Quotient = -3}$$

Rewriting our basic formula to calculate the remainder:

$$\mathbf{Remainder = (Dividend - Quotient \ \times \ Divisor) = -7 \ - \ (-3\boldsymbol{x} \ + \ 2)}$$
$$\mathbf{= -7 \ - \ (-6) = -1}$$

So,

$$-7 \div +2 : \text{Quotient} = -3, \text{Remainder} = -1$$

Checking the results again:

$$-7 = -3 \times 2 + (-1) = -6 - 1$$

The reason the answer isn't a quotient of −4 and a remainder of +1, which would also fit this formula, is that the absolute value of the quotient would then change depending on the sign of the dividend and the divisor! Clearly, if

$$-(x \div y) \neq (-x) \div y$$

programming would be an even greater challenge. This anomalous behavior is avoided by following the rule that the dividend and remainder must have the same signs, no matter what the signs of the divisor and quotient.

We calculate the other combinations by following the same rule:

$$+7 \div -2 : \text{Quotient} = -3, \text{Remainder} = +1$$
$$-7 \div -2 : \text{Quotient} = +3, \text{Remainder} = -1$$

Thus the correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match the dividend.

Faster division

Moore's Law applies to division hardware as well as multiplication, so we would like to be able to speed up division by throwing hardware at it. We used many adders to speed up multiply, but we cannot do the same trick for divide. The reason is that we need to know the sign of the difference before we can perform the next step of the algorithm, whereas with multiply we could calculate the 32 partial products immediately.

There are techniques to produce more than one bit of the quotient per step. The *SRT division* technique tries to **predict** several quotient bits per step, using a table lookup based on the upper bits of the dividend and remainder. It relies on subsequent steps to correct wrong predictions. A typical value today is 4 bits. The key is guessing the value to subtract. With binary division, there is only a single choice. These algorithms use 6 bits from the remainder and 4 bits from the divisor to index a table that determines the guess for each step.

The accuracy of this fast method depends on having proper values in the lookup table. The fallacy in COD Section 3.9 (Fallacies and pitfalls) shows what can happen if the table is incorrect.

Divide in MIPS

You may have already observed that the same sequential hardware can be used for both multiply and divide in COD Figures 3.5 (Refined version of the multiplication hardware) and the figure above (An improved version of the division hardware). The only requirement is a 64-bit register that can shift left or right and a 32-bit ALU that adds or subtracts. Hence, MIPS uses the 32-bit Hi and 32-bit Lo registers for both multiply and divide.

As we might expect from the algorithm above, Hi contains the remainder, and Lo contains the quotient after the divide instruction completes.

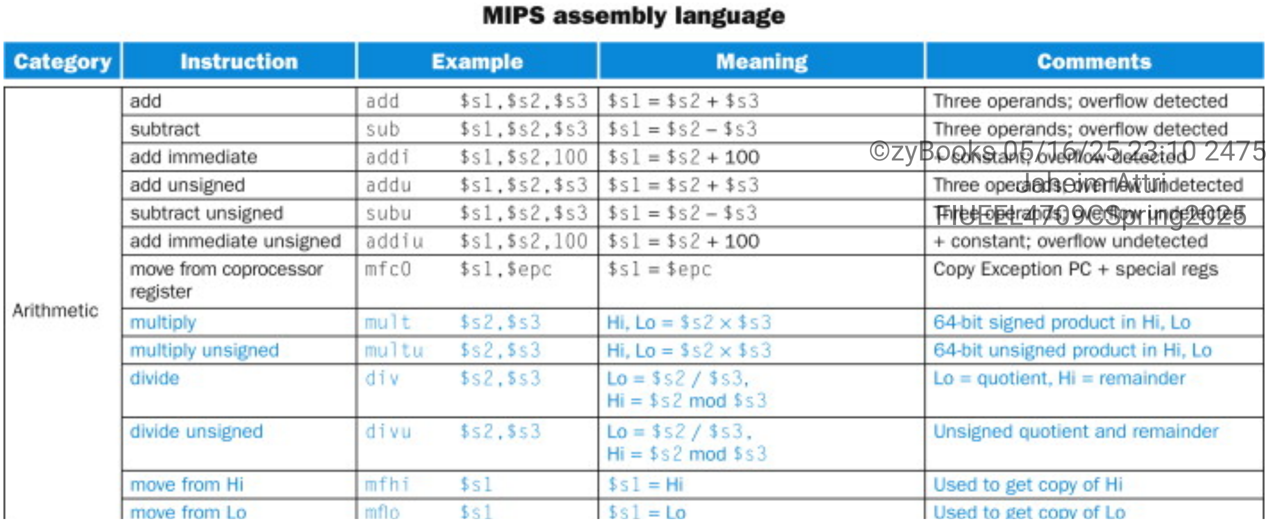
To handle both signed integers and unsigned integers, MIPS has two instructions: *divide* (`div`) and *divide unsigned* (`divu`). The MIPS assembler allows divide instructions to specify three registers, generating the `mflo` or `mfhi` instructions to place the desired result into a general-purpose register.

Summary

The common hardware support for multiply and divide allows MIPS to provide a single pair of 32-bit registers that are used both for multiply and divide. We accelerate division by predicting multiple quotient bits and then correcting mispredictions later, the figure below summarizes the enhancements to the MIPS instruction set for COD Section 3.3 (Multiplication) and COD Section 3.4 (Division).

Figure 3.4.4: MIPS core architecture (COD Figure 3.12).

The memory and registers of the MIPS architecture are not included for space reasons, but this section added the Hi and Lo registers to support multiply and divide.



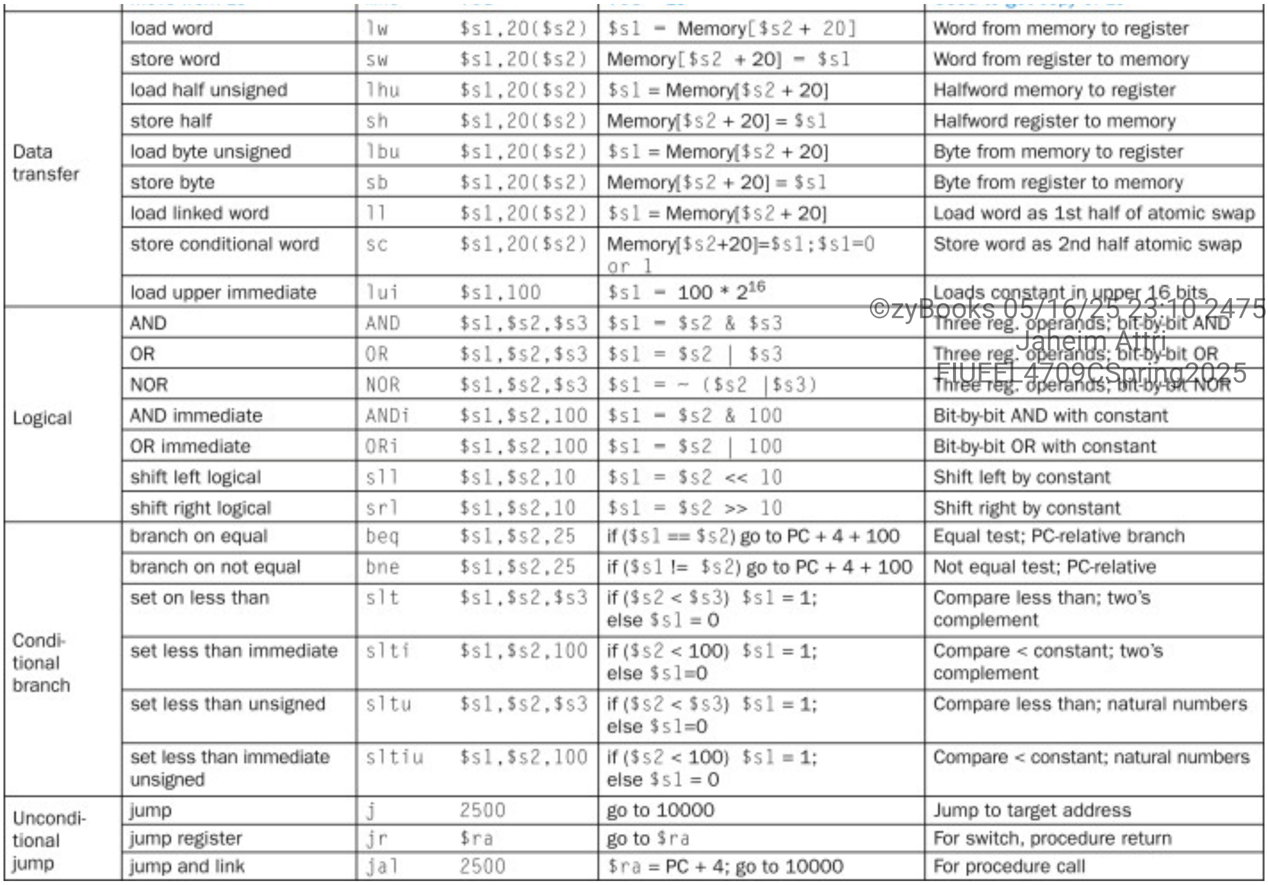
MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow detected
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow detected
	add immediate	addi $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow detected
	add unsigned	addu $s1,$s2,$s3	$s1 = $s2 + $s3	Three operands; overflow undetected
	subtract unsigned	subu $s1,$s2,$s3	$s1 = $s2 – $s3	Three operands; overflow undetected
	add immediate unsigned	addiu $s1,$s2,100	$s1 = $s2 + 100	+ constant; overflow undetected
	move from coprocessor register	mfc0 $s1,$epc	$s1 = $epc	Copy Exception PC + special regs
	multiply	mult $s2,$s3	Hi, Lo = $s2 × $s3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $s2,$s3	Hi, Lo = $s2 × $s3	64-bit unsigned product in Hi, Lo

	divide	div $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Lo = quotient, Hi = remainder

	divide unsigned	divu $s2,$s3	Lo = $s2 / $s3,
			Hi = $s2 mod $s3
				Unsigned quotient and remainder

	move from Hi	mfhi $s1	$s1 = Hi	Used to get copy of Hi
	move from Lo	mflo $s1	$s1 = Lo	Used to get copy of Lo



Data transfer	load word	lw	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw	$s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half unsigned	lhu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh	$s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte unsigned	lbu	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb	$s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll	$s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store conditional word	sc	$s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half atomic swap
	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
Condi-tional 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; natural numbers

	set less than immediate unsigned	sltiu	$s1,$s2,100	if ($s2 < 100) $s1 = 1;
				else $s1 = 0
					Compare < constant; natural numbers

Uncondi-tional 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

Hardware/Software Interface

MIPS divide instructions ignore overflow, so software must determine whether the quotient is too large. In addition to overflow, division can also result in an improper calculation: division by 0. Some computers distinguish these two anomalous events. MIPS software must check the divisor to discover division by 0 as well as overflow.

Elaboration

An even faster algorithm does not immediately add the divisor back if the remainder is negative. It simply adds the dividend to the shifted remainder in the following step, since (r + d) × 2 - d = r × 2 + d × 2 - d = r × 2 + d. This nonrestoring division algorithm, which takes 1 clock cycle per step, is explored further in the exercises; the algorithm above is called restoring division. A third algorithm that doesn't save the result of the *subtract if it's negative is called a nonperforming division algorithm. It averages one-third fewer arithmetic operations.*

PARTICIPATION ACTIVITY | 3.4.8: MIPS multiply and divide instructions.

1) Goal: $s6 × $s7 (signed mulitplication)

`____ $s6, $s7`

[ ]

Check Show answer

2) Goal: $s6 / $s7 (unsigned division)

`____ $s6, $s7`

[ ]

Check Show answer

3) Goal: Place the remainder resulting from the divide operation into register $a3.

```
div $a0, $a1
____
```

[ ]

Check Show answer

PARTICIPATION ACTIVITY | 3.4.9: MIPS division.

1) The division hardware supports signed division.

○ True

○ False

2) Faster division, like faster multiplication, can be achieved by increasing the number of ALUs.

○ True

○ False

3) The divide operations, `div` and `divu`, detect overflow and division by 0.

○ True

○ False

3.5 Floating point

> Speed gets you nowhere if you're headed the wrong way.
> *American proverb.*

Going beyond signed and unsigned integers, programming languages support numbers with fractions, which are called reals in mathematics. Here are some examples of reals:

- 3.14159265 … $_{\text{ten}}$ (pi)
- 2.71828 … $_{\text{ten}}$ (*e*)
- 0.000000001_{ten} or $1.0_{\text{ten}} \times 10^{-9}$ (seconds in a nanosecond)
- $3{,}155{,}760{,}000_{\text{ten}}$ or $3.15576_{\text{ten}} \times 10^{9}$ (seconds in a typical century)

Notice that in the last case, the number didn't represent a small fraction, but it was bigger than we could represent with a 32-bit signed integer. The alternative notation for the last two numbers is called *scientific notation*, which has a single digit to the left of the decimal point. A number in scientific notation that has no leading 0s is called a *normalized* number, which is the usual way to write it. For example, $1.0_{\text{ten}} \times 10^{-9}$ is in normalized scientific notation, but $0.1_{\text{ten}} \times 10^{-8}$ and $10.0_{\text{ten}} \times 10^{-10}$ are not.

Scientific notation: A notation that renders numbers with a single digit to the left of the decimal point.

Normalized: A number in floating-point notation that has no leading 0s.

Just as we can show decimal numbers in scientific notation, we can also show binary numbers in scientific notation:

$$1.0_{\text{two}} \times 2^{-1}$$

To keep a binary number in normalized form, we need a base that we can increase or decrease by exactly the number of bits the number must be shifted to have one nonzero digit to the left of the decimal point. Only a base of 2 fulfills our need. Since the base is not 10, we also need a new name for decimal point; binary point will do fine.

Computer arithmetic that supports such numbers is called *floating point* because it represents numbers in which the binary point is not fixed, as it is for integers. The programming language C uses the name float for such numbers. Just as in scientific notation, numbers are represented as a single nonzero digit to the left of the binary point. In binary, the form is

$$1.xxxxxxxxx_{\text{two}} \times 2^{yyyy}$$

Floating point: Computer arithmetic that represents numbers in which the binary point is not fixed.

(Although the computer represents the exponent in base 2 as well as the rest of the number, to simplify the notation we show the exponent in decimal.)

A standard scientific notation for reals in normalized form offers three advantages. It simplifies exchange of data that includes floating-point numbers; it simplifies the floating-point arithmetic algorithms to know that numbers will always be in this form; and it increases the accuracy of the numbers that can be stored in a word, since the unnecessary leading 0s are replaced by real digits to the right of the binary point.

PARTICIPATION ACTIVITY | 3.5.1: Scientific notation.

Indicate if the following numbers are in scientific notation.

1) $3.56_{\text{ten}} \times 10^{-6}$

○ Yes

○ No

2) $25.11_{\text{ten}} \times 10^{-8}$

○ Yes

○ No

3) $9.8_{\text{ten}} \times 10^{12}$

○ Yes

○ No

4) $1.1101_{two} \times 2^{3}$

○ Yes

○ No

5) $100.10_{two} \times 2^{-6}$

○ Yes

○ No

6) Indicate if the following number is in *normalized* scientific notation:
$0.0514_{ten} \times 10^{-6}$

○ Yes

○ No

Floating-point representation

A designer of a floating-point representation must find a compromise between the size of the *fraction* and the size of the *exponent*, because a fixed word size means you must take a bit from one to add a bit to the other. This tradeoff is between precision and range: increasing the size of the fraction enhances the precision of the fraction, while increasing the size of the exponent increases the range of numbers that can be represented. As our design guideline from COD Chapter 2 (Instructions: Language of the Computer) reminds us, good design demands good compromise.

Fraction: The value, generally between 0 and 1, placed in the fraction field. The fraction is also called the *mantissa*.

Exponent: In the numerical representation system of floating-point arithmetic, the value that is placed in the exponent field.

Floating-point numbers are usually a multiple of the size of a word. The representation of a MIPS floating-point number is shown below, where *s* is the sign of the floating-point number (1 meaning negative), *exponent* is the value of the 8-bit exponent field (including the sign of the exponent), and *fraction* is the 23-bit number. As we recall from COD Chapter 2 (Instructions: Language of the Computer), this representation is *sign and magnitude*, since the sign is a separate bit from the rest of the number.

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

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>s</td><td colspan="8">exponent</td><td colspan="23">fraction</td></tr>
<tr><td>1 bit</td><td colspan="8">8 bits</td><td colspan="23">23 bits</td></tr>
</table>

In general, floating-point numbers are of the form

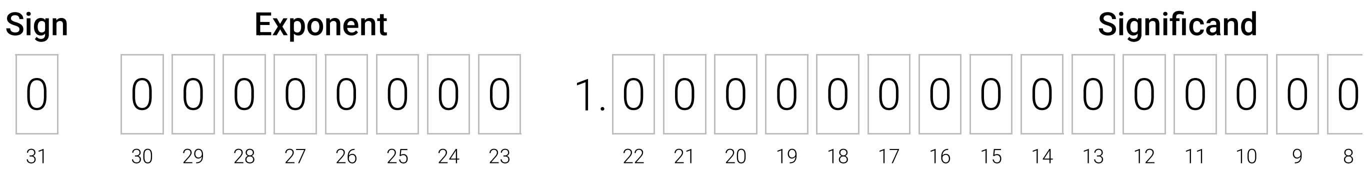
$$(-1)^{\mathbf{S}} \times \mathbf{F} \times \mathbf{2}^{\mathbf{E}}$$

F involves the value in the fraction field and E involves the value in the exponent field; the exact relationship to these fields will be spelled out soon. (We will shortly see that MIPS does something slightly more sophisticated.)

PARTICIPATION ACTIVITY 3.5.2: Representation of single precision floating-point values.

In the tool below, "significand" is another name for the mantissa.

Enter a decimal value: ______



Sign

0

31

Exponent

0	0	0	0	0	0	0	0
30	29	28	27	26	25	24	23

Significand

1.	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8

These chosen sizes of exponent and fraction give MIPS computer arithmetic an extraordinary range. Fractions almost as small as $2.0_{\text{ten}} \times 10^{-38}$ and numbers almost as large as $2.0_{\text{ten}} \times 10^{38}$ can be represented in a computer. Alas, extraordinary differs from infinite, so it is still possible for numbers to be too large. Thus, overflow interrupts can occur in floating-point arithmetic as well as in integer arithmetic. Notice that *overflow* here means that the exponent is too large to be represented in the exponent field.

> ***Overflow (floating-point)***: A situation in which a positive exponent becomes too large to fit in the exponent field.

Floating point offers a new kind of exceptional event as well. Just as programmers will want to know when they have calculated a number that is too large to be represented, they will want to know if the nonzero fraction they are calculating has become so small that it cannot be represented; either event could result in a program giving incorrect answers. To distinguish it from overflow, we call this event *underflow*. This situation occurs when the negative exponent is too large to fit in the exponent field.

Underflow (floating-point): A situation in which a negative exponent becomes too large to fit in the exponent field.

One way to reduce chances of underflow or overflow is to offer another format that has a larger exponent. In C this number is called double, and operations on doubles are called *double precision* floating-point arithmetic; *single precision* floating point is the name of the earlier format.

Double precision: A floating-point value represented in two 32-bit words.

Single precision: A floating-point value represented in a single 32-bit word.

The representation of a double precision floating-point number takes two MIPS words, as shown below, where *s* is still the sign of the number, *exponent* is the value of the 11-bit exponent field, and *fraction* is the 52-bit number in the fraction field.

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
s	exponent											fraction																			
1 bit	11 bits											20 bits																			

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
fraction																															
32 bits																															

MIPS double precision allows numbers almost as small as $2.0_{ten} \times 10^{-308}$ and almost as large as $2.0_{ten} \times 10^{308}$. Although double precision does increase the exponent range, its primary advantage is its greater precision because of the much larger fraction.

PARTICIPATION ACTIVITY | 3.5.3: Floating-point representation.

1) A calculation that leads to a number being too large to represent is called _____.

○ overflow

○ underflow

○ a fraction

2) Increasing the size of the _____ used to represent a floating-point number impacts the number's precision.

○ fraction

○ exponent

3) A _____ precision floating-point number is represented with two MIPS words.

○ single

○ double

These formats go beyond MIPS. They are part of the IEEE 754 *floating-point standard*, found in virtually every computer invented since 1980. This standard has greatly improved both the ease of porting floating-point programs and the quality of computer arithmetic.

To pack even more bits into the significand, IEEE 754 makes the leading 1-bit of normalized binary numbers implicit. Hence, the number is actually 24 bits long in single precision (implied 1 and a 23-bit fraction), and 53 bits long in double precision (1 + 52). To be precise, we use the term *significand* to represent the 24- or 53-bit number that is 1 plus the fraction, and *fraction* when we mean the 23- or 52-bit number. Since 0 has no leading 1, it is given the reserved exponent value 0 so that the hardware won't attach a leading 1 to it.

Thus 00 … 00_{two} represents 0; the representation of the rest of the numbers uses the form from before with the hidden 1 added:

$$(-1)^{\mathbf{S}} \times (\mathbf{1} + \mathbf{Fraction}) \times \mathbf{2}^{\mathbf{E}}$$

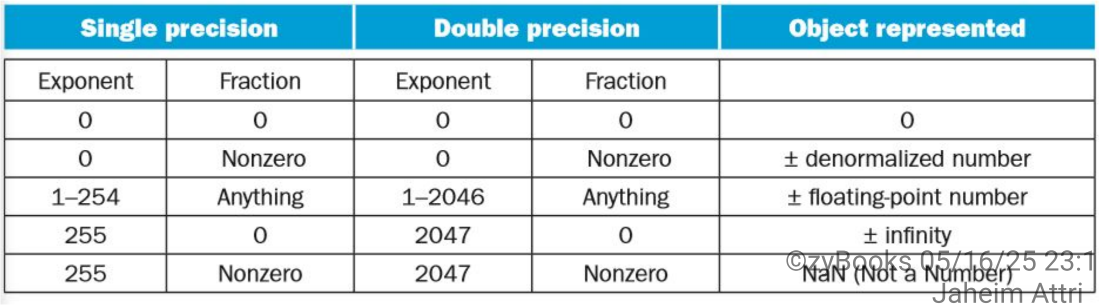
where the bits of the fraction represent a number between 0 and 1 and E specifies the value in the exponent field, to be given in detail shortly. If we number the bits of the fraction from *left to right* s1, s2, s3, …, then the value is

$$(-1)^{\mathbf{S}} \times (\mathbf{1} + (s\mathbf{1} \times \mathbf{2}^{-1}) + (s\mathbf{2} \times \mathbf{2}^{-2}) + (s\mathbf{3} \times \mathbf{2}^{-3}) + (s\mathbf{4} \times \mathbf{2}^{-4}) + \ldots) \times \mathbf{2}^{\mathbf{E}}$$

The figure below shows the encodings of IEEE 754 floating-point numbers. Other features of IEEE 754 are special symbols to represent unusual events. For example, instead of interrupting on a divide by 0, software can set the result to a bit pattern representing +∞ or −∞; the largest exponent is reserved for these special symbols. When the programmer prints the results, the program will print an infinity symbol. (For the mathematically trained, the purpose of infinity is to form topological closure of the reals.)

Figure 3.5.1: IEEE 754 encoding of floating-point numbers (COD Figure 3.13).

A separate sign bit determines the sign. Denormalized numbers are described in the Elaboration towards the end of this section.



Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

IEEE 754 even has a symbol for the result of invalid operations, such as 0/0 or subtracting infinity from infinity. This symbol is ***NaN***, for ***Not a Number***. The purpose of NaNs is to allow programmers to postpone some tests and decisions to a later time in the program when they are convenient.

The designers of IEEE 754 also wanted a floating-point representation that could be easily processed by integer comparisons, especially for sorting. This desire is why the sign is in the most significant bit, allowing a quick test of less than, greater than, or equal to 0. (It's a little more complicated than a simple integer sort, since this notation is essentially sign and magnitude rather than two's complement.)

Placing the exponent before the significand also simplifies the sorting of floating-point numbers using integer comparison instructions, since numbers with bigger exponents look larger than numbers with smaller exponents, as long as both exponents have the same sign.

Negative exponents pose a challenge to simplified sorting. If we use two's complement or any other notation in which negative exponents have a 1 in the most significant bit of the exponent field, a negative exponent will look like a big number. For example, $1.0_{two} \times 2^{-1}$ would be represented as

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
●	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(Remember that the leading 1 is implicit in the significand.) The value $1.0_{two} \times 2^{+1}$ would look like the smaller binary number

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	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The desirable notation must therefore represent the most negative exponent as $00 \dots 00_{two}$ and the most positive as $11 \dots 11_{two}$. This convention is called *biased notation*, with the bias being the number subtracted from the normal, unsigned representation to determine the real value.

IEEE 754 uses a bias of 127 for single precision, so an exponent of -1 is represented by the bit pattern of the value $-1 + 127_{ten}$, or $126_{ten} = 0111\ \ 1110_{two}$, and +1 is represented by 1 + 127, or $128_{ten} = 1000\ \ 0000_{two}$. The exponent bias for double precision is 1023. Biased exponent means that the value represented by a floating-point number is really

$$(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$$

The range of single precision numbers is then from as small as

$$\pm 1.00000000000000000000000_{two} \times 2^{-126}$$

to as large as

$$\pm 1.11111111111111111111111_{two} \times 2^{+127}.$$

Let's demonstrate.

PARTICIPATION ACTIVITY 3.5.4: Example of floating-point representation.

Show the IEEE 754 binary representation of the number -0.75$_{ten}$ in single and double precision.

$-\frac{3}{4}$ or $-\frac{3}{2^2}$ Rewrite as a fraction

$-\frac{11_{two}}{2^2}$ or -0.11_{two} Rewrite as a binary fraction

$-0.11_{two} \times 2^{0}$ Rewrite as a normalized scientific notation

$-1.1_{two} \times 2^{-1}$

Single precision binary representation:

$-1 \times 1.1_{two} \times 2^{-1}$ $(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - 127})}$

$(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(126 - 127)}$

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
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 8 bits 23 bits

Double precision binary representation:

$-1 \times 1.1_{two} \times 2^{-1}$ $(-1)^{S} \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - 1023})}$

$(-1)^{1} \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two}) \times 2^{(10}$

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
1	0	1	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit 11 bits 20 bits

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	0	0	0

32 bits

Animation content:

Static figure: The participation activity explanation states "Show the IEEE 754 binary representation of the number -0.75 base ten in single and double precision.". The next line states "negative three fourths or negative 3 divided by 2 squared" with explanation "Rewrite as a fraction". The next line states "negative 11 base two divided by 2 squared" with explanation "Rewrite as a binary fraction". The next line states "negative 0.11 base 2 or negative 0.11 base two times 2 to the power of 0 or negative 1.1 base two times 2 to the power of negative 1" with explanation "Rewrite as a normalized scientific notation".

The next section is labeled Single precision binary representation. The normalized scientific representation, negative 1 times 1.1 base two times 2 to the power of negative 1, is compared to the single-precision representation, left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 127 right parenthesis. The next line updates the single-precision representation with the values from the normalized scientific representation to produce left parenthesis negative 1 right parenthesis to the power of 1 time left parenthesis 1 plus point 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 base two right parenthesis times 2 to the power of left parenthesis 126 minus 127 right parenthesis. Below the single-precision representation are 32 bits labeled from 31 to 0. Bit 31 is labeled 1 bit wide and contains the value 1. Bits 30 to 23 are labeled 8 bits wide and contain the value 0 1 1 1 1 1 1 0. Bits 22 to 0 are labeled 23 bits wide and contain the value 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

The next section is labeled Double precision binary representation. The normalized scientific representation, negative 1 times 1.1 base two times 2 to the power of negative 1, is compared to the double-precision representation, left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 1023 right parenthesis. The next line updates the double-precision representation with the values from the normalized scientific representation to produce left parenthesis negative 1 right parenthesis to the power of 1 times left parenthesis 1 plus point 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 base two right parenthesis times 2 to the power of left parenthesis 1022 minus 1023 right parenthesis. Below the double-precision representation are 32 bits labeled from 31 to 0. Bit 31 is labeled 1 bit wide and contains the value 1. Bits 30 to 20 are labeled 11 bits wide and contain the value 0 1 1 1 1 1 1 1 1 1 0. Bits 19 to 0 are labeled 20 bits wide and contain the value 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0. Below the 32 bits are another 32 bits with all bits equal to 0.

Step 1: The number -0.75 is rewritten to binary normalized scientific notation.
The explanation states "Show the IEEE 754 binary representation of the number -0.75 base ten in single and double precision.". The next line states "negative three fourths or negative 3 divided by 2 squared" with explanation "Rewrite as a fraction". The next line states "negative 11 base two divided by 2 squared" with explanation "Rewrite as a binary fraction". The next line states

"negative 0.11 base 2 or negative 0.11 base two times 2 to the power of 0 or negative 1.1 base two times 2 to the power of negative 1" with explanation "Rewrite as a normalized scientific notation".

Step 2: The number is next converted to the single precision representation.
The next section is labeled Single precision binary representation. The normalized scientific representation appears, equal to negative 1 times 1.1 base two times 2 to the power of negative 1. The single-precision representation appears, equal to left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 127 right parenthesis. Next, in the normalized scientific representation, negative 1 is highlighted. Then, in the single-precision representation, left parenthesis negative 1 right parenthesis to the power of S is highlighted. Left parenthesis negative 1 right parenthesis to the power of 1 appears below the single-precision representation. Next, in the normalized scientific representation, 1.1 base two is highlighted. Then, in the single-precision representation, left parenthesis 1 plus Fraction right parenthesis is highlighted. And, added to the right of the expression below the single-precision representation is the multiplication operator followed by left parenthesis 1 plus point 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 base two right parenthesis. Next, in the normalized scientific representation, 2 to the power of negative 1 is highlighted. Then, in the single-precision representation, 2 to the power of left parenthesis Exponent minus 127 right parenthesis is highlighted. And, added to the right of the expression below the single-precision representation is 2 to the power of left parenthesis 126 minus 127 right parenthesis.

Step 3: The corresponding sign, fraction, and exponent bits are filled in.
32 bits labeled from 31 to 0. Bit 31 is labeled 1 bit wide, bits 30 to 23 are labeled 8 bits wide, and bits 22 to 0 are labeled 23 bits wide. The 1 in left parenthesis negative 1 right parenthesis to the 1 is highlighted and bit 31 is assigned 1. The 126 in 2 to the power of left parenthesis 126 minus 127 right parenthesis is highlighted and bits 30 to 23 are assigned 0 1 1 1 1 1 1 0. The 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 in 1 plus point 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 base two is highlighted and bits 22 to 0 are assigned 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.

Step 4: The number is next converted to the double precision representation.
The next section is labeled Double precision binary representation. The normalized scientific representation appears, equal to negative 1 times 1.1 base two times 2 to the power of negative 1. The double-precision representation appears, equal to Left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 1023 right parenthesis. Next, in the normalized scientific representation, negative 1 is highlighted. Then, in the double-precision representation, left parenthesis negative 1 right parenthesis to the power of S is highlighted. Left parenthesis negative 1 right parenthesis to the power of 1 appears below the double-precision representation. Next, in the normalized scientific representation, 1.1 base two is highlighted. Then, in the double-precision representation, left parenthesis 1 plus Fraction right parenthesis is highlighted. And,

added to the right of the expression below the double-precision representation is the multiplication operator followed by left parenthesis 1 plus point 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 base two right parenthesis. Next, in the normalized scientific representation, 2 to the power of negative 1 is highlighted. Then, in the double-precision representation, 2 to the power of left parenthesis Exponent minus 127 right parenthesis is highlighted. And, added to the right of the expression below the double-precision representation is 2 to the power of left parenthesis 1022 minus 1023 right parenthesis.

Step 5: The corresponding sign, fraction, and exponent bits are filled in.
Two rows of 32 bits labeled from 31 to 0. In the first row, bit 31 is labeled 1 bit wide, bits 30 to 23 are labeled 8 bits wide, and bits 22 to 0 are labeled 23 bits wide. The second row is labeled 32 bits wide. The 1 in left parenthesis negative 1 right parenthesis to the 1 is highlighted and bit 31 is assigned 1. The 1022 in 2 to the power of left parenthesis 1022 minus 1023 right parenthesis is highlighted and bits 30 to 23 are assigned 0 1 1 1 1 1 1 0. The 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 in 1 plus point 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 base two is highlighted and bits 22 to 0 are assigned 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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0. Bits 31 to 0 in the second row are assigned 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 0 0 0.

Animation captions:

1. The number is rewritten to binary normalized scientific notation.
2. The number is next converted to the single precision representation.
3. The corresponding sign, fraction, and exponents bits are filled in.
4. The number is next converted to the double precision representation.
5. The corresponding sign, fraction, and exponents bits are filled in.

PARTICIPATION ACTIVITY | 3.5.5: Single precision floating-point representation.

Show the IEEE 754 binary representation of the number $+0.375_{ten}$ in single precision:
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - 127})}$

How to use this tool ⌄

$(-1)^0 \times (1 + .1000\ 0000\ 0000\ 0000\ 0000\ 0000) \times 2^{(125 - 127)}$ **$1.1_{two} \times 2^{-2}$** **125**

or **or 0.011_{two}** **.1000 0000 0000 0000 0000 0000** **0**

Rewrite as a fraction

Rewrite as a binary number

Rewrite as normalized scientific notation

S = ?

Exponent = ?

Fraction = ?

IEEE 754 binary single precision representation

Reset

PARTICIPATION ACTIVITY 3.5.6: Double precision floating-point representation.

Show the IEEE 754 binary representation of the number -0.9375_{ten} in double precision: $(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - 1023})}$

How to use this tool

.1110 0000 … 0000

or 0.1111_{two}

$(-1)^1 \times (1 + .1110\ 0000 \ldots 0000) \times 2^{(1022 - 1023)}$

1

$1.111_{two} \times 2^{-1}$

or

1022

Rewrite as a fraction

Rewrite as a binary number

Rewrite as normalized scientific

notation

S = ?

Exponent = ?

Fraction = ?

IEEE 754 binary double precision representation

Reset

Now let's try going the other direction.

PARTICIPATION ACTIVITY 3.5.7: Example of converting binary to decimal floating point.

What decimal number is represented by this single precision float?

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
1	1	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	.	.	.

1 bit 8 bits 23 bits

sign 1

exponent field $10000001_{two} = 129_{ten}$

fraction field $010\ldots_{two} = 1 \times 2^{-2} = \frac{1}{4} = 0.25_{ten}$

Single precision binary representation:

$$
\begin{aligned}
(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - 127})} &= (-1)^1 \times (1 + 0.25) \times 2^{(129\ -127)} \\
&= -1 \times 1.25 \times 2^2 \\
&= -1.25 \times 4 \\
&= -5.0
\end{aligned}
$$

Animation content:

Static figure: The question "What decimal number is represented by this single precision float?" is displayed. Below the question are 32 bits labeled from 31 to 0. Bit 31 is labeled 1 bit wide and contains the value 1. Bits 30 to 23 are labeled 8 bits wide and contain the value 1 0 0 0 0 0 0 1. Bits 22 to 0 are labeled 23 bits wide and contain the value 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0. Below the 32 bits is the statement "sign 1". The next line below is the statement "exponent field 1 0 0 0 0 0 0 1 base two equals 129 base ten". The next line below is the statement "fraction field 0 1 0 dot dot dot base two equals 1 times 2 to the power of negative 2 equals 1 divided by 4 equals 0.25 base 10". The next section is labeled Single precision binary representation. Below the label is the single-precision representation, left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 127 right parenthesis, which is equated to left parenthesis negative 1 right parenthesis to the power of 1 times left parenthesis 1 plus 0.25 right parenthesis times 2 to the power of left parenthesis 129 minus 127 right parenthesis. One the next line to the right of an equals sign is negative 1 times 1.25 times 2 to the power of 2. On the next line below to the right of an equals sign is negative 1.25 times 4. On the next line below to the right of an equals sign is negative 5.0.

Step 1: The sign bit is 1, the exponent field contains 129, and the fraction field contains 0.25.
The term sign appears and the 1 in bit 31 is highlighted. Sign is assigned 1. The term exponent field appears and the 1 0 0 0 0 0 0 1 in bits 30 to 23 are highlighted. The exponent field is assigned 1 0 0 0 0 0 0 1 base two equals 129 base ten. The term fraction field appears and the 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 dot dot dot in bits 22 to 0 are highlighted. The fraction field is assigned 0 1 0 dot dot dot base two equal to 1 times 2 to the power of negative 2 equals 1 divided by 4 equal to 0.25 base ten.
Step 2: The sign, exponent, and fraction are plugged into the basic equation.
The next section is labeled Single precision binary representation. Below the label is the single-precision representation, left parenthesis negative 1 right parenthesis to the power of S times left parenthesis 1 plus Fraction right parenthesis times 2 to the power of left parenthesis Exponent minus 127 right parenthesis equals. In the single-precision representation, left parenthesis negative 1 right parenthesis to the power of S is highlighted. Then, on the right side of the equals sign, left parenthesis negative 1 right parenthesis times appears, followed by the multiplication operator. Then, in the single-precision representation, left parenthesis 1 plus Fraction right parenthesis is highlighted. Then, on the right side of the equals sign, left parenthesis 1 plus 0.25 right parentheses times appears to the right of the multiplication operator. Then, in the single-precision representation, 2 to the power of left parenthesis Exponent minus 127 right parenthesis is highlighted. Then, on the right side of the equals sign, 2 to the power of left parenthesis 129 minus 127 right parenthesis appears to the right of the multiplication operator.

Step 3: The binary single precision floating pointing value represents -5.0 in decimal.

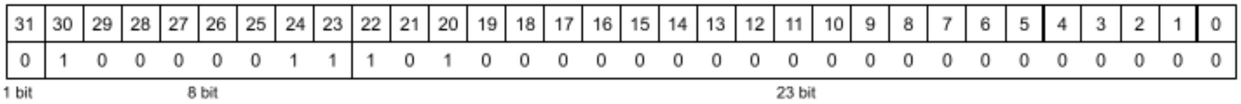
On the next line below to the right of an equals sign appears -1 times 1.25 times 2 to the power of 2. On the next line below to the right of an equals sign appears negative 1.25 times 4. On the next line below to the right of an equals sign appears negative 5.0.

Animation captions:

1. The sign bit is 1, the exponent field contains 129, and the fraction field contains 0.25.
2. The sign, exponent, and fraction are plugged into the basic equation.
3. The binary single precision floating pointing value represents −5.0 in decimal.

PARTICIPATION ACTIVITY 3.5.8: Converting a single precision binary floating-point representation to decimal.

Convert the single precision binary floating-point representation to decimal.



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	1	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

1 bit
8 bit
23 bit

1) The sign bit is ____.

Check Show answer

2) Exponent field is ____ $_{\text{ten}}$.

Check Show answer

3) The fraction field is ____ $_{\text{ten}}$.

Check Show answer

4) $(-1)^{0} \times (\textbf{?} + 0.625) \times 2^{(131 - 127)}$

Check Show answer

5) What decimal number is

represented by the above single precision binary floating-point number?

$(-1)^0 \times (1 + 0.625) \times 2^{(131 - 127)}$
$= 1 \times (1 + 0.625) \times 2^4$
$=$ **?**

Check Show answer

In the next few subsections, we will give the algorithms for floating-point addition and multiplication. At their core, they use the corresponding integer operations on the significands, but extra bookkeeping is necessary to handle the exponents and normalize the result. We first give an intuitive derivation of the algorithms in decimal and then give a more detailed, binary version in the animations.

Following IEEE guidelines, the IEEE 754 committee was reformed 20 years after the standard to see what changes, if any, should be made. The revised standard IEEE 754-2008 includes nearly all the IEEE 754-1985 and adds a 16-bit format ("half precision") and a 128-bit format ("quadruple precision"). Half precision has a 1-bit sign, 5-bit exponent (with a bias of 15), and a 10-bit fraction. Quadruple precision has a 1-bit sign, 15-bit exponent (with a bias of 262143), and a 112-bit fraction. No hardware has yet been built that supports quadruple precision, but it will surely come. The revised standard also adds decimal floating point arithmetic, which IBM mainframes have implemented.

Elaboration

In an attempt to increase range without removing bits from the significand, some computers before the IEEE 754 standard used a base other than 2. For example, the IBM 360 and 370 mainframe computers use base 16. Since changing the IBM exponent by one means shifting the significand by 4 bits, "normalized" base 16 numbers can have up to 3 leading bits of 0s! Hence, hexadecimal digits mean that up to 3 bits must be dropped from the significand, which leads to surprising problems in the accuracy of floating-point arithmetic. IBM mainframes now support IEEE 754 as well as the hex format.

Floating-point addition

Let's add numbers in scientific notation by hand to illustrate the problems in floating-point addition: $9.999_{\text{ten}} \times 10^{1} + 1.610_{\text{ten}} \times 10^{-1}$. Assume that we can store only four decimal digits of the significand and two decimal digits of the exponent.

Step 1

To be able to add these numbers properly, we must align the decimal point of the number that has the smaller exponent. Hence, we need a form of the smaller number, $1.610_{\text{ten}} \times 10^{-1}$, that matches the larger exponent. We obtain this by observing that there are multiple representations of an unnormalized floating-point number in scientific notation:

$$\mathbf{1.610_{ten} \times 10^{-1} = 0.1610_{ten} \times 10^{0} = 0.01610_{ten} \times 10^{1}}$$

The number on the right is the version we desire, since its exponent matches the exponent of the larger number, $9.999_{\text{ten}} \times 10^{1}$. Thus, the first step shifts the significand of the smaller number to the right until its corrected exponent matches that of the larger number. But we can represent only four decimal digits so, after shifting, the number is really

$$\mathbf{0.016 \times 10^{1}}$$

Step 2

Next comes the addition of the significands:

$$\begin{array}{r} \mathbf{9.999_{ten}} \\ + \ \mathbf{0.016_{ten}} \\ \hline \mathbf{10.015_{ten}} \end{array}$$

The sum is $10.015_{\text{ten}} \times 10^{1}$.

Step 3

This sum is not in normalized scientific notation, so we need to adjust it:

$$\mathbf{10.015_{ten} \times 10^{1} = 1.0015_{ten} \times 10^{2}}$$

After the addition we may have to shift the sum to put it into normalized form, adjusting the exponent appropriately. This example shows shifting to the right, but if one number were positive and the other were negative, it would be possible for the sum to have many leading 0s, requiring left shifts. Whenever the exponent is increased or decreased, we must check for overflow or underflow—that is, we must make sure that the exponent still fits in its field.

Step 4

Since we assumed that the significand can be only four digits long (excluding the sign), we must round the number. In our grammar school algorithm, the rules truncate the number if the digit to the right of the desired point is between 0 and 4 and add 1 to the digit if the number to the right is between 5 and 9. The number

$$1.0015_{\mathbf{ten}} \times 10^2$$

is rounded to four digits in the significand to

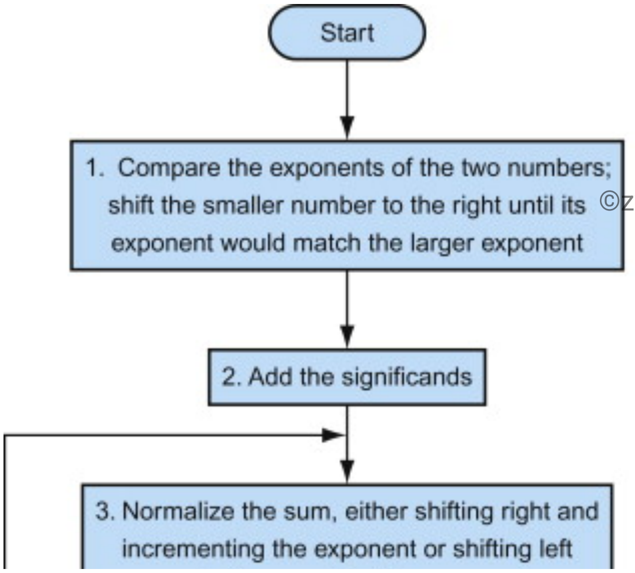
$$1.002_{\mathbf{ten}} \times 10^2$$

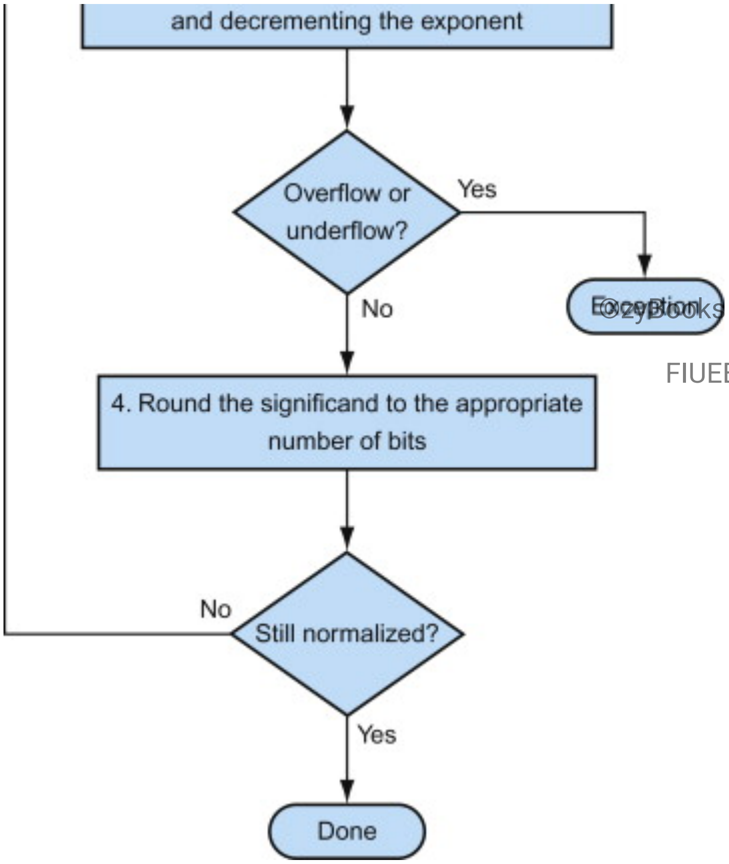
since the fourth digit to the right of the decimal point was between 5 and 9. Notice that if we have bad luck on rounding, such as adding 1 to a string of 9s, the sum may no longer be normalized and we would need to perform step 3 again.

The figure below shows the algorithm for binary floating-point addition that follows this decimal example. Steps 1 and 2 are similar to the example just discussed: adjust the significand of the number with the smaller exponent and then add the two significands. Step 3 normalizes the results, forcing a check for overflow or underflow. The test for overflow and underflow in step 3 depends on the precision of the operands. Recall that the pattern of all 0 bits in the exponent is reserved and used for the floating-point representation of zero. Moreover, the pattern of all 1 bits in the exponent is reserved for indicating values and situations outside the scope of normal floating-point numbers. For the example below, remember that for single precision, the maximum exponent is 127, and the minimum exponent is -126.

Figure 3.5.2: Floating-point addition (COD Figure 3.14).

The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.


Start
1. Compare the exponents of the two numbers; shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left


and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
Done

PARTICIPATION ACTIVITY 3.5.9: Example of binary floating-point addition.

Add the numbers 0.5_{ten} and -0.4375_{ten} in binary using the floating-point addition algorithm.

$$0.5_{ten} = \frac{1}{2}_{ten} = \frac{1}{2^1}_{ten} = 0.1_{two} = 0.1_{two} \times 2^0 = 1.000_{two} \times 2^{-1}$$

$$-0.4375_{ten} = -\frac{7}{16}_{ten} = -\frac{7}{2^4}_{ten} = -0.0111_{two} = -0.0111_{two} \times 2^0 = -1.110_{two} \times 2^{-2}$$

Follow the algorithm:

$1.000_{two} \times 2^{-1}$ $-0.111_{two} \times 2^{-1}$ 1. Rewrite to match exponents

$$\begin{array}{rl} & 1.000_{two} \times 2^{-1} \\ + & -0.111_{two} \times 2^{-1} \\ & 0.001_{two} \times 2^{-1} \end{array}$$

2. Add significands

$1.000_{two} \times 2^{-4}$

3. Normalize sum
   No overflow/underflow

4. Round sum
   Already 4 bits

Animation content:

Static figure: The participation activity instruction states "Add the numbers 0.5 base ten and negative 0.4375 base 10 in binary using the floating-point addition algorithm.". The next line below has the equation 0.5 base ten equal to 1 divided by 2 base ten, which equals 1 divided by 2 to the power of 1 base ten, which equals 0.1 base two, which equals 0.1 base two times 2 to the power of 0, which equals 1 point 0 0 0 base two times 2 to the power of negative 1. The next line below has the equation negative 0.4375 base ten equal to negative 7 divided by 16 base ten, which equals negative 7 over 2 to the power of 4 base ten, which equals negative 0 point 0 1 1 1 base two, which equals negative 0 point 0 1 1 1 base two times 2 to the power of 0, which equals negative 1 point 1 1 0 base two times 2 to the power of negative 2.

The next section states the instruction "Follow the algorithm". The next line below shows 1 point 0 0 0 base two times 2 to the power of negative 1 and negative 0 point 1 1 1 base two times 2 to the power of negative 1. To the right is the instruction, "1 Rewrite to match exponents". The next line below shows the addition, 1 point 0 0 0 base two times 2 to the power of negative 1 plus negative 0 point 1 1 1 base two times 2 to the power of negative 1. The next line below shows the result, equal to 0 point 0 0 1 base two times 2 to the power of negative 1. To the right is the instruction, "2 Add significands". The next line below shows 1 point 0 0 0 base two times 2 to the power of negative 4. To the right is the instruction, "3 Normalize sum" followed by the statement "No overflow forward slash underflow". The next line below contains the instruction, "4 Round sum" followed by the statement "Already 4 bits". The next line contains the statement "Check: 1 point 0 0 0 base two times 2 to the power of negative 4 equals 0 point 0 0 0 1 0 0 0 base two equals 0 dot 0 0 0 1 base two equals 1 over 2 to the power of 4 base ten equals 1 over 16 base ten equals 0.0625 base ten".

Step 1: Assuming 4 bits of precision, the numbers are rewritten in normalized scientific notation. The instruction "Add the numbers 0.5 base ten and negative 0.4375 base 10 in binary using the floating-point addition algorithm" appears. Then, on the line below appears 0.5 base ten equals 1 over 2 base ten equals 1 over 2 to the power of 1 base ten equals 0.1 base two equals 0.1 base two times 2 to the power of 0 equals 1 point 0 0 0 base two times 2 to the power of negative 1. Then, on the line below appears negative 0.4375 base ten equals negative 7 over 16 base ten equals negative 7 over 2 to the power of 4 base ten equals negative 0 point 0 1 1 1 base two equals negative 0 point 0 1 1 1 base two times 2 to the power of 0 equals negative 1 point 1 1 0 base two times 2 to the power of negative 2.

Step 2: Step 1. The significand of the number with the lesser exponent is shifted right until its exponent matches the larger number.
The instruction "Follow the algorithm" appears. On the line below appears the instruction, "1. Rewrite to match exponents". Then, to the left of the instruction appears 1 point 0 0 0 base two times 2 to the power of negative 1 and negative 1 point 1 1 0 base two times 2 to the power of negative 2. Then, negative 1 point 1 1 0 base two times 2 to the power of negative 2 gets rewritten to negative 0 point 1 1 1 base two times two to the power of negative 1.

Step 3: Step 2. Add the significands.
The instruction "2 Add significands" appears. Then, to the left of the instruction appears the addition, 1 point 0 0 0 base two times 2 to the power of negative 1 plus negative 0 point 1 1 1 base two times 2 to the power of negative 1. Then the result appears, equal to 0 point 0 0 1 base two times 2 to the power of negative 1.

Step 4: Step 3. Normalize the sum, checking for overflow or underflow.
The instruction "3 Normalize sum" appears. The result of the addition, 0 point 0 0 1 base two times 2 to the power of negative 1, changes to 0 point 0 1 0 base two times 2 to the power of negative 2, then changes to 0 point 1 0 0 base two times 2 to the power of negative 3, and finally changes to 1 point 0 0 0 base two times 2 to the power of negative 4.

Step 5: Since 127 ≥ -4 ≥ -126, there is no overflow or underflow. The biased exponent would be -4 + 127, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.
The negative 4 exponent in 1 point 0 0 0 base two times 2 to the power of negative 4 is highlighted and "No overflow forward slash underflow" appears.

Step 6: Step 4. Round the sum.
The instruction "4. Round sum" appears. 1 point 0 0 0 in 1 point 0 0 0 base two times 2 to the power of negative 4 is highlighted and "Already 4 bits" appears.

Step 7: Check the result.
The statement "Check:" appears, followed by the expression, 1 point 0 0 0 base two times 2 to the power of negative 4 equals 0 point 0 0 0 1 0 0 0 base two equals 0 dot 0 0 0 1 base two equals 1 over 2 to the power of 4 base ten equals 1 over 16 base ten equals 0.0625 base ten.

Animation captions:

1. Assuming 4 bits of precision, the numbers are rewritten in normalized scientific notation.
2. Step 1. The significand of the number with the lesser exponent is shifted right until its exponent matches the larger number.
3. Step 2. Add the significands.
4. Step 3. Normalize the sum, checking for overflow or underflow.

5. Since $127 \geq -4 \geq -126$, there is no overflow or underflow. The biased exponent would be $-4 + 127$, or 123, which is between 1 and 254, the smallest and largest unreserved biased exponents.
6. Step 4. Round the sum. The sum already fits exactly in 4 bits, so there is no change to the bits due to rounding.
7. Check the result. This sum is what we would expect from adding 0.5 to -0.4375.

PARTICIPATION ACTIVITY | 3.5.10: Binary floating-point addition.

Add the following numbers using the floating-point addition algorithm. Assume 4 bits of precision.

1) 1.010 × 2^{-3} + 0.011 × 2^{-3} = ?

- ○ 1.101
- ○ 1.101 x 2^{-6}
- ○ 1.101 x 2^{-3}

2) 1.001 × 2^{-4} + 1.000 × 2^{-6} = ?

- ○ 10.001 × 2^{-4}
- ○ 1.011 × 2^{-4}

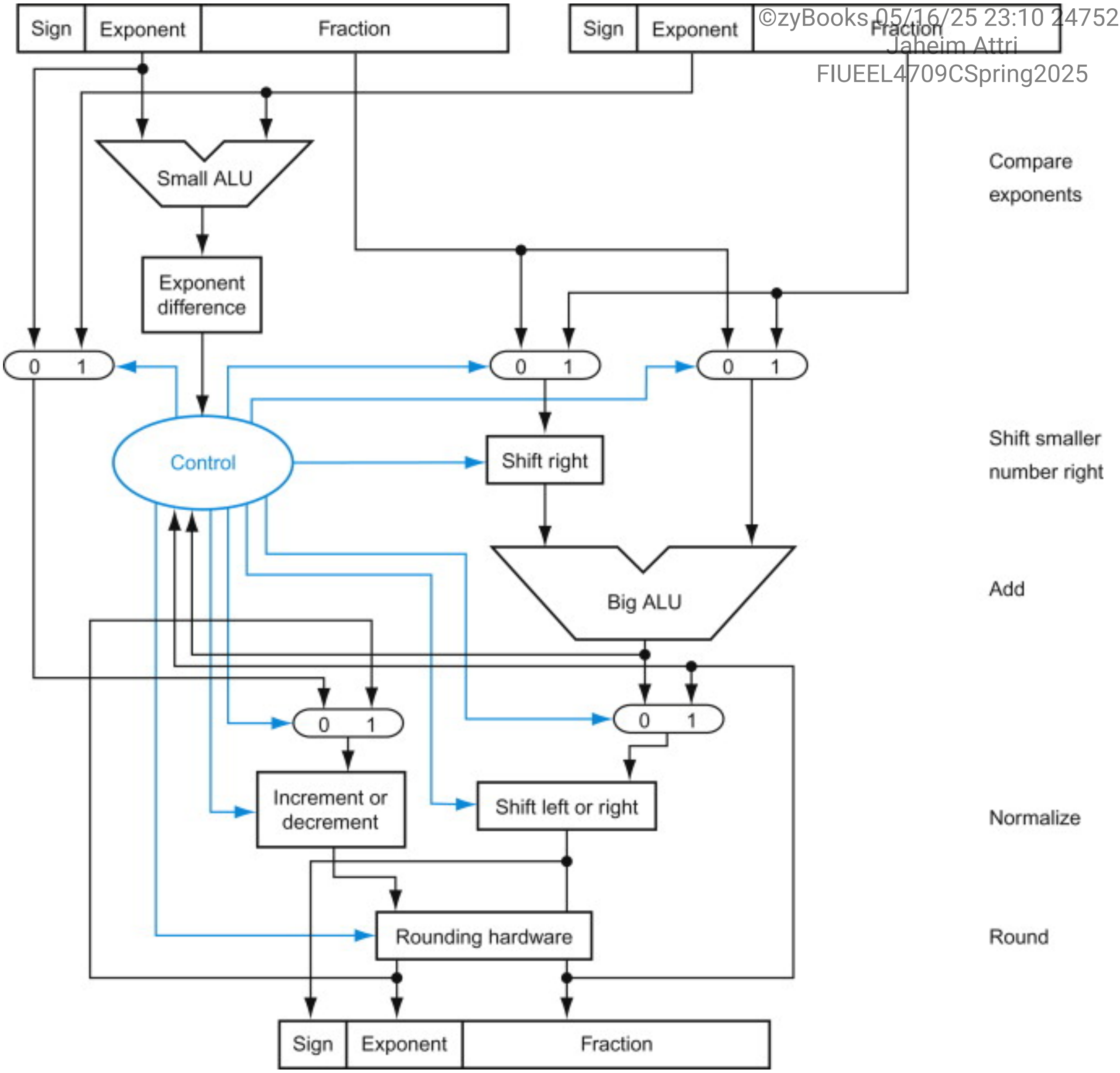
3) 1.000 × 2^{3} + 0.011 × 2^{5} = ?

- ○ 1.010 × 2^{4}
- ○ 0.101 × 2^{5}
- ○ 10.001 × 2^{5}

Many computers dedicate hardware to run floating-point operations as fast as possible. The figure below sketches the basic organization of hardware for floating-point addition.

Figure 3.5.3: Block diagram of an arithmetic unit dedicated to floating-point addition (COD Figure 3.15).

The steps of the figure above (Floating-point addition) correspond to each block, from top to bottom. First, the exponent of one operand is subtracted from the other using the small ALU to determine which is larger and by how much. This difference controls the three multiplexors; from left to right, they select the larger exponent, the significand of the smaller number, and the significand of the larger number. The smaller significand is shifted right, and then the significands are added together using the big ALU. The normalization step then shifts the sum left or right and increments or decrements the exponent. Rounding then creates the final result, which may require normalizing again to produce the actual final result.



PARTICIPATION ACTIVITY 3.5.11: Floating-point addition hardware.

1) The small ALU is used to add the significands.

○ True

○ False

2) The "Increment or decrement" and "Shift left or right" hardware normalize the sum.

○ True

○ False

3) Rounding may require the result to be normalized again.

○ True

○ False

Floating-point multiplication

Now that we have explained floating-point addition, let's try floating-point multiplication. We start by multiplying decimal numbers in scientific notation by hand: $1.110_{ten} \times 10^{10} \times 9.200_{ten} \times 10^{-5}$. Assume that we can store only four digits of the significand and two digits of the exponent.

Step 1

Unlike addition, we calculate the exponent of the product by simply adding the exponents of the operands together:

$$\textbf{New exponent} = 10 + (-5) = 5$$

Let's do this with the biased exponents as well to make sure we obtain the same result: 10 + 127 = 137, and -5 + 127 = 122, so

$$\textbf{New exponent} = 137 + 122 = 259$$

This result is too large for the 8-bit exponent field, so something is amiss! The problem is with the bias because we are adding the biases as well as the exponents:

$$\textbf{New exponent} = (10 + 127) + (-5 + 127) = (5 + 2 \times 127) = 259$$

Accordingly, to get the correct biased sum when we add biased numbers, we must subtract the bias from the sum:

$$\textbf{New exponent} = 137 + 122 - 127 = 259 - 127 = 132 = (5 + 127)$$

and 5 is indeed the exponent we calculated initially.

Step 2

$$\begin{array}{lr} & 1.110_{\text{ten}} \\ \times & 9.200_{\text{ten}} \\ \hline & 0000\phantom{_{\text{ten}}} \\ & 0000\phantom{0_{\text{ten}}} \\ & 2220\phantom{00_{\text{ten}}} \\ & 9990\phantom{000_{\text{ten}}} \\ \hline & 10212000_{\text{ten}} \end{array}$$

There are three digits to the right of the decimal point for each operand, so the decimal point is placed six digits from the right in the product significand:

$$10.212000_{\text{ten}}$$

Assuming that we can keep only three digits to the right of the decimal point, the product is 10.212 × 10^5.

Step 3

This product is unnormalized, so we need to normalize it:

$$10.212_{\text{ten}} \times 10^5 = 1.0212_{\text{ten}} \times 10^6$$

Thus, after the multiplication, the product can be shifted right one digit to put it in normalized form, adding 1 to the exponent. At this point, we can check for overflow and underflow. Underflow may occur if both operands are small—that is, if both have large negative exponents.

Step 4

We assumed that the significand is only four digits long (excluding the sign), so we must round the number. The number

$$1.0212_{\text{ten}} \times 10^6$$

is rounded to four digits in the significand to

$$1.021_{\text{ten}} \times 10^6$$

Step 5

The sign of the product depends on the signs of the original operands. If they are both the same, the sign is positive; otherwise, it's negative. Hence, the product is

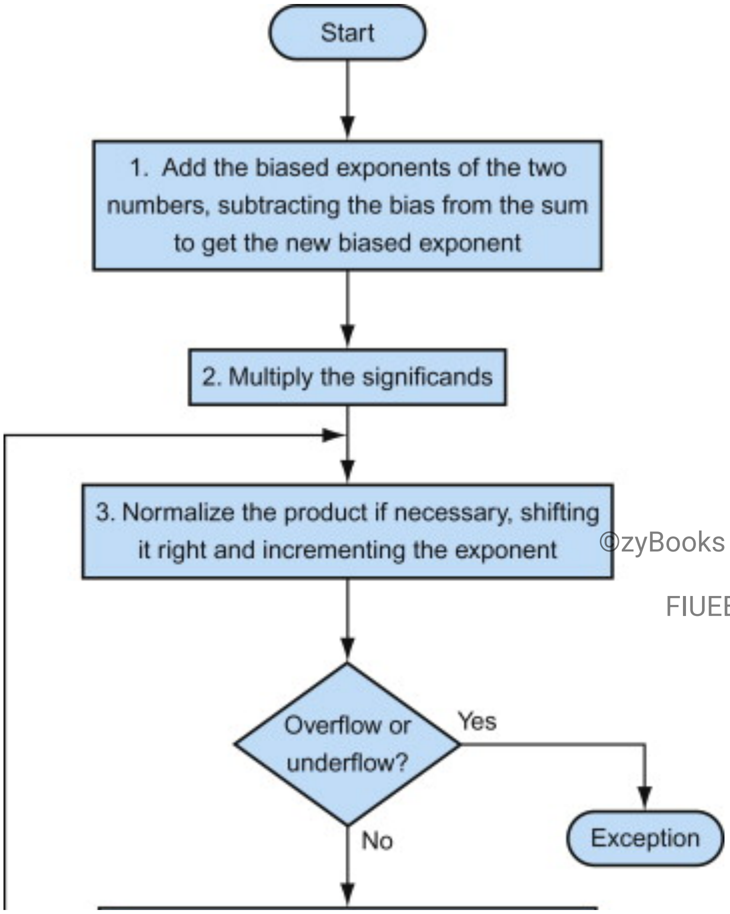
$$+1.021_{\text{ten}} \times 10^6$$

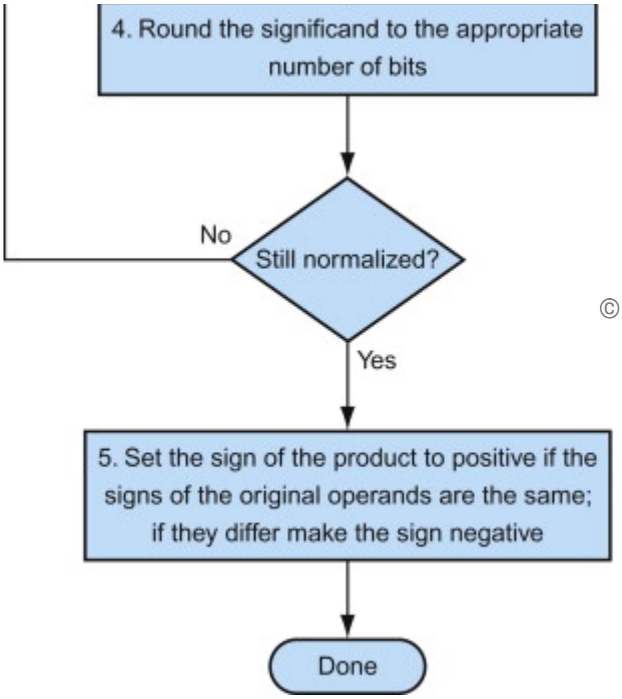
The sign of the sum in the addition algorithm was determined by addition of the significands, but in multiplication, the sign of the product is determined by the signs of the operands.

Once again, as the figure below shows, multiplication of binary floating-point numbers is quite similar to the steps we have just completed. We start with calculating the new exponent of the product by adding the biased exponents, being sure to subtract one bias to get the proper result. Next is multiplication of significands, followed by an optional normalization step. The size of the exponent is checked for overflow or underflow, and then the product is rounded. If rounding leads to further normalization, we once again check for exponent size. Finally, set the sign bit to 1 if the signs of the operands were different (negative product) or to 0 if they were the same (positive product).

Figure 3.5.4: Floating-point multiplication (COD Figure 3.16).

The normal path is to execute steps 3 and 4 once, but if rounding causes the sum to be unnormalized, we must repeat step 3.


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No


4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

PARTICIPATION ACTIVITY 3.5.12: Example of binary floating-point multiplication.

Multiply the numbers 0.5_{ten} and -0.4375_{ten} using the floating-point multiplication algorithm.

$$0.5_{ten} = 1.000_{two} \text{ x } 2^{-1}$$

$$-0.4375_{ten} = -1.110_{two} \text{ x } 2^{-2}$$

Follow the algorithm:

1. Add the exponents

New exponent (no bias) = $-1 + (-2)$
= -3

New exponent (biased) = $(-1 + 127) + (-2 + 127) - 127$
= $-3 + 127$
= 124

2. Multiply significands

```
           1.000 two
x          1.110 two
            0000
           1000
          1000
+        1000
         1110000 two
```

product = $1.110_{two} \text{ x } 2^{-3}$

3. Normalize sum
Normalized, no overflow/underflow

$1.110_{two} \text{ x } 2^{-3}$

4. Round sum
Already 4 bits

$1.110_{two} \text{ x } 2^{-3}$

5. Set sign
Signs differ, set to negative

$-\ 1.110_{two} \text{ x } 2^{-3}$

Check:

$$-1.110_{two} \text{ x } 2^{-3} = -0.001110_{two} = -0.00111_{two} = \frac{-7}{2^5} = \frac{-7}{32} = -0.21875_{ten}$$

Animation content:

Static figure: The participation activity instruction states "Multiply the numbers 0.5 base ten and -0.4375 base ten, using the floating-point multiplication algorithm". The next line below shows 0.5 base ten equal to 1 point 0 0 0 base two times 2 to the power of negative 1. The next line below shows negative 0.4375 equal to negative 1 point 1 1 0 base two times 2 to the power of negative 2.

The next section states the instruction "Follow the algorithm". The line below contains the instruction, "Step 1. Add the exponents". The line below contains the explanation, "New exponent no bias equals negative 1 plus negative 2 equals negative". The line below contains the explanation, "New exponent biased equals left parenthesis negative 1 plus 127 right parenthesis plus left parenthesis negative 2 plus 127 right parenthesis minus 127, equal to negative 3 plus 127, equal to 124. To the right is the instruction, "Step 2. Multiply significands". Shown below the instruction is the multiplication base two. The multiplicand equals 1 point 0 0 0 base two, and the multiplier equals 1 point 1 1 0. The line below shows 0 0 0 0, the next line below shows the multiplicand 1 0 0 0 with each digit shifted one to the left, the next line below shows multiplicand 1 0 0 0 with each digit shifted one to the left, and the next line below shows the multiplicand 1 0 0 0 with each digit shifted one to the left. The next line shows the results equal to 1 1 1 0 0 0 0 base two. To the right is the statement, "Product equals 1 point 1 1 0 base two times 2 to the power of negative 3".

The next section states the instruction, "Step 3. Normalize sum", and below the instruction is the statement, "Normalized, no overflow forward slash underflow". On the line below is the number, 1 point 1 1 0 base two times 2 to the power of negative 3. To the right is the instruction, "Step 4. Round sum", and below the instruction is the statement, "Already 4 bits". On the line below is the number, 1 point 1 1 0 base two times 2 to the power of negative 3. Further to the right is the instruction, "Step 5. Set sign", and below the instruction is the statement, "Signs differ, set to negative". On the line below is the number, negative 1 point 1 1 0 base two times 2 to the power of negative 3.

The next section states "Check: negative 1 point 1 1 0 base two times 2 to the power of negative 3 equals negative 0 point 0 0 1 1 1 0 base two equals negative 0 point 0 0 1 1 1 base two equals negative 7 over 2 to the power of 5 base ten equals negative 7 over 32 equals negative 0.21875.

Step 1: Assuming 4 bits of precision, the numbers are rewritten in normalized scientific notation. The instruction "Multiply the numbers 0.5 base ten and -0.4375 base ten, using the floating-point multiplication algorithm" appears. Below the instruction appears the expression, 0.5 base ten equals 1 point 0 0 0 base two times 2 to the power of negative 1. On the line below appears the expression, negative 0.4375 equals negative 1 point 1 1 0 base two times 2 to the power of negative 2.

Step 2: The exponents are added without bias and with the biased representation.
The statement "Follow the algorithm" appears. Then the instruction "1 Add the exponents" appears. On the line below appears the statement "New exponent no bias equals negative 1 plus negative 2 equals negative 3. On the line below appears the statement, "New exponent biased equals left parenthesis negative 1 plus 127 right parenthesis plus left parenthesis negative 2 plus 127 right parenthesis minus 127 equals negative 3 plus 127 equals 124

Step 3: The significands are multiplied and then rewritten to use 4 bits.
The instruction "2 Multiply significands" appears. Below the instruction appears the equation 1 point 0 0 0 base two times 1 point 1 1 0 base two. All bits of 1 point 0 0 0 are highlighted. The 0 in 1 point 1 1 0 base two is highlighted. 0 0 0 0 appears on the line below. The third 1 in 1 point 1 1 0 base two is highlighted and 1 0 0 0 appears on the line below, shifted one to the left. The second 1 in 1 point 1 1 0 base two is highlighted and 1 0 0 0 appears on the line below, shifted one to the left. The first 1 in 1 point 1 1 0 base two is highlighted and 1 0 0 0 appears on the line below, shifted one to the left. The sum 1 1 1 0 0 0 base two appears. Product equals appears. 1 1 1 0 0 0 0 duplicates and moves to the right of the equals sign and becomes 1 point 1 1 0 0 0 0. The last 3 zeros disappear and the 1 point 1 1 0 is multiplied by 2 to the power of negative 3.

Step 4: The product is normalized, and the exponent checked for overflow or underflow.
The instruction, "3 Normalize sum" appears. Below the instruction appears the number 1 point 1 1 0 base two times 2 to the power of negative 3. 1 point 1 1 0 is highlighted and the term Normalized appears.

Step 5: There is no overflow or underflow of the new exponent because 127 is greater than or equal to negative 3 greater than or equal to negative 126. Using the biased representation, 254 is greater than or equal to 124 is greater than or equal to 1, so again the exponent fits in the field.
The negative 3 in 1 point 1 1 0 base two times 2 to the power of negative 3 is highlighted and the statement "no overflow forward slash underflow" appears.

Step 6: Rounding the product makes no change, as the product already contains four bits.
The instruction "4 Round sum" appears. Below the instruction appears the number 1 point 1 1 0 base two times 2 to the power of negative 3. 1 point 1 1 0 is highlighted and the term Already 4 bits appears.

Step 7: The sign of the product is made negative because the signs of the original operands differ.
The instruction "5 Set sign" appears. Below the instruction appears the number 1 point 1 1 0 base two times 2 to the power of negative 3. Then the statement, "Signs differ, set to negative"

appears. A negative sign appears in front of the number to become negative 1 point 1 1 0 base two times 2 to the power of negative 3.

Step 8: The result is converted to decimal to verify. The product of $\mathbf{0.5}$ and $\mathbf{-0.4375}$ is indeed $\mathbf{-0.21875}$.
The statement "Check:" appears. On the line below appears the expression, negative 1 point 1 1 0 base two times 2 to the power of negative 3 equals negative 0 point 0 0 1 1 1 0 base two equals negative 0 point 0 0 1 1 1 base two equals negative 7 over 2 to the power of 5 base ten equals negative 7 over 32 equals negative 0.21875.

Animation captions:

1. Assuming 4 bits of precision, the numbers are rewritten in normalized scientific notation.
2. Step 1. The exponents are added without bias and with the biased representation.
3. Step 2. The significands are multiplied and then rewritten to use 4 bits.
4. Step 3. The product is normalized, and the exponent checked for overflow or underflow.
5. There is no overflow or underflow of the new exponent because $\mathbf{127 \geq -3 \geq -126}$. Using the biased representation, $\mathbf{254 \geq 124 \geq 1}$, so again the exponent fits in the field.
6. Step 4. Rounding the product makes no change, as the product already contains four bits.
7. Step 5. The sign of the product is made negative because the signs of the original operands differ.
8. The result is converted to decimal to verify. The product of 0.5 and $\mathbf{-0.4375}$ is indeed $\mathbf{-0.21875}$.

PARTICIPATION ACTIVITY 3.5.13: Multiplication of binary floating-point numbers.

Multiply -14$_{ten}$ and -0.25$_{ten}$, or -1.110 × 2^3 × -1.000 × 2^{-2}. Assume 4 bits of precision.

How to use this tool ⌄

1.110000 **3.5$_{ten}$** **2^1** **1.110000$_{two}$ × 2^1** **3 + (-2) = 1** **1.110$_{two}$ × 2^1** **+**

Adding the non-biased exponents of the operands

Multiply the significands:
1.110 × 1.000 = ?

Product = 1.110000 × ?

Normalize the product

Round the product

Set the sign of the product: ?

$1.110_{two} \times 2^1$

$-14_{ten} \times -0.25_{ten}$ = ?

Reset

Floating-point instructions in MIPS

MIPS supports the IEEE 754 single precision and double precision formats with these instructions:

- Floating-point *addition, single* `(add.s)` and *addition, double* `(add.d)`
- Floating-point *subtraction, single* `(sub.s)` and *subtraction, double* `(sub.d)`
- Floating-point *multiplication, single* `(mul.s)` and *multiplication, double* `(mul.d)`
- Floating-point *division, single* `(div.s)` and *division, double* `(div.d)`
- Floating-point *comparison, single* `(c.x.s)` and *comparison, double* `(c.x.d)`, where x may be *equal* `(eq)`, *not equal* `(neq)`, *less than* `(lt)`, *less than or equal* `(le)`, *greater than* `(gt)`, or *greater than or equal* `(ge)`
- Floating-point *branch, true* `(bc1t)` and *branch, false* `(bc1f)`

Floating-point comparison sets a bit to true or false, depending on the comparison condition, and a floating-point branch then decides whether or not to branch, depending on the condition.

The MIPS designers decided to add separate floating-point registers—called `$f0, $f1, $f2, ...`—used either for single precision or double precision. Hence, they included separate loads and stores for floating-point registers: `lwc1` and `swc1`. The base registers for floating-point data transfers which are used for addresses remain integer registers. The MIPS code to load two single precision numbers from memory, add them, and then store the sum might look like this:

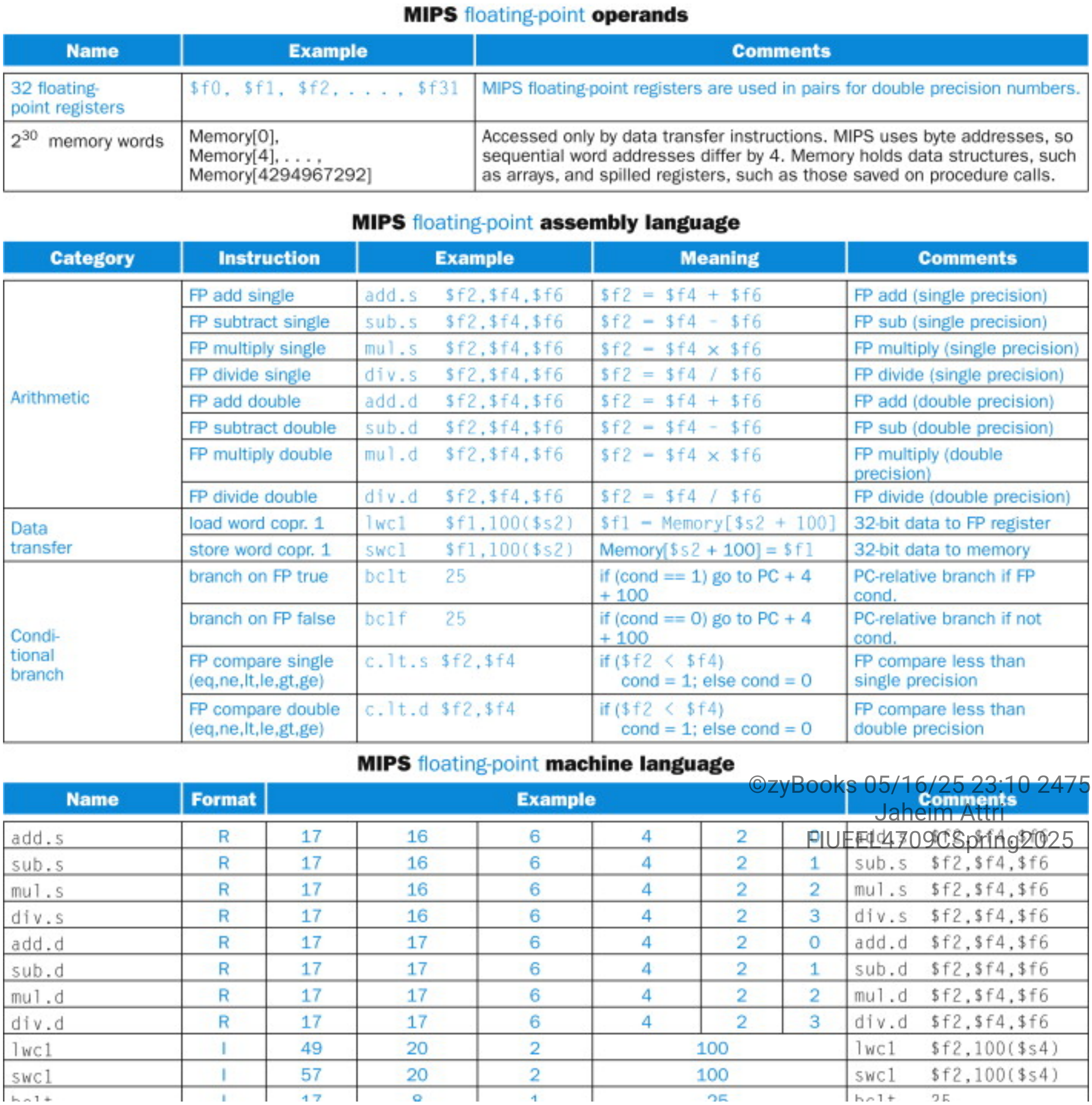
```
lwc1   $f4, c($sp)      # Load 32-bit F.P. number into F4
lwc1   $f6, a($sp)      # Load 32-bit F.P. number into F6
add.s  $f2, $f4, $f6    # F2 = F4 + F6 single precision
swc1   $f2, b($sp)      # Store 32-bit F.P. number from F2
```

A double precision register is really an even-odd pair of single precision registers, using the even register number as its name. Thus, the pair of single precision registers `$f2` and `$f3` also form the double precision register named `$f2`.

The figure below summarizes the floating-point portion of the MIPS architecture revealed in this chapter, with the additions to support floating point shown in color.

Figure 3.5.5: MIPS floating-point architecture revealed thus far (COD Figure 3.17).

See COD Appendix A (Assemblers, Linkers, and the SPIM Simulator), COD Section A.10 (MIPS R2000 assembly language), for more detail.



MIPS floating-point operands

Name	Example	Comments
32 floating-point registers	`$f0, $f1, $f2, . . . , $f31`	MIPS floating-point registers are used in pairs for double precision numbers.
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 floating-point assembly language

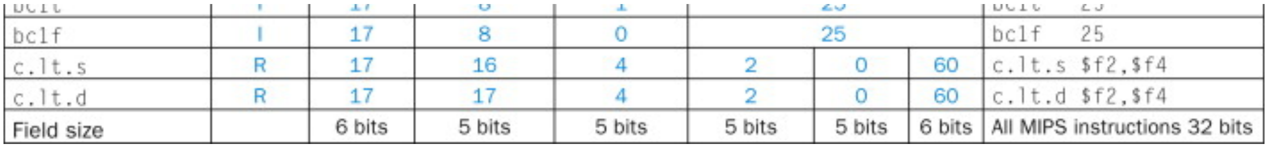
Category	Instruction	Example	Meaning	Comments
Arithmetic	FP add single	`add.s $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (single precision)
	FP subtract single	`sub.s $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (single precision)
	FP multiply single	`mul.s $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (single precision)
	FP divide single	`div.s $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (single precision)
	FP add double	`add.d $f2,$f4,$f6`	`$f2 = $f4 + $f6`	FP add (double precision)
	FP subtract double	`sub.d $f2,$f4,$f6`	`$f2 = $f4 - $f6`	FP sub (double precision)
	FP multiply double	`mul.d $f2,$f4,$f6`	`$f2 = $f4 × $f6`	FP multiply (double precision)
	FP divide double	`div.d $f2,$f4,$f6`	`$f2 = $f4 / $f6`	FP divide (double precision)
Data transfer	load word copr. 1	`lwc1 $f1,100($s2)`	`$f1 = Memory[$s2 + 100]`	32-bit data to FP register
	store word copr. 1	`swc1 $f1,100($s2)`	Memory[`$s2` + 100] = `$f1`	32-bit data to memory
Conditional branch	branch on FP true	`bc1t 25`	if (cond == 1) go to PC + 4 + 100	PC-relative branch if FP cond.
	branch on FP false	`bc1f 25`	if (cond == 0) go to PC + 4 + 100	PC-relative branch if not cond.

	FP compare single (eq,ne,lt,le,gt,ge)	`c.lt.s $f2,$f4`	if (`$f2 < $f4`)
			cond = 1; else cond = 0
				FP compare less than single precision

	FP compare double (eq,ne,lt,le,gt,ge)	`c.lt.d $f2,$f4`	if (`$f2 < $f4`)
			cond = 1; else cond = 0
				FP compare less than double precision

MIPS floating-point machine language

Name	Format	Example						Comments
`add.s`	R	17	16	6	4	2	0	`add.s $f2,$f4,$f6`
`sub.s`	R	17	16	6	4	2	1	`sub.s $f2,$f4,$f6`
`mul.s`	R	17	16	6	4	2	2	`mul.s $f2,$f4,$f6`
`div.s`	R	17	16	6	4	2	3	`div.s $f2,$f4,$f6`
`add.d`	R	17	17	6	4	2	0	`add.d $f2,$f4,$f6`
`sub.d`	R	17	17	6	4	2	1	`sub.d $f2,$f4,$f6`
`mul.d`	R	17	17	6	4	2	2	`mul.d $f2,$f4,$f6`
`div.d`	R	17	17	6	4	2	3	`div.d $f2,$f4,$f6`
`lwc1`	I	49	20	2	100			`lwc1 $f2,100($s4)`
`swc1`	I	57	20	2	100			`swc1 $f2,100($s4)`
`bc1t`	I	17	8	1	25			`bc1t 25`

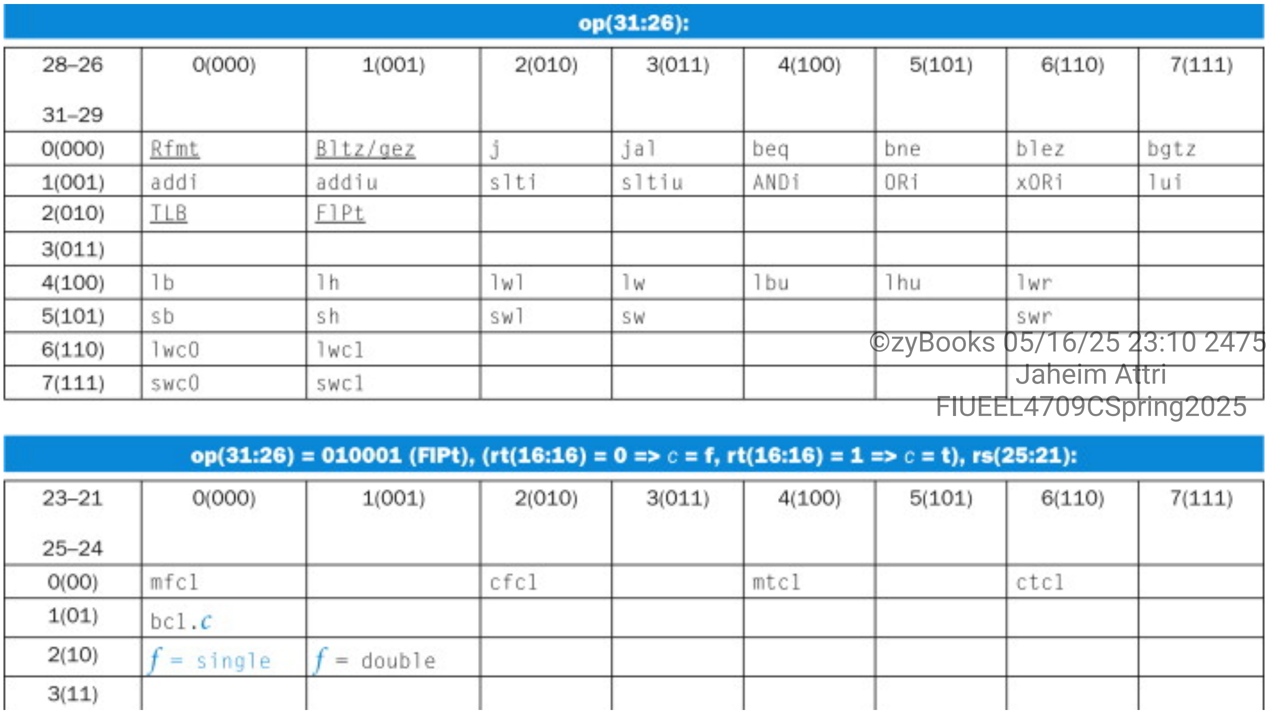


bc1t	I	17	8	1	25			bc1t 25
bc1f	I	17	8	0	25			bc1f 25
c.lt.s	R	17	16	4	2	0	60	c.lt.s $f2,$f4
c.lt.d	R	17	17	4	2	0	60	c.lt.d $f2,$f4
Field size		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions 32 bits

Similar to COD Figure 2.19 (MIPS instruction encoding), the figure below shows the encoding of these instructions.

Figure 3.5.6: MIPS floating-point instruction encoding (COD Figure 3.18).

This notation gives the value of a field by row and by column. For example, in the top portion of the figure, lw is found in row number 4 (100_{two} for bits 31—29 of the instruction) and column number 3 (011_{two} for bits 28—26 of the instruction), so the corresponding value of the op field (bits 31—26) is 100011_{two}. Underscore means the field is used elsewhere. For example, FlPt in row 2 and column 1 (op = 010001_{two}) is defined in the bottom part of the figure. Hence sub.f in row 0 and column 1 of the bottom section means that the funct field (bits 5—0) of the instruction) is 000001_{two} and the op field (bits 31-26) is 010001_{two}. Note that the 5-bit rs field, specified in the middle portion of the figure, determines whether the operation is single precision (*f* = *s*, so rs = 10000) or double precision (*f* = *d*, so rs = 10001). Similarly, bit 16 of the instruction determines if the bc1.c instruction tests for true (bit 16 = 1 => bc1.t) or false (bit 16 = 0 => bc1.f). Instructions in color are described in COD Chapter 2 (Instructions: Language of the Computer) or this chapter, with COD Appendix A (Assemblers, Linkers, and the SPIM Simulator) covering all instructions.

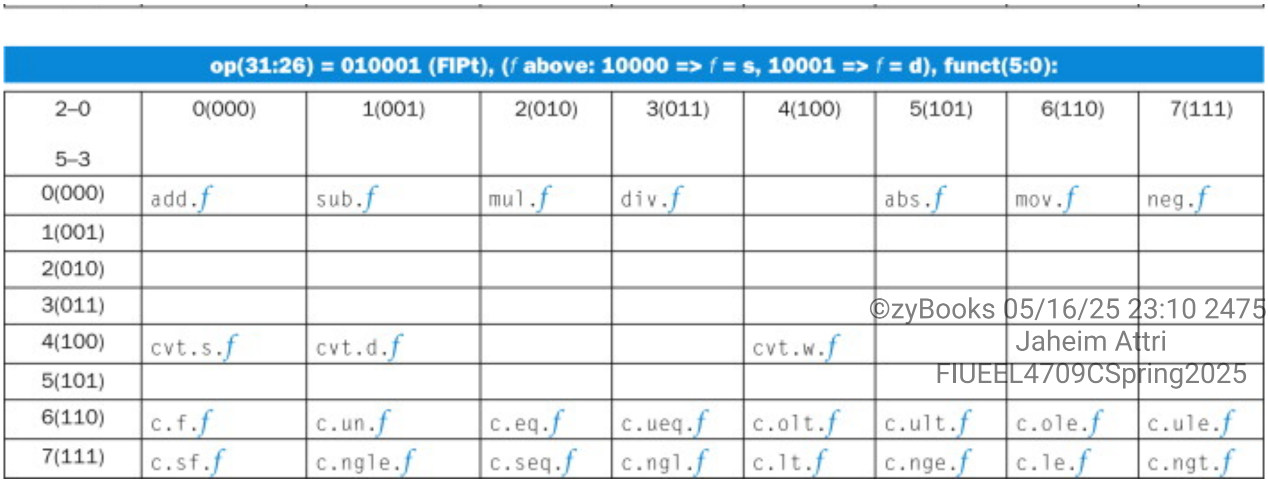


op(31:26):

28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	Rfmt	Bltz/gez	j	jal	beq	bne	blez	bgtz
1(001)	addi	addiu	slti	sltiu	ANDi	ORi	xORi	lui
2(010)	TLB	FlPt						
3(011)								
4(100)	lb	lh	lwl	lw	lbu	lhu	lwr	
5(101)	sb	sh	swl	sw			swr	
6(110)	lwc0	lwc1						
7(111)	swc0	swc1						

op(31:26) = 010001 (FlPt), (rt(16:16) = 0 => *c* = f, rt(16:16) = 1 => *c* = t), rs(25:21):

23–21 25–24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc1		cfc1		mtc1		ctc1	
1(01)	bc1.*c*							
2(10)	*f* = single	*f* = double						
3(11)								



op(31:26) = 010001 (FlPt), (*f* above: 10000 => *f* = s, 10001 => *f* = d), funct(5:0):

2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	add.*f*	sub.*f*	mul.*f*	div.*f*		abs.*f*	mov.*f*	neg.*f*
1(001)								
2(010)								
3(011)								
4(100)	cvt.s.*f*	cvt.d.*f*			cvt.w.*f*			
5(101)								
6(110)	c.f.*f*	c.un.*f*	c.eq.*f*	c.ueq.*f*	c.olt.*f*	c.ult.*f*	c.ole.*f*	c.ule.*f*
7(111)	c.sf.*f*	c.ngle.*f*	c.seq.*f*	c.ngl.*f*	c.lt.*f*	c.nge.*f*	c.le.*f*	c.ngt.*f*

Hardware/Software Interface

One issue that architects face in supporting floating-point arithmetic is whether to use the same registers used by the integer instructions or to add a special set for floating point. Because programs normally perform integer operations and floating-point operations on different data, separating the registers will only slightly increase the number of instructions needed to execute a program. The major impact is to create a separate set of data transfer instructions to move data between floating-point registers and memory.

The benefits of separate floating-point registers are having twice as many registers without using up more bits in the instruction format, having twice the register bandwidth by having separate integer and floating-point register sets, and being able to customize registers to floating point; for example, some computers convert all sized operands in registers into a single internal format.

Example 3.5.1: Compiling a floating-point C program into MIPS assembly code.

Let's convert a temperature in Fahrenheit to Celsius:

```
float f2c (float fahr)
{
   return ((5.0/9.0) *(fahr - 32.0));
}
```

Assume that the floating-point argument `fahr` is passed in `$f12` and the result should go in `$f0`. (Unlike integer registers, floating-point register 0 can contain a number.) What is the MIPS assembly code?

Answer

We assume that the compiler places the three floating-point constants in memory within easy reach of the global pointer `$gp`. The first two instructions load the constants 5.0 and 9.0 into floating-point registers:

```
f2c:
    lwc1 $f16, const5($gp)   # $f16 = 5.0 (5.0 in memory)
    lwc1 $f18, const9($gp)   # $f18 = 9.0 (9.0 in memory)
```

They are then divided to get the fraction 5.0/9.0:

```
    div.s $f16, $f16, $f18   # $f16 = 5.0 / 9.0
```

(Many compilers would divide 5.0 by 9.0 at compile time and save the single constant 5.0/9.0 in memory, thereby avoiding the divide at runtime.) Next, we load the constant 32.0 and then subtract it from `fahr ($f12)`:

```
    lwc1 $f18, const32($gp)  # $f18 = 32.0
    sub.s $f18, $f12, $f18   # $f18 = fahr - 32.0
```

Finally, we multiply the two intermediate results, placing the product in `$f0` as the return result, and then return

```
    mul.s $f0, $f16, $f18    # $f0 = (5/9)*(fahr - 32.0)
    jr $ra                   # return
```

Now let's perform floating-point operations on matrices, code commonly found in scientific programs.

Example 3.5.2: Compiling floating-point C procedure with two-dimensional matrices into MIPS.

Most floating-point calculations are performed in double precision. Let's perform matrix multiply of C = C + A * B. It is commonly called DGEMM, for Double precision, General

Matrix Multiply. We'll see versions of DGEMM again in COD Section 3.8 (Going Faster: Subword Parallelism and Matrix Multiply) and subsequently in COD Chapters 4 (The Processor), 5 (Large and Fast: Exploiting Memory Hierarchy), and 6 (Parallel Processor from Client to Cloud). Let's assume C, A, and B are all square matrices with 32 elements in each dimension.

```
void mm (double c[][], double a[][], double b[][])
{
    int i, j, k;
    for (i = 0; i != 32; i = i + 1)
    for (j = 0; j != 32; j = j + 1)
    for (k = 0; k != 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] *b[k][j];
}
```

The array starting addresses are parameters, so they are in `$a0, $a1,` and `$a2`. Assume that the integer variables are in `$s0, $s1,` and `$s2`, respectively. What is the MIPS assembly code for the body of the procedure?

Answer

Note that `c[i][j]` is used in the innermost loop above. Since the loop index is `k`, the index does not affect `c[i][j]`, so we can avoid loading and storing `c[i][j]` each iteration. Instead, the compiler loads `c[i][j]` into a register outside the loop, accumulates the sum of the products of `a[i][k]` and `b[k][j]` in that same register, and then stores the sum into `c[i][j]` upon termination of the innermost loop.

We keep the code simpler by using the assembly language pseudoinstructions `li` (which loads a constant into a register), and `l.d` and `s.d` (which the assembler turns into a pair of data transfer instructions, `lwc1` or `swc1`, to a pair of floating-point registers).

The body of the procedure starts with saving the loop termination value of 32 in a temporary register and then initializing the three *for* loop variables:

```
mm:...
    li $t1, 32             # $t1 = 32 (row size/loop end)
    li $s0, 0              # i = 0; initialize 1st for loop
L1: li $s1, 0              # j = 0; restart 2nd for loop
L2: li $s2, 0              # k = 0; restart 3rd for loop
```

To calculate the address of `c[i][j]`, we need to know how a 32 × 32, two-dimensional array is stored in memory. As you might expect, its layout is the same as if there were 32 single-dimension arrays, each with 32 elements. So the first step is to skip over the `i` "single-dimensional arrays," or rows, to get the one we want. Thus, we multiply the index in the first dimension by the size of the row, 32. Since 32 is a power of 2, we can use a shift instead:

```
sll $t2, $s0, 5            # $t2 = i * 2^5 (size of row of c)
```

Now we add the second index to select the jth element of the desired row:

```
addu  $t2, $t2, $s1        # $t2 = i * size(row) + j
```

To turn this sum into a byte index, we multiply it by the size of a matrix element in bytes. Since each element is 8 bytes for double precision, we can instead shift left by 3:

```
sll $t2, $t2, 3            # $t2 = byte offset of [i][j]
```

Next we add this sum to the base address of `c`, giving the address of `c[i][j]`, and then load the double precision number `c[i][j]` into `$f4`:

```
addu $t2, $a0, $t2         # $t2 = byte address of c[i][j]
l.d  $f4, 0($t2)           # $f4 = 8 bytes of c[i][j]
```

The following five instructions are virtually identical to the last five: calculate the address and then load the double precision number `b[k][j]`.

```
L3: sll  $t0,  $s2, 5      # $t0 = k * 2^5 (size of row of b)
    addu $t0,  $t0, $s1    # $t0 = k * size(row) + j
    sll  $t0,  $t0, 3      # $t0 = byte offset of [k][j]
    addu $t0,  $a2, $t0    # $t0 = byte address of b[k][j]
    l.d  $f16, 0($t0)      # $f16 = 8 bytes of b[k][j]
```

Similarly, the next five instructions are like the last five: calculate the address and then load the double precision number `a[i][k]`.

```
sll  $t0,  $s0, 5          # $t0 = i * 2^5 (size of row of a)
addu $t0,  $t0, $s2        # $t0 = i * size(row) + k
sll  $t0,  $t0, 3          # $t0 = byte offset of [i][k]
addu $t0,  $a1, $t0        # $t0 = byte address of a[i][k]
l.d  $f18, 0($t0)          # $f18 = 8 bytes of a[i][k]
```

Now that we have loaded all the data, we are finally ready to do some floating-point operations! We multiply elements of `a` and `b` located in registers `$f18` and `$f16`, and then accumulate the sum in `$f4`.

```
mul.d $f16, $f18, $f16     # $f16 = a[i][k] * b[k][j]
```

```
add.d $f4,  $f4,  $f16    # $f4 = c[i][j] + a[i][k] * b[k][j]
```

The final block increments the index `k` and loops back if the index is not 32. If it is 32, and thus the end of the innermost loop, we need to store the sum accumulated in `$f4` into `c[i][j]`.

```
addiu  $s2, $s2, 1         # $k = k + 1
bne    $s2, $t1, L3        # if (k != 32) go to L3
s.d    $f4, 0($t2)         # c[i][j] = $f4
```

Similarly, these final four instructions increment the index variable of the middle and outermost loops, looping back if the index is not 32 and exiting if the index is 32.

```
addiu  $s1, $s1, 1         # $j = j + 1
bne    $s1, $t1, L2        # if (j != 32) go to L2
addiu  $s0, $s0, 1         # $i = i + 1
bne    $s0, $t1, L1        # if (i != 32) go to L1
...
```

COD Figure 3.22 (The x86 assembly language for the body of the nested loops ...) below shows the x86 assembly language code for a slightly different version of DGEMM in COD Figure 3.21 (Unoptimized C version of a double precision matrix multiply ...).

Elaboration

The array layout discussed in the example, called row-major order, is used by C and many other programming languages. Fortran instead uses column-major order, whereby the array is stored column by column.

Elaboration

Only 16 of the 32 MIPS floating-point registers could originally be used for double precision operations: `$f0, $f2, $f4, ..., $f30`. Double precision is computed using pairs of these single precision registers. The odd-numbered floating-point registers were used only to load and store the right half of 64-bit floating-point numbers. MIPS-32 added `l.d` and `s.d` to the instruction set. MIPS-32 also added "paired single" versions of all floating-point instructions, where a single instruction *results in two parallel floating-point operations on two 32-bit operands inside 64-bit registers (see COD Section 3.6 (Parallelism and computer arithmetic: Subword Parallelism)). For example,* `add.ps $f0, $f2, $f4` *is equivalent to* `add.s $f0, $f2, $f4` *followed by* `add.s $f1, $f3, $f5`*.*

Elaboration

Another reason for separate integers and floating-point registers is that microprocessors in the 1980s didn't have enough transistors to put the floating-point unit on the same chip as the integer unit. Hence, the floating-point unit, including the floating-point registers, was optionally available as a second chip. Such optional accelerator chips are called coprocessors, and explain the acronym for floating-point loads in MIPS: `lwc1` *means load word to coprocessor 1, the floating-point unit. (Coprocessor 0 deals with virtual memory, described in COD Chapter 5 (Large and Fast: Exploiting Memory Hierarchy).) Since the early 1990s, microprocessors have integrated floating point (and just about everything else) on chip, and hence the term coprocessor joins accumulator and core memory as quaint terms that date the speaker.*

Elaboration

As mentioned in COD Section 3.4 (Division), accelerating division is more challenging than multiplication. In addition to SRT, another technique to leverage a fast multiplier is Newton's iteration, where division is recast as finding the zero of a function to find the reciprocal 1/c, which is then multiplied by the other operand. Iteration techniques cannot be rounded properly without calculating many extra bits. A TI chip solved this problem by calculating an extra-precise reciprocal.

Elaboration

Java embraces IEEE 754 by name in its definition of Java floating-point data types *and operations. Thus, the code in the first example could have well been generated for a class method that converted Fahrenheit to Celsius.*

The second example above uses multiple dimensional arrays, which are not explicitly supported in Java. Java allows arrays of arrays, but each array may have its own length, unlike multiple dimensional arrays in C. Like the examples in COD Chapter 2 (Instructions: Language of the Computer), a Java version of this second example would require a good deal of checking code for array bounds, including a new length calculation at the end of row access. It would also need to check that the object reference is not null.

Accurate arithmetic

Unlike integers, which can represent exactly every number between the smallest and largest number, floating-point numbers are normally approximations for a number they can't really represent. The reason is that an infinite variety of real numbers exists between, say, 0 and 1, but no more than 2^{53} can be represented exactly in double precision floating point. The best we can do is getting the floating-point representation close to the actual number. Thus, IEEE 754 offers several modes of rounding to let the programmer pick the desired approximation.

Rounding sounds simple enough, but to round accurately requires the hardware to include extra bits in the calculation. In the preceding examples, we were vague on the number of bits that an intermediate representation can occupy, but clearly, if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round. IEEE 754, therefore, always keeps two extra bits on the right during intermediate additions, called *guard* and *round*, respectively. Let's do a decimal example to illustrate their value.

Guard: The first of two extra bits kept on the right during intermediate calculations of floating-point numbers; used to improve rounding accuracy.

Round: Method to make the intermediate floating-point result fit the floating-point format; the goal is typically to find the nearest number that can be represented in the format.

Example 3.5.3: Rounding with guard digits.

Add $2.56_{\text{ten}} \times 10^0$ to $2.34_{\text{ten}} \times 10^2$, assuming that we have three significant decimal digits. Round to the nearest decimal number with three significant decimal digits, first with guard and round digits, and then without them.

Answer

First we must shift the smaller number to the right to align the exponents, so $2.56_{\text{ten}} \times 10^0$ becomes $0.0256_{\text{ten}} \times 10^2$. Since we have guard and round digits, we are able to represent the two least significant digits when we align exponents. The guard digit holds 5 and the round digit holds 6. The sum is

$$\begin{array}{r} \mathbf{2.3400_{ten}} \\ + \mathbf{0.0256_{ten}} \\ \hline \mathbf{2.3656_{ten}} \end{array}$$

Thus the sum is $2.3656_{\text{ten}} \times 10^2$. Since we have two digits to round, we want values 0 to 49 to round down and 51 to 99 to round up, with 50 being the tiebreaker. Rounding the sum up with three significant digits yields $2.37_{\text{ten}} \times 10^2$.

Doing this *without* guard and round digits drops two digits from the calculation. The new sum is then

$$\begin{array}{r} \mathbf{2.34_{ten}} \\ + \mathbf{0.02_{ten}} \\ \hline \mathbf{2.36_{ten}} \end{array}$$

The answer is $2.36_{\text{ten}} \times 10^2$, off by 1 in the last digit from the sum above.

Since the worst case for rounding would be when the actual number is halfway between two floating-point representations, accuracy in floating point is normally measured in terms of the number of bits in error in the least significant bits of the significand; the measure is called the number of *units in the last place*, or *ulp*. If a number were off by 2 in the least significant bits, it would be called off by 2 ulps. Provided there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half ulp.

Units in the last place (***ulp***): The number of bits in error in the least significant bits of the significand between the actual number and the number that can be represented.

PARTICIPATION ACTIVITY 3.5.14: Rounding floating-point numbers.

1) A floating-point value represented in IEEE 754 is typically an

approximation.

○ True

○ False

2) Intermediate calculations of floating-point numbers append two extra bits to improve rounding accuracy.

○ True

○ False

3) *ulp* is a measure of accuracy in floating point numbers.

○ True

○ False

Elaboration

Although the example above really needed just one extra digit, multiply can need two. A binary product may have one leading 0 bit; hence, the normalizing step must shift the product one bit left. This shifts the guard digit into the least significant bit of the product, leaving the round bit to help accurately round the product.

IEEE 754 has four rounding modes: always round up (toward +∞), always round down (toward -∞), truncate, and round to nearest even. The final mode determines what to do if the number is exactly halfway in between. The U.S. Internal Revenue Service (IRS) always rounds 0.50 dollars up, possibly to the benefit of the IRS. A more equitable way would be to round up this case half the time and round down the other half. IEEE 754 says that if the least significant bit retained in a halfway case would be odd, add one; if it's even, truncate. This method always creates a 0 in the least significant bit in the tie-breaking case, giving the rounding mode its name. This mode is the most commonly used, and the only one that Java supports.

The goal of the extra rounding bits is to allow the computer to get the same results as if the intermediate results were calculated to infinite precision and then rounded. To support this goal and round to the nearest even, the standard has a third bit in addition to guard and round; it is set whenever there are nonzero bits to the right of the round bit. This sticky bit allows the computer to see the difference between 0.50 … 00_{ten} and 0.50 … 01_{ten} when rounding.

The sticky bit may be set, for example, during addition, when the smaller number is shifted to the right. Suppose we added $5.01_{ten} \times 10^{-1}$ to $2.34_{ten} \times 10^{2}$ in the example *above. Even with guard and round, we would be adding 0.0050 to 2.34, with a sum of 2.3450. The sticky bit would be set, since there are nonzero bits to the right. Without the sticky bit to remember whether any 1s were shifted off, we would assume the number is equal to 2.345000 … 00 and round to the nearest even of 2.34. With the sticky bit to remember that the number is larger than 2.345000 ... 00, we round instead to 2.35.*

Sticky bit: A bit used in rounding in addition to guard and round that is set whenever there are nonzero bits to the right of the round bit.

Elaboration

PowerPC, SPARC64, AMD SSE5, and Intel AVX architectures provide a single instruction that does a multiply and add on three registers: a = a + (b × c). Obviously, this instruction allows potentially higher floating-point performance for this common operation. Equally important is that instead of performing two roundings—after the multiply and then after the add—which would happen with separate instructions, the multiply add instruction can perform a single rounding after the add. A single rounding step increases the precision of multiply add. Such operations with a single rounding are called fused multiply add. It was added to the IEEE 754-2008 standard (see COD Section 3.11 (Historical perspective and further reading)).

Fused multiply add: A floating-point instruction that performs both a multiply and an add, but rounds only once after the add.

Summary

The Big Picture that follows reinforces the stored-program concept from COD Chapter 2 (Instructions: Language of the Computer); the meaning of the information cannot be determined just by looking at the bits, for the same bits can represent a variety of objects. This section shows that computer arithmetic is finite and thus can disagree with natural arithmetic. For example, the IEEE 754 standard floating-point representation

$$(-1)^S \times (1 + \textbf{Fraction}) \times 2^{(\textbf{Exponent}-\textbf{Bias})}$$

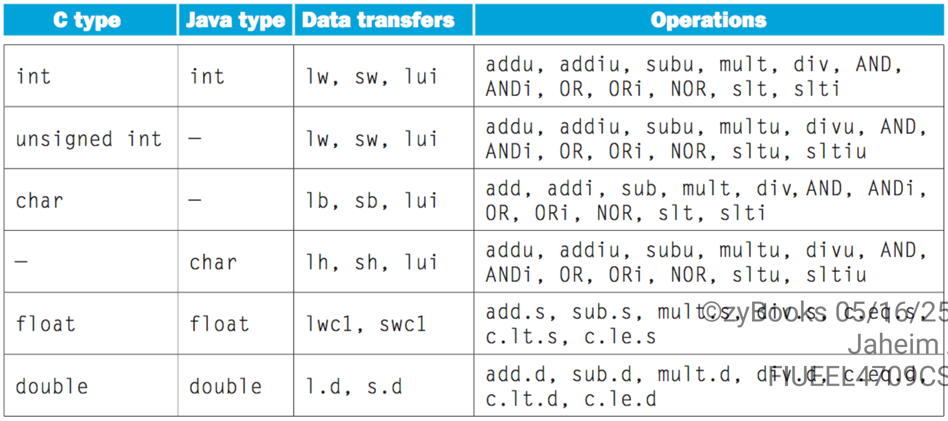
is almost always an approximation of the real number. Computer systems must take care to minimize this gap between computer arithmetic and arithmetic in the real world, and programmers at times need to be aware of the implications of this approximation.

The Big Picture

Bit patterns have no inherent meaning. They may represent signed integers, unsigned integers, floating-point numbers, instructions, and so on. What is represented depends on the instruction that operates on the bits in the word.

The major difference between computer numbers and numbers in the real world is that computer numbers have limited size and hence limited precision; it's possible to calculate a number too big or too small to be represented in a word. Programmers must remember these limits and write programs accordingly.

Figure 3.5.7: C and Java data types, the MIPS data transfer instructions, and instructions that operate on those types.



C type	Java type	Data transfers	Operations
int	int	lw, sw, lui	addu, addiu, subu, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
unsigned int	—	lw, sw, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
char	—	lb, sb, lui	add, addi, sub, mult, div, AND, ANDi, OR, ORi, NOR, slt, slti
—	char	lh, sh, lui	addu, addiu, subu, multu, divu, AND, ANDi, OR, ORi, NOR, sltu, sltiu
float	float	lwc1, swc1	add.s, sub.s, mult.s, div.s, c.eq.s, c.lt.s, c.le.s
double	double	l.d, s.d	add.d, sub.d, mult.d, div.d, c.eq.d, c.lt.d, c.le.d

Hardware/Software Interface

In the last chapter, we presented the storage classes of the programming language C (see the Hardware/Software Interface section in COD Section 2.7 (Instructions for making decisions)). The table above shows some of the C and Java data types, the MIPS data transfer instructions, and instructions that operate on those types that appear in COD Chapter 2 (Instructions: Language of the Computer) and this chapter. Note that Java omits unsigned integers.

PARTICIPATION ACTIVITY | 3.5.15: Check yourself: Half precision floating-point format.

1) The revised IEEE 754-2008 standard added a 16-bit floating-point format with five exponent bits. What do you think is the likely range of numbers the half precision format could represent?

○ 1.0000 00 × 2^{0} to 1.1111 1111 11 × 2^{31}, 0

○ ± 1.0000 0000 0 × 2^{-14} to ±1.1111 1111 1 × 2^{15}, ±0, ±∞, NaN

○ ±1.0000 0000 00 × 2^{-14} to ±1.1111 1111 11 × 2^{15}, ±0, ±∞, NaN

○ ±1.0000 0000 00 × 2^{-15} to ±1.1111 1111 11 × 2^{14}, ±0, ±∞, NaN

Elaboration

To accommodate comparisons that may include NaNs, the standard includes ordered and unordered as options for compares. Hence, the full MIPS (Java does not support unordered compares.)

In an attempt to squeeze every last bit of precision from a floating-point operation, the standard allows some numbers to be represented in unnormalized form. Rather than *having a gap between 0 and the smallest normalized number, IEEE allows denormalized numbers (also known as denorms or subnormals). They have the same exponent as zero but a nonzero fraction. They allow a number to degrade in significance until it becomes 0, called gradual underflow. For example, the smallest positive single precision normalized number is*

$$\mathbf{1.0000\ 0000\ 0000\ 0000\ 0000\ 000_{two} \times 2^{-126}}$$

but the smallest single precision denormalized number is

$$\mathbf{0.0000\ 0000\ 0000\ 0000\ 0000\ 001_{two} \times 2^{-126}, \text{or}\ 1.0_{two} \times 2^{-149}}$$

For double precision, the denorm gap goes from 1.0 × 2^{-1022} to 1.0 × 2^{-1074}.

The possibility of an occasional unnormalized operand has given headaches to floating-point designers who are trying to build fast floating-point units. Hence, many computers cause an exception if an operand is denormalized, letting software complete the operation. Although software implementations are perfectly valid, their lower performance has lessened the popularity of denorms in portable floating-point software. Moreover, if programmers do not expect denorms, their programs may surprise them.

3.6 Parallelism and computer arithmetic: Subword parallelism

Since every desktop microprocessor and smart phone by definition has its own graphical displays, as transistor budgets increased it was inevitable that support would be added for graphics operations.

Many graphics systems originally used 8 bits to represent each of the three primary colors plus 8 bits for a location of a pixel. The addition of speakers and microphones for teleconferencing and video games suggested support of sound as well. Audio samples need more than 8 bits of precision, but 16 bits are sufficient.

Every microprocessor has special support so that bytes and halfwords take up less space when stored in memory (see COD Section 2.9 (Communicating with people)), but due to the infrequency of arithmetic operations on these data sizes in typical integer programs, there was little support beyond data transfers. Architects recognized that many graphics and audio applications would perform the same operation on vectors of this data. By partitioning the carry chains within a 128-bit adder, a processor could use **parallelism** to perform simultaneous operations on short vectors of sixteen 8-bit operands, eight 16-bit operands, four 32-bit operands, or two 64-bit operands. The cost of such partitioned adders was small .

Given that the parallelism occurs within a wide word, the extensions are classified as *subword parallelism*. It is also classified under the more general name of *data level parallelism*. They have been also called vector or SIMD, for single instruction, multiple data (see COD Section 6.6 (Introduction to graphics processing units)). The rising popularity of multimedia applications led to arithmetic instructions that support narrower operations that can easily operate in parallel.

PARTICIPATION ACTIVITY | 3.6.1: Subword parallelism.

1) Subword parallelism takes advantage of byte- and halfword-sized data by _____.

○ turning off the unused bits of a 128-bit adder

○ splitting the add operation to execute across multiple cycles

○ partitioning the adder to perform multiple operations in parallel

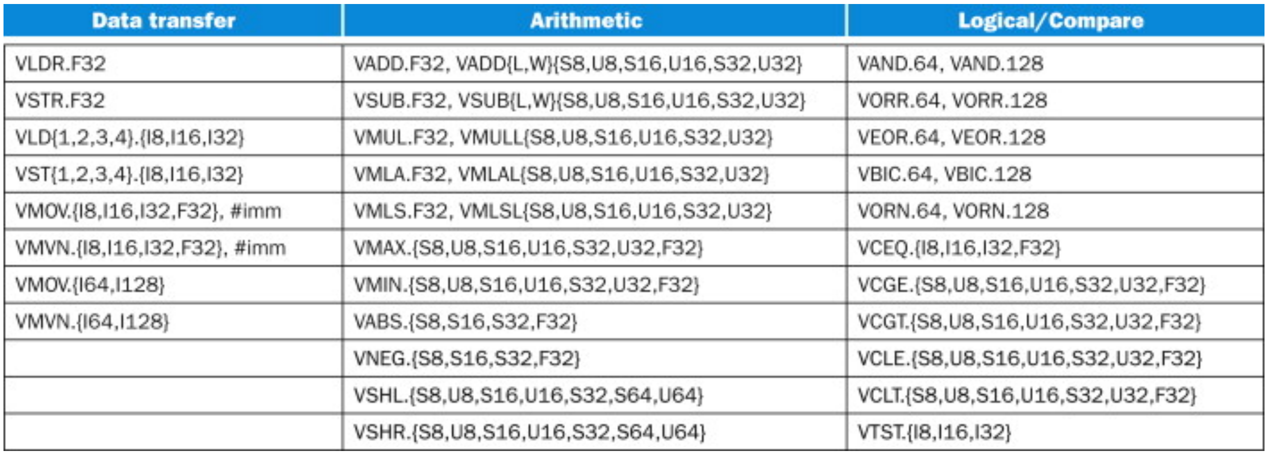
For example, ARM added more than 100 instructions in the NEON multimedia instruction extension to support subword parallelism, which can be used either with ARMv7 or ARMv8. It added 256 bytes of new registers for NEON that can be viewed as 32 registers 8 bytes wide or 16 registers 16 bytes wide. NEON supports all the subword data types you can imagine *except* 64-bit floating point numbers:

- 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers
- 32-bit floating point numbers

The figure below gives a summary of the basic NEON instructions.

Figure 3.6.1: Summary of ARM NEON instructions for subword parallelism (COD Figure 3.19).

We use the curly brackets {} to show optional variations of the basic operations: {S8, U8, 8} stand for signed and unsigned 8-bit integers or 8-bit data where type doesn't matter, of which 16 fit in a 128-bit register; {S16, U16, 16} stand for signed and unsigned 16-bit integers or 16-bit type-less data, of which 8 fit in a 128-bit register; {S32, U32, 32} stand for signed and unsigned 32-bit integers or 32-bit type-less data, of which 4 fit in a 128-bit register; {S64, U64, 64} stand for signed and unsigned 64-bit integers or type-less 64-bit data, of which 2 fit in a 128-bit register; {F32} stand for signed and unsigned 32-bit floating point numbers, of which 4 fit in a 128-bit register. Vector Load reads one n-element structure from memory into 1, 2, 3, or 4 NEON registers. It loads a single n-element structure to one lane (See COD Section 6.6 (Introduction to graphics processing units)), and elements of the register that are not loaded are unchanged. Vector Store writes one n-element structure into memory from 1, 2, 3, or 4 NEON registers.



Data transfer	Arithmetic	Logical/Compare
VLDR.F32	VADD.F32, VADD{L,W}{S8,U8,S16,U16,S32,U32}	VAND.64, VAND.128
VSTR.F32	VSUB.F32, VSUB{L,W}{S8,U8,S16,U16,S32,U32}	VORR.64, VORR.128
VLD{1,2,3,4}.{I8,I16,I32}	VMUL.F32, VMULL{S8,U8,S16,U16,S32,U32}	VEOR.64, VEOR.128
VST{1,2,3,4}.{I8,I16,I32}	VMLA.F32, VMLAL{S8,U8,S16,U16,S32,U32}	VBIC.64, VBIC.128
VMOV.{I8,I16,I32,F32}, #imm	VMLS.F32, VMLSL{S8,U8,S16,U16,S32,U32}	VORN.64, VORN.128
VMVN.{I8,I16,I32,F32}, #imm	VMAX.{S8,U8,S16,U16,S32,U32,F32}	VCEQ.{I8,I16,I32,F32}
VMOV.{I64,I128}	VMIN.{S8,U8,S16,U16,S32,U32,F32}	VCGE.{S8,U8,S16,U16,S32,U32,F32}
VMVN.{I64,I128}	VABS.{S8,S16,S32,F32}	VCGT.{S8,U8,S16,U16,S32,U32,F32}
	VNEG.{S8,S16,S32,F32}	VCLE.{S8,U8,S16,U16,S32,U32,F32}
	VSHL.{S8,U8,S16,U16,S32,S64,U64}	VCLT.{S8,U8,S16,U16,S32,U32,F32}
	VSHR.{S8,U8,S16,U16,S32,S64,U64}	VTST.{I8,I16,I32}

Elaboration

In addition to signed and unsigned integers, ARM includes "fixed-point" format of four sizes called I8, I16, I32, and I64, of which 16, 8, 4, and 2 fit in a 128- bit register, respectively. A portion of the fixed point is for the fraction (to the right of the binary point) and the rest of the data is the integer portion (to the left of the binary point). The location of the binary point is up to the software. Many ARM processors do not have floating point hardware and thus floating point operations must be performed by library routines. Fixed point arithmetic can be significantly faster than software floating point routines, but more work for the programmer.

PARTICIPATION ACTIVITY 3.6.2: Support for subword parallelism.

1) ARMv7 and ARMv8 added new registers for NEON that can be

viewed as 32 8-byte wide registers or 16 _____-byte wide registers to support subword parallelism.

○ 32

○ 16

2) The NEON multimedia instruction extension has little support for subword parallelism beyond data transfer instructions.

○ True

○ False

3) The vector add instruction, VADD, includes support for 8-bit, 16-bit, and 32-bit integers. Operands can be signed or unsigned.

○ True

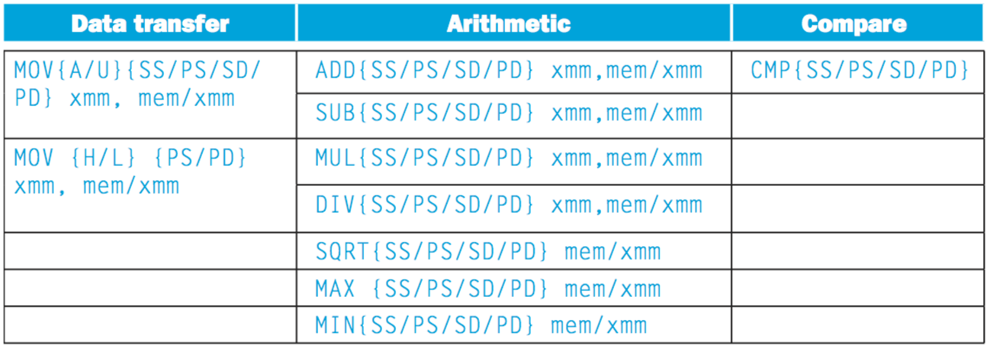
○ False

3.7 Real stuff: Streaming SIMD extensions and advanced vector extensions in x86

The original MMX (*MultiMedia eXtension*) and SSE (*Streaming SIMD Extension*) instructions for the x86 included similar operations to those found in ARM NEON. COD Chapter 2 (Instructions: Language of the Computer) notes that in 2001 Intel added 144 instructions to its architecture as part of SSE2, including double precision floating-point registers and operations. It includes eight 64-bit registers that can be used for floating-point operands. AMD expanded the number to 16 registers, called XMM, as part of AMD64, which Intel relabeled EM64T for its use. The figure below summarizes the SSE and SSE2 instructions.

Figure 3.7.1: The SSE/SSE2 floating-point instructions of the x86 (COD Figure 3.20).

xmm means one operand is a 128-bit SSE2 register, and mem/xmm means the other operand is either in memory or it is an SSE2 register. We use the curly brackets {} to show optional variations of the basic operations: {SS} stands for Scalar Single precision floating point, or one 32-bit operand in a 128-bit register; {PS} stands for Packed Single precision floating point, or four 32-bit operands in a 128-bit register; {SD} stands for Scalar Double precision floating point, or one 64-bit operand in a 128-bit register; {PD} stands for Packed Double precision floating point, or two 64-bit operands in a 128-bit register; {A} means the 128-bit operand is aligned in memory; {U} means the 128-bit operand is unaligned in memory; {H} means move the high half of the 128-bit operand; and {L} means move the low half of the 128-bit operand.



Data transfer	Arithmetic	Compare
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD{SS/PS/SD/PD} xmm,mem/xmm	CMP{SS/PS/SD/PD}
	SUB{SS/PS/SD/PD} xmm,mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL{SS/PS/SD/PD} xmm,mem/xmm	
	DIV{SS/PS/SD/PD} xmm,mem/xmm	
	SQRT{SS/PS/SD/PD} mem/xmm	
	MAX {SS/PS/SD/PD} mem/xmm	
	MIN{SS/PS/SD/PD} mem/xmm	

In addition to holding a single precision or double precision number in a register, Intel allows multiple floating-point operands to be packed into a single 128-bit SSE2 register: four single precision or two double precision. Thus, the 16 floating-point registers for SSE2 are actually 128 bits wide. If the operands can be arranged in memory as 128-bit aligned data, then 128-bit data transfers can load and store multiple operands per instruction. This packed floating-point format is supported by arithmetic operations that can operate simultaneously on four singles (PS) or two doubles (PD).

In 2011 Intel doubled the width of the registers again, now called YMM, with *Advanced Vector Extensions (AVX)*. Thus, a single operation can now specify eight 32-bit floating-point operations or four 64-bit floating-point operations. The legacy SSE and SSE2 instructions now operate on the lower 128 bits of the YMM registers. Thus, to go from 128-bit and 256-bit operations, you prepend the letter "v" (for vector) in front of the SSE2 assembly language operations and then use the YMM register names instead of the XMM register name. For example, the SSE2 instruction to perform two 64-bit floating-point multiplies

```
addpd %xmm0, %xmm4
```

It becomes

```
vaddpd %ymm0, %ymm4
```

which now produces four 64-bit floating-point multiplies

In 2015, Intel doubled the registers again to 512 bits, now called ZIMM, with AVX512 in some of its microprocessors.

Elaboration

AVX also added three address instructions to x86. For example, `vaddpd` *can now specify*

```
vaddpd %ymm0, %ymm1, %ymm4    # %ymm4 = %ymm1 + %ymm0
```

instead of the standard two address version

```
addpd %xmm0, %xmm4            # %xmm4 = %xmm4 + %xmm0
```

(Unlike MIPS, the destination is on the right in x86.) Three addresses can reduce the number of registers and instructions needed for a computation.

PARTICIPATION ACTIVITY 3.7.1: x86 floating-point registers and operations.

1) `mulss`, `mulps`, `mulsd`, and `mulpd` are valid x86 instructions.

○ True

○ False

2) Multiple floating-point operands can be packed into a 128-bit SSE2 register.

○ True

○ False

3) The YMM extension was introduced to support floating-point numbers represented in a 256-bit representation.

○ True

○ False

3.8 Going faster: Subword parallelism and matrix multiply

The declaration on line 7 of the figure below uses the __m512d data type, which tells the compiler the variable will hold 8 double-precision floating-point values (8 x 64 bits = 512 bits). The intrinsic _mm512_load_pd() also on line 7 uses AVX instructions to load 8 double-precision floating-point numbers in parallel (_pd) from the matrix C into c0. The address calculation C+i+j*n represents element C[i+j*n]. Symmetrically, the final step on line 13 uses the intrinsic _mm256_store_pd() to store 8 double-precision floating-point numbers from c0 into the matrix C. As we're going through 8 elements each iteration, the outer for loop on line 4 increments i by 8 instead of by 1 as on line 3 of COD Figure 2.43 of COD Chapter 2.

Figure 3.8.1: Optimized version of DGEMM using C intrinsics to generate AVX512 subword-parallel instructions for the x86. COD Figure 3.22 shows the assembly language produced by the compiler for the inner loop. (COD Figure 3.21).

Inside the loops, on line 10 we first load 8 elements of A again using _mm512_ load_pd(). To multiply these elements by one element of B, we first use the intrinsic _mm512_broadcast_sd(), which makes 8 identical copies of the scalar double precision number--in this case an element of B--in one of the ZMM registers. We then use _mm512_fmadd_pd on line 11 to multiply the 8 double-precision results in parallel and then add the 8 products to the 8 sums in c0.

```
1  #include <x86intrin.h>
2  void dgemm (int n, double* A, double* B, double* C)
3  {
4     for (int i = 0; i < n; i+=8)
5        for (int j = 0; j < n; ++j)
6                {
7           __m512d c0 = _mm512_load_pd(C+i+j*n);   // c0 = C[i][j]
8           for (int k = 0; k < n; k++)
9           {  // c0 += A[i][k]*B[k][j]
10              __m512d bb = _mm512_broadcastsd_pd(_mm_load_sd(B+j*n+k)
11             c0 = _mm512_fmadd_pd(_mm512_load_pd(A+n*k+i), bb, c0);
12          }
13          _m512_store_pd(C+i+j*n, c0);  // C[i][j] = c0
```

```
        }
}
```

The figure below shows resulting x86 code for the body of the inner loops produced by the compiler. You can see the four AVX512 instructions—they all start with v and use pd for parallel double precision—that correspond to the C intrinsics mentioned above. The code is very similar to that in COD Figure 2.44 of COD Chapter 2: the integer instructions are nearly identical (but different registers), and the floating-point instruction differences are generally just going from scalar double (sd) using XMM registers to parallel double (pd) with ZMM registers. One exception is line 4 of the figure below. Every element of A must be multiplied by one element of B. One solution is to place eight identical copies of the 64-bit B element side-by-side into the 512-bit ZMM register, which is just what the instruction vbroadcastsd does. The other difference is that the original program has separate multiply and add floating point operations, whereas the AVX512 version using a single floating point operation in line 6 that performs multiply and add.

Figure 3.8.2: The x86 assembly language for the body of the nested loops generated by compiling the optimized C code in the figure above (COD Figure 3.22).

Note the similarities to COD Figure 2.44 of COD Chapter 2, with the primary difference being that t original floating-point operations are now using ZMM registers and using the pd versions of the instructions for parallel double precision instead of the sd version for scalar double precision and performing a single multiply-add instruction instead of a separate multiply instruction and a separ add instruction.

```
 1 vmovapd      (%r11),%zmm1              # Load 8 elements of C into %
 2 mov          %rbx,%rcx                 # register %rcx = %rbx
 3 xor          %eax,%eax                 # register %eax = 0
 4 vbroadcastsd (%rax,%r8,8),%zmm0        # Make 8 copies of B element
 5 add          &0x8,%rax                 # register $ras = $rax + 8
 6 vfmadd231pd  ($rcx),%zmm0,%zmm1        # Parallel mul & add %zmm0, %
 7 add          %r9,%rcx                  # register %rcx = %rcx
 8 cmp          %r10,%rax                 # compare %r10 to %rax
 9 jne          50 <dgemm+0x50>           # jump if %r10 != %rax
10 add          $0x1,%esi                 # register %esi = %esi + 1
11 vmovap       %zmm1, (%r11)             # Store %zmm1 into 8 C elemer
```

The AVX version is 7.5 times as fast, which is very close to the factor of 8.0 increase that you might hope for from performing 8 times as many operations at a time by using subword parallelism.

PARTICIPATION ACTIVITY 3.8.1: Optimized DGEMM example.

1) _____ intrinsic uses the AVX instruction to load 8 double-precision floating point values .

○ `_mm512_load_pd()`

○ `dgemm()`

○ `_mm512_broadcastsd_pd()`

2) Complete the outer loop of the optimized C version of DGEMM:

```
for (int i = 0; i < n;
______ )
```

○ `++i`

○ `i+=8`

3) Compiling the optimized C code replaces most of the x86 floating-point instructions from a `sd` variation to a _____ variation of the instruction.

○ `pd`

○ `zmm`

○ `__m512d`

3.9 Fallacies and pitfalls

> “ Thus mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.
> *Bertrand Russell, Recent Words on the Principles of Mathematics, 1901.*

Arithmetic fallacies and pitfalls generally stem from the difference between the limited precision of computer arithmetic and the unlimited precision of natural arithmetic.

Fallacy: Just as a left shift instruction can replace an integer multiplication by a power of 2, a right shift is the same as an integer division by a power of 2.

Recall that a binary number *c*, where x_i means the *i*th bit, represents the number

$$\cdots + (x^3 \times 2^3) + (x^2 \times 2^2) + (x^1 \times 2^1) + (x^0 \times 2^0)$$

Shifting the bits of *c* right by *n* bits would seem to be the same as dividing by 2*n*. And this *is* true for unsigned integers. The problem is with signed integers. For example, suppose we want to divide -5_{ten} by 4_{ten}; the quotient should be -1_{ten}. The two's complement representation of -5_{ten} is

$$\mathbf{1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1011_{two}}$$

According to this fallacy, shifting right by two should divide by 4_{ten} (2^2):

$$\mathbf{0011\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}}$$

With a 0 in the sign bit, this result is clearly wrong. The value created by the shift right is actually $1{,}073{,}741{,}822_{\text{ten}}$ instead of -1_{ten}.

A solution would be to have an arithmetic right shift that extends the sign bit instead of shifting in 0s. A 2-bit arithmetic shift right of -5_{ten} produces

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

The result is -2_{ten} instead of -1_{ten}; close, but no cigar.

PARTICIPATION ACTIVITY | 3.9.1: Unsigned and signed integer division.

1) Shifting the bits of an unsigned integer right by n bits divides the integer by 2^n.

○ True

○ False

2) Shifting the bits of a *signed* integer right by n bits divides the integer by 2^n if the shifter extends the sign bit instead of shifting in 0s.

○ True

○ False

Pitfall: Floating-point addition is not associative.

Associativity holds for a sequence of two's complement integer additions, even if the computation overflows. Alas, because floating-point numbers are approximations of real numbers and because computer arithmetic has limited precision, it does not hold for floating-point numbers. Given the great range of numbers that can be represented in floating point, problems occur when adding two large numbers of opposite signs plus a small number. For example, let's see if c + (a + b) = (c + a) + b. Assume c = $-1.5_{\text{ten}} \times 10^{38}$, a = $1.5_{\text{ten}} \times 10^{38}$, and b = 1.0, and that these are all single precision numbers.

$$\begin{aligned} c + (a + b) &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38} + 1.0) \\ &= -1.5_{\text{ten}} \times 10^{38} + (1.5_{\text{ten}} \times 10^{38}) \\ &= 0 \end{aligned}$$

$$\begin{aligned} (c + a) + b &= (-1.5_{\text{ten}} \times 10^{38} + 1.5_{\text{ten}} \times 10^{38}) + 1.0 \\ &= (0.0_{\text{ten}}) + 1.0 \\ &= 1.0 \end{aligned}$$

Since floating-point numbers have limited precision and result in approximations of real results, $1.5_{\text{ten}} \times 10^{38}$ is so much larger than 1.0_{ten} that $1.5_{\text{ten}} \times 10^{38} + 1.0$ is still $1.5_{\text{ten}} \times 10^{38}$. That is why the sum of *c*, *a*, and *b* is 0.0 or 1.0, depending on the order of the floating-point additions, so c + (a + b) ≠ (c + a) + b. Therefore, floating-point addition is *not* associative.

Fallacy: Parallel execution strategies that work for integer data types also work for floating-point data types.

Programs have typically been written first to run sequentially before being rewritten to run concurrently, so a natural question is, "Do the two versions get the same answer?" If the answer is no, you presume there is a bug in the parallel version that you need to track down.

This approach assumes that computer arithmetic does not affect the results when going from sequential to parallel. That is, if you were to add a million numbers together, you would get the same results whether you used 1 processor or 1000 processors. This assumption holds for two's complement integers, since integer addition is associative. Alas, since floating-point addition is not associative, the assumption does not hold.

A more vexing version of this fallacy occurs on a parallel computer where the operating system scheduler may use a different number of processors depending on what other programs are running on a parallel computer. As the varying number of processors from each run would cause the floating-point sums to be calculated in different orders, getting slightly different answers each time despite running identical code with identical input may flummox unaware parallel programmers.

Given this quandary, programmers who write parallel code with floating-point numbers need to verify whether the results are credible even if they don't give the same exact answer as the sequential code. The field that deals with such issues is called numerical analysis, which is the subject of textbooks in its own right. Such concerns are one reason for the popularity of numerical libraries such as LAPACK and SCALAPAK, which have been validated in both their sequential and parallel forms.

PARTICIPATION ACTIVITY | 3.9.2: Floating-point fallacies and pitfalls.

1) Floating-point addition is _____.

○ associative

○ not associative

2) Rewriting programs that contain _____ arithmetic operations to execute in parallel may affect the result.

○ two's complement integer

○ floating-point

3) A parallel program containing floating-point arithmetic operations executing on 10 processors may produce a different result than the same program executing on 1,000 processors.

○ True

○ False

Pitfall: The MIPS instruction add immediate unsigned (addiu) sign-extends its 16-bit immediate field.

Despite its name, add immediate unsigned (`addiu`) is used to add constants to signed integers when we don't care about overflow. MIPS has no subtract immediate instruction, and negative numbers need sign extension, so the MIPS architects decided to sign-extend the immediate field.

Fallacy: Only theoretical mathematicians care about floating-point accuracy.

Newspaper headlines of November 1994 prove this statement is a fallacy (see the figure below). The following is the inside story behind the headlines.

Figure 3.9.1: A sampling of newspaper and magazine articles from November 1994, including the New York Times, San Jose Mercury News, San Francisco Chronicle, and Infoworld (COD Figure 3.23).

The Pentium floating-point divide bug even made the "Top 10 List" of the David Letterman Late Show on television. Intel eventually took a $500 million write-off to replace the buggy chips.

A Closer Look

Warning: Intel Inside

BUSINESS

Intel to work on software patch for Pentium bug

Grove says he's sorry about Pentium bug

Flawed Chip Bruises Intel

Investors react, stock plunges

INFO WORLD

Bug Dodge Booed

Intel Knocked For Response On Flaw

PERSONAL TECHNOLOGY

Chip Shot

Computer Giants' War Over Flaw in Pentium Jolts the PC Industry

Some Scientists Are Angry Over Flaw In Pentium Chip, and Intel's Response

BUSINESS

Intel's Pentium Problem Persists

Pentium woes continue

Faulty FPU flubs math in certain equations

Multithreading gets lost on P100 systems

IBM to stop shipping Pentium PCs

TECHNOLOGY

Intel's Grove Airs Apology for Pentium Over the Internet

The Pentium used a standard floating-point divide algorithm that generates multiple quotient bits per step, using the most significant bits of divisor and dividend to guess the next 2 bits of the quotient. The guess is taken from a lookup table containing -2, -1, 0, +1, or +2. The guess is multiplied by the divisor and subtracted from the remainder to generate a new remainder. Like nonrestoring division, if a previous guess gets too large a remainder, the partial remainder is adjusted in a subsequent pass.

Evidently, there were five elements of the table from the 80486 that Intel engineers thought could never be accessed, and they optimized the logic to return 0 instead of 2 in these situations on the Pentium. Intel was wrong: while the first 11 bits were always correct, errors would show up occasionally in bits 12 to 52, or the 4th to 15th decimal digits.

A math professor at Lynchburg College in Virginia, Thomas Nicely, discovered the bug in September 1994. After calling Intel technical support and getting no official reaction, he posted his discovery on the Internet. This post led to a story in a trade magazine, which in turn caused Intel to issue a press release. It called the bug a glitch that would affect only theoretical mathematicians, with the average spreadsheet user seeing an error every 27,000 years. IBM Research soon counterclaimed that the average spreadsheet user would see an error every 24 days. Intel soon threw in the towel by making the following announcement on December 21:

> “ We at Intel wish to sincerely apologize for our handling of the recently publicized Pentium processor flaw. The Intel Inside symbol means that your computer has a microprocessor second to none in quality and performance. Thousands of Intel employees work very hard to ensure that this is true. But no microprocessor is ever perfect. What Intel continues to believe is technically an extremely minor problem has taken on a life of its own. Although Intel firmly stands behind the quality of the current version of the Pentium processor, we recognize that many users have concerns. We want to resolve these concerns. Intel will exchange the current version of the Pentium processor for an updated version, in which this floating-point divide flaw is corrected, for any owner who requests it, free of charge anytime during the life of their computer.

Analysts estimate that this recall cost Intel $500 million, and Intel engineers did not get a Christmas bonus that year. This story brings up a few points for everyone to ponder. How much cheaper would it have been to fix the bug in July 1994? What was the cost to repair the damage to Intel's reputation? And what is the corporate responsibility in disclosing bugs in a product so widely used and relied upon as a microprocessor?

PARTICIPATION ACTIVITY | 3.9.3: Implementation details.

The add immediate unsigned (`addiu`) instruction can be used to subtract a constant from a signed integer when a programmer doesn't care about overflow. A subtract immediate instruction is not available, thus adding a negative constant achieves the desired result. Ex: `$s1 = $s2 + (-15) = $s2 - 15`
The operands may be signed, so the immediate field is sign-extended.

1) The add immediate unsigned

(`addiu`) instruction can be used to subtract a constant from a signed integer when a programmer doesn't care about overflow.

○ True

○ False

2) An inaccuracy in floating-point division cost Intel an estimated $500 million.

○ True

○ False

3.10 Concluding remarks

Over the decades, computer arithmetic has become largely standardized, greatly enhancing the portability of programs. Two's complement binary integer arithmetic is found in every computer sold today, and if it includes floating point support, it offers the IEEE 754 binary floating-point arithmetic.

Computer arithmetic is distinguished from paper-and-pencil arithmetic by the constraints of limited precision. This limit may result in invalid operations through calculating numbers larger or smaller than the predefined limits. Such anomalies, called "overflow" or "underflow," may result in exceptions or interrupts, emergency events similar to unplanned subroutine calls. COD Chapters 4 (The Processor) and 5 (Large and Fast: Exploiting Memory Hierarchy) discuss exceptions in more detail.

Floating-point arithmetic has the added challenge of being an approximation of real numbers, and care needs to be taken to ensure that the computer number selected is the representation closest to the actual number. The challenges of imprecision and limited representation of floating point are part of the inspiration for the field of numerical analysis. The recent switch to **parallelism** shines the searchlight on numerical analysis again, as solutions that were long considered safe on sequential computers must be reconsidered when trying to find the fastest algorithm for parallel computers that still achieves a correct result.

Data-level parallelism, specifically subword parallelism, offers a simple path to higher performance for programs that are intensive in arithmetic operations for either integer or floating-point data. We showed that we could speed up matrix multiply nearly fourfold by using instructions that could execute four floating-point operations at a time.

PARTICIPATION ACTIVITY 3.10.1: Computer arithmetic.

How to use this tool ⌄

standard **underflow** **numerical analysis**

IEEE 754 is a _____ for floating-point representation and computation.

The field of _____ examines how to solve mathematical problems using imprecision and limited representation of data.

A situation where an operation yields a number that is smaller in magnitude (closer to zero) than the smallest value representable is called _____.

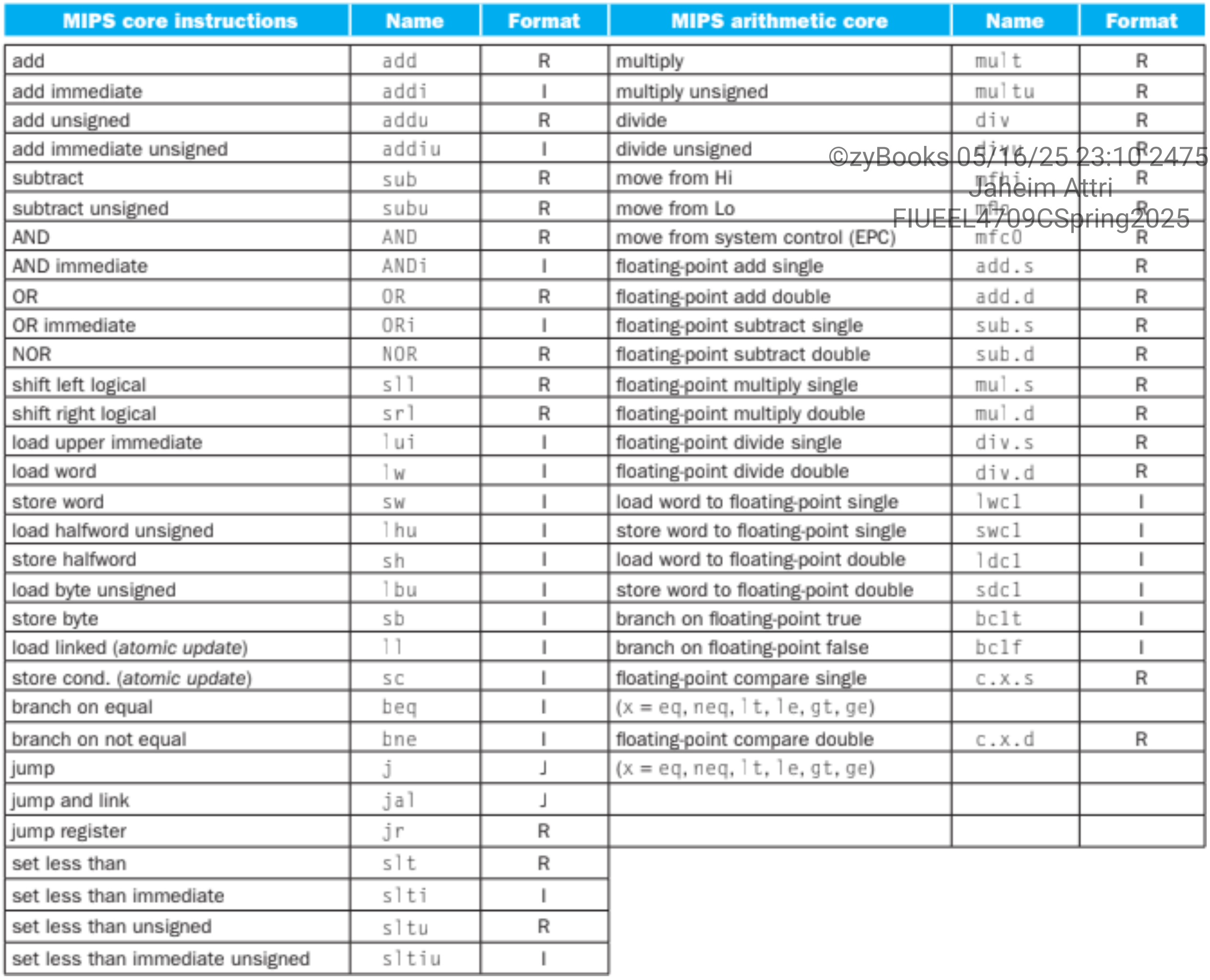
Reset

With the explanation of computer arithmetic in this chapter comes a description of much more of the MIPS instruction set. One point of confusion is the instructions covered in these chapters versus instructions executed by MIPS chips versus the instructions accepted by MIPS assemblers. Two figures try to make this clear.

The figure below (The MIPS instruction set) lists the MIPS instructions covered in this chapter and COD Chapter 2 (Instructions: Language of the Computer). We call the set of instructions on the left-hand side of the figure the *MIPS core*. The instructions on the right we call the *MIPS arithmetic core*. On the left of the other figure below (Remaining MIPS-32 and Pseudo MIPS instruction sets) are the instructions the MIPS processor executes that are not found in the figure below (The MIPS instruction set). We call the full set of hardware instructions MIPS-32. On the right of the other figure below (Remaining MIPS-32 and Pseudo MIPS instruction sets) are the instructions accepted by the assembler that are not part of MIPS-32. We call this set of instructions *Pseudo MIPS*.

Figure 3.10.1: The MIPS instruction set (COD Figure 3.24).

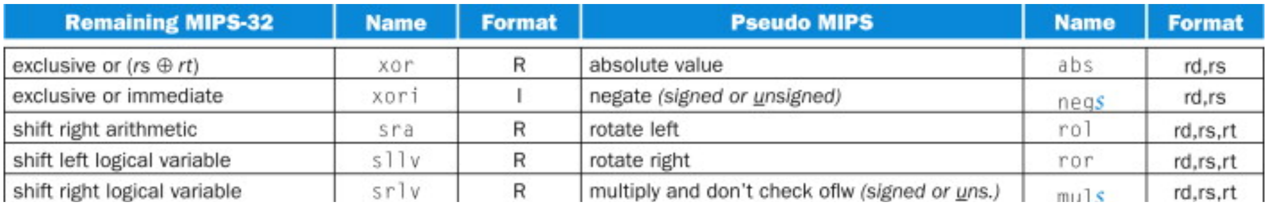
This book concentrates on the instructions in the left column.



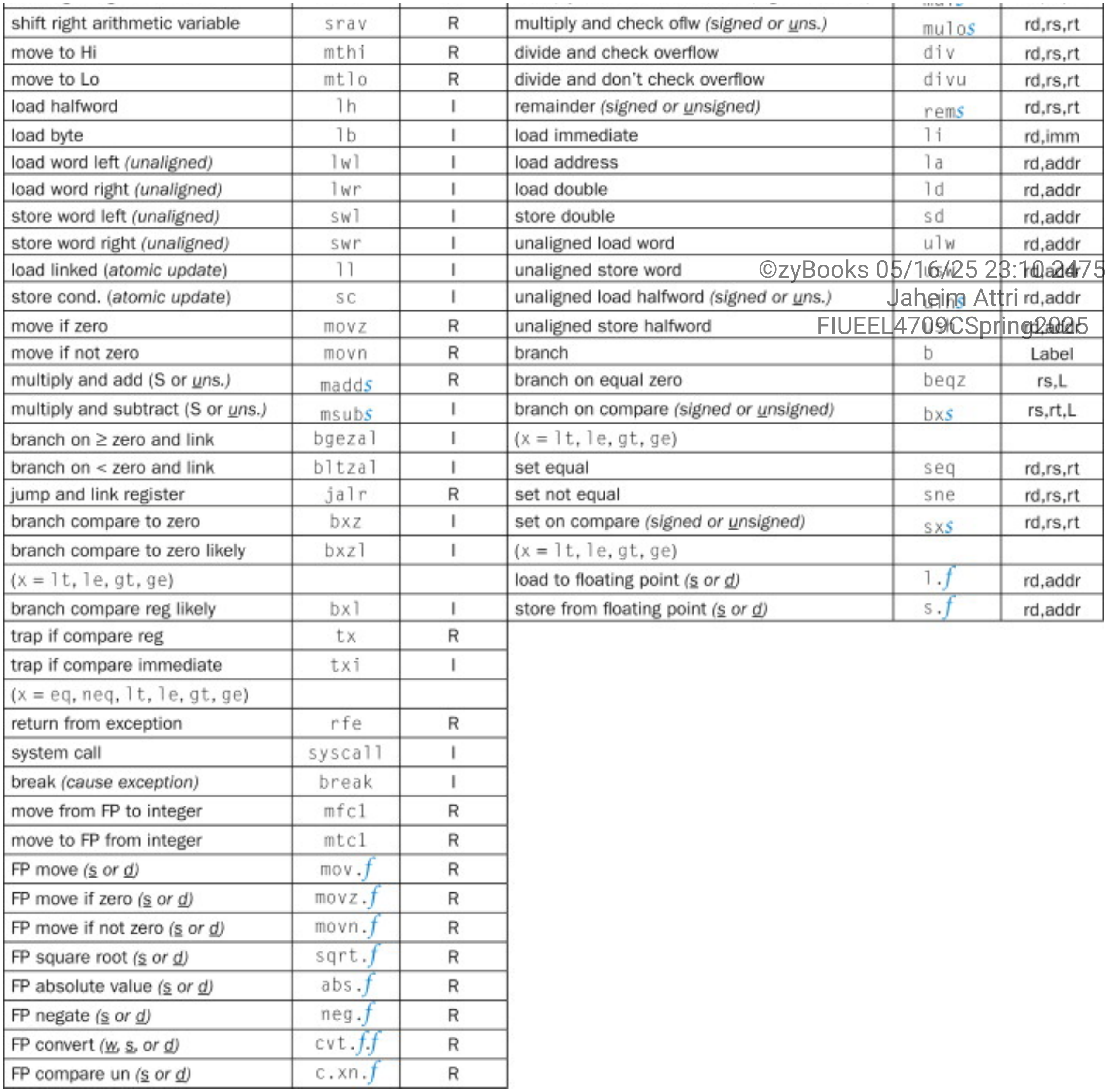
MIPS core instructions	Name	Format	MIPS arithmetic core	Name	Format
add	add	R	multiply	mult	R
add immediate	addi	I	multiply unsigned	multu	R
add unsigned	addu	R	divide	div	R
add immediate unsigned	addiu	I	divide unsigned	divu	R
subtract	sub	R	move from Hi	mfhi	R
subtract unsigned	subu	R	move from Lo	mflo	R
AND	AND	R	move from system control (EPC)	mfc0	R
AND immediate	ANDi	I	floating-point add single	add.s	R
OR	OR	R	floating-point add double	add.d	R
OR immediate	ORi	I	floating-point subtract single	sub.s	R
NOR	NOR	R	floating-point subtract double	sub.d	R
shift left logical	sll	R	floating-point multiply single	mul.s	R
shift right logical	srl	R	floating-point multiply double	mul.d	R
load upper immediate	lui	I	floating-point divide single	div.s	R
load word	lw	I	floating-point divide double	div.d	R
store word	sw	I	load word to floating-point single	lwc1	I
load halfword unsigned	lhu	I	store word to floating-point single	swc1	I
store halfword	sh	I	load word to floating-point double	ldc1	I
load byte unsigned	lbu	I	store word to floating-point double	sdc1	I
store byte	sb	I	branch on floating-point true	bc1t	I
load linked (*atomic update*)	ll	I	branch on floating-point false	bc1f	I
store cond. (*atomic update*)	sc	I	floating-point compare single	c.x.s	R
branch on equal	beq	I	(x = eq, neq, lt, le, gt, ge)		
branch on not equal	bne	I	floating-point compare double	c.x.d	R
jump	j	J	(x = eq, neq, lt, le, gt, ge)		
jump and link	jal	J			
jump register	jr	R			
set less than	slt	R			
set less than immediate	slti	I			
set less than unsigned	sltu	R			
set less than immediate unsigned	sltiu	I			

Figure 3.10.2: Remaining MIPS-32 and Pseudo MIPS instruction sets (COD Figure 3.25).

f means single (s) or double (d) precision floating-point instructions, and *s* means signed and unsigned (u) versions. MIPS-32 also has FP instructions for multiply and add/sub (madd.*f*/ msub.*f*), ceiling (ceil.*f*), truncate (trunc.*f*), round (round.*f*), and reciprocal (recip.*f*). The underscore represents the letter to include to represent that datatype.



Remaining MIPS-32	Name	Format	Pseudo MIPS	Name	Format
exclusive or (*rs* ⊕ *rt*)	xor	R	absolute value	abs	rd,rs
exclusive or immediate	xori	I	negate (*signed or unsigned*)	negs	rd,rs
shift right arithmetic	sra	R	rotate left	rol	rd,rs,rt
shift left logical variable	sllv	R	rotate right	ror	rd,rs,rt
shift right logical variable	srlv	R	multiply and don't check oflw (*signed or uns.*)	muls	rd,rs,rt



shift right arithmetic variable	srav	R	multiply and check oflw *(signed or uns.)*	mulo*s*	rd,rs,rt
move to Hi	mthi	R	divide and check overflow	div	rd,rs,rt
move to Lo	mtlo	R	divide and don't check overflow	divu	rd,rs,rt
load halfword	lh	I	remainder *(signed or unsigned)*	rem*s*	rd,rs,rt
load byte	lb	I	load immediate	li	rd,imm
load word left *(unaligned)*	lwl	I	load address	la	rd,addr
load word right *(unaligned)*	lwr	I	load double	ld	rd,addr
store word left *(unaligned)*	swl	I	store double	sd	rd,addr
store word right *(unaligned)*	swr	I	unaligned load word	ulw	rd,addr
load linked *(atomic update)*	ll	I	unaligned store word	usw	rd,addr
store cond. *(atomic update)*	sc	I	unaligned load halfword *(signed or uns.)*	ulh*s*	rd,addr
move if zero	movz	R	unaligned store halfword	ush	rd,addr
move if not zero	movn	R	branch	b	Label
multiply and add (S or *uns.)*	madd*s*	R	branch on equal zero	beqz	rs,L
multiply and subtract (S or *uns.)*	msub*s*	I	branch on compare *(signed or unsigned)*	bx*s*	rs,rt,L
branch on ≥ zero and link	bgezal	I	(x = lt, le, gt, ge)		
branch on < zero and link	bltzal	I	set equal	seq	rd,rs,rt
jump and link register	jalr	R	set not equal	sne	rd,rs,rt
branch compare to zero	bxz	I	set on compare *(signed or unsigned)*	sx*s*	rd,rs,rt
branch compare to zero likely	bxzl	I	(x = lt, le, gt, ge)		
(x = lt, le, gt, ge)			load to floating point *(s or d)*	l.*f*	rd,addr
branch compare reg likely	bxl	I	store from floating point *(s or d)*	s.*f*	rd,addr
trap if compare reg	tx	R			
trap if compare immediate	txi	I			
(x = eq, neq, lt, le, gt, ge)					
return from exception	rfe	R			
system call	syscall	I			
break *(cause exception)*	break	I			
move from FP to integer	mfc1	R			
move to FP from integer	mtc1	R			
FP move *(s or d)*	mov.*f*	R			
FP move if zero *(s or d)*	movz.*f*	R			
FP move if not zero *(s or d)*	movn.*f*	R			
FP square root *(s or d)*	sqrt.*f*	R			
FP absolute value *(s or d)*	abs.*f*	R			
FP negate *(s or d)*	neg.*f*	R			
FP convert *(w, s, or d)*	cvt.*f*.*f*	R			
FP compare un *(s or d)*	c.xn.*f*	R			

PARTICIPATION ACTIVITY 3.10.2: Classifications of MIPS instructions.

Refer to the above tables to match the classification to the instruction.

How to use this tool ⌄

MIPS core **MIPS-32** **Pseudo MIPS**

subu

movz

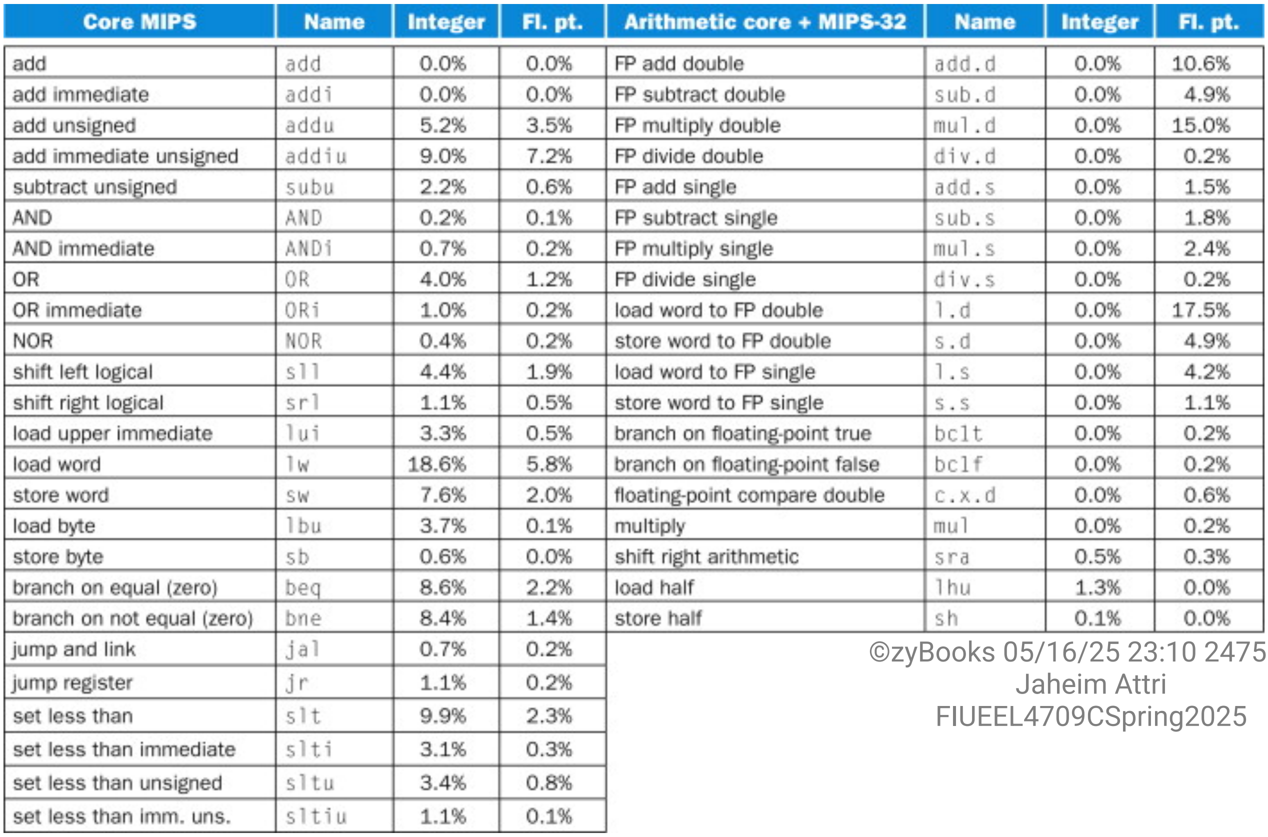
ror

Reset

The figure below gives the popularity of the MIPS instructions for SPEC CPU2006 integer and floating-point benchmarks. All instructions are listed that were responsible for at least 0.2% of the instructions executed.

Figure 3.10.3: The frequency of the MIPS instructions for SPEC CPU2006 integer and floating point (COD Figure 3.26).

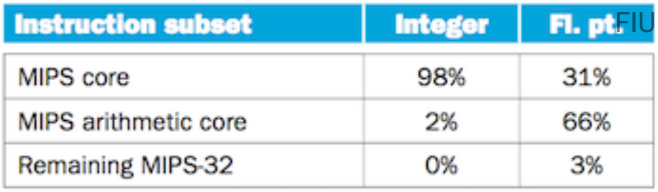
All instructions that accounted for at least 0.2% of the instructions are included in the table. Pseudoinstructions are converted into MIPS-32 before execution, and hence do not appear here.



Core MIPS	Name	Integer	Fl. pt.	Arithmetic core + MIPS-32	Name	Integer	Fl. pt.
add	add	0.0%	0.0%	FP add double	add.d	0.0%	10.6%
add immediate	addi	0.0%	0.0%	FP subtract double	sub.d	0.0%	4.9%
add unsigned	addu	5.2%	3.5%	FP multiply double	mul.d	0.0%	15.0%
add immediate unsigned	addiu	9.0%	7.2%	FP divide double	div.d	0.0%	0.2%
subtract unsigned	subu	2.2%	0.6%	FP add single	add.s	0.0%	1.5%
AND	AND	0.2%	0.1%	FP subtract single	sub.s	0.0%	1.8%
AND immediate	ANDi	0.7%	0.2%	FP multiply single	mul.s	0.0%	2.4%
OR	OR	4.0%	1.2%	FP divide single	div.s	0.0%	0.2%
OR immediate	ORi	1.0%	0.2%	load word to FP double	l.d	0.0%	17.5%
NOR	NOR	0.4%	0.2%	store word to FP double	s.d	0.0%	4.9%
shift left logical	sll	4.4%	1.9%	load word to FP single	l.s	0.0%	4.2%
shift right logical	srl	1.1%	0.5%	store word to FP single	s.s	0.0%	1.1%
load upper immediate	lui	3.3%	0.5%	branch on floating-point true	bc1t	0.0%	0.2%
load word	lw	18.6%	5.8%	branch on floating-point false	bc1f	0.0%	0.2%
store word	sw	7.6%	2.0%	floating-point compare double	c.x.d	0.0%	0.6%
load byte	lbu	3.7%	0.1%	multiply	mul	0.0%	0.2%
store byte	sb	0.6%	0.0%	shift right arithmetic	sra	0.5%	0.3%
branch on equal (zero)	beq	8.6%	2.2%	load half	lhu	1.3%	0.0%
branch on not equal (zero)	bne	8.4%	1.4%	store half	sh	0.1%	0.0%
jump and link	jal	0.7%	0.2%				
jump register	jr	1.1%	0.2%				
set less than	slt	9.9%	2.3%				
set less than immediate	slti	3.1%	0.3%				
set less than unsigned	sltu	3.4%	0.8%				
set less than imm. uns.	sltiu	1.1%	0.1%				

Note that although programmers and compiler writers may use MIPS-32 to have a richer menu of options, MIPS core instructions dominate integer SPEC CPU2006 execution, and the integer core plus arithmetic core dominate SPEC CPU2006 floating point, as the figure below shows.

Figure 3.10.4: The frequency of MIPS core, MIPS arithmetic core, and Remaining MIPS-32 instructions for SPEC CPU2006 integer and floating point.



Instruction subset	Integer	Fl. pt.
MIPS core	98%	31%
MIPS arithmetic core	2%	66%
Remaining MIPS-32	0%	3%

For the rest of the book, we concentrate on the MIPS core instructions—the integer instruction set excluding multiply and divide—to make the explanation of computer design easier. As you can see, the MIPS core includes the most popular MIPS instructions; be assured that understanding a computer that runs the MIPS core will give you sufficient background to understand even more ambitious computers. No matter what the instruction set or its size—MIPS, RISC-V, ARM, x86—never forget that bit patterns have no inherent meaning. The same bit pattern may represent a signed integer, unsigned integer, floating-point number, string, instruction, and so on. In stored program computers, it is the operation on the bit pattern that determines its meaning.

PARTICIPATION ACTIVITY 3.10.3: Concluding remarks.

1) Which of the following instruction classifications occur most frequently in the execution of the SPEC CPU2006 floating-point benchmarks?

○ MIPS core

○ Remaining MIPS-32

○ Pseudo MIPS

2) The leading 1 indicates that the bit pattern below represents a negative number.

```
1010 ... 0010 1100
```

○ True

○ False

○ Not enough information to determine what the bit pattern represents.

3.11 Historical perspective and further reading

> Gresham's Law ("Bad money drives out Good") for computers would say, "The Fast drives out the Slow even if the Fast is wrong."
> *W. Kahan, 1992.*

This section surveys the history of the floating point going back to von Neumann, including the surprisingly controversial IEEE standards effort, the rationale for the 80-bit stack architecture for floating point in the IA-32, and an update on the next round of the standard.

At first it may be hard to imagine a subject of less excitement than the correctness of computer arithmetic or its accuracy, and harder still to understand why a subject so old and mathematical should be so controversial. Computer arithmetic is as old as computing itself, and some of the subject's earliest notions, like the economical reuse of registers during serial multiplication and division, still command respect today. Maurice Wilkes [1985] recalled a conversation about that notion during his visit to the United States in 1946, before the earliest stored-program computer had been built:

> … a project under von Neumann was to be set up at the Institute of Advanced Studies in Princeton. … Goldstine explained to me the principal features of the design, including the device whereby the digits of the multiplier were put into the tail of the accumulator and shifted out as the least significant part of the product was shifted in. I expressed some admiration at the way registers and shifting circuits were arranged … and Goldstine remarked that things of that nature came very easily to von Neumann.

There is no controversy here; it can hardly arise in the context of exact integer arithmetic, so long as there is general agreement on what integer the correct result should be. However, as soon as approximate arithmetic enters the picture, so does controversy, as if one person's "negligible" must be another's "everything."

The first dispute

Floating-point arithmetic kindled disagreement before it was ever built. John von Neumann was aware of Konrad Zuse's proposal for a computer in Germany in 1939 that was never built, probably because the floating point made it appear too complicated to finish before the Germans expected World War II to end. Hence, von Neumann refused to include it in the computer he built at Princeton. In an influential report coauthored in 1946 with H. H. Goldstine and A. W. Burks, he gave the arguments for and against floating point. In favor:

> “ … to retain in a sum or product as many significant digits as possible and … to free the human operator from the burden of estimating and inserting into a problem "scale factors"—multiplication constants which serve to keep numbers within the limits of the machine.

Floating point was excluded for several reasons:

> “ There is, of course, no denying the fact that human time is consumed in arranging for the introduction of suitable scale factors. We only argue that the time consumed is a very small percentage of the total time we will spend in preparing an interesting problem for our machine. The first advantage of the floating point is, we feel, somewhat illusory. In order to have such a floating point, one must waste memory capacity which could otherwise be used for carrying more digits per word. It would therefore seem to us not at all clear whether the modest advantages of a floating binary point offset the loss of memory capacity and the increased complexity of the arithmetic and control circuits.

The argument seems to be that most bits devoted to exponent fields would be bits wasted. Experience has proved otherwise.

One software approach to accommodate reals without floating-point hardware was called *floating vectors*; the idea was to compute at runtime one scale factor for a whole array of numbers, choosing the scale factor so that the array's biggest number would barely fill its field. By 1951, James H. Wilkinson had used this scheme extensively for matrix computations. The problem proved to be that a program might encounter a very large value, and hence the scale factor must accommodate these rare large numbers. The common numbers would thus have many leading 0s, since all numbers had to use a single scale factor. Accuracy was sacrificed, because the least significant bits had to be lost on the right to accommodate leading 0s. This wastage became obvious to practitioners on early computers that displayed all their memory bits as dots on cathode ray tubes (like TV screens) because the loss of precision was visible. Where floating point deserved to be used, no practical alternative existed.

Thus, true floating-point hardware became popular because it was useful. By 1957, floating-point hardware was almost ubiquitous. A decimal floating-point unit was available for the IBM 650, and soon the IBM 704, 709, 7090, 7094 … series would offer binary floating-point hardware for double as well as single precision.

As a result, everybody had floating point, but every implementation was different.

Diversity versus portability

Since roundoff introduces some error into almost all floating-point operations, to complain about another bit of error seems picayune. So for 20 years, nobody complained much that those operations behaved a little differently on different computers. If software required clever tricks to circumvent those idiosyncrasies and finally deliver results correct in all but the last several bits, such tricks were deemed part of the programmer's art. For a long time, matrix computations mystified most people who had no notion of error analysis; perhaps this continues to be true. That may be why people are still surprised that numerically stable matrix computations depend upon the quality of arithmetic in so few places, far fewer than are generally supposed. Books by Wilkinson and widely used software packages like Linpack and Eispack sustained a false impression, widespread in the early 1970s, that a modicum of skill sufficed to produce *portable* numerical software.

"Portable" here means that the software is distributed as source code in some standard language to be compiled and executed on practically any commercially significant computer, and that it will then perform its task as well as any other program performs that task on that computer. Insofar as numerical software has often been thought to consist entirely of computer-independent mathematical formulas, its portability has often been taken for granted; the mistake in that presumption will become clear shortly.

Packages like Linpack and Eispack cost so much to develop—over a hundred dollars per line of Fortran delivered—that they could not have been developed without U.S. government subsidy; their portability was a precondition for that subsidy. But nobody thought to distinguish how various components contributed to their cost. One component was algorithmic—devise an algorithm that deserves to work on at least one computer despite its roundoff and over-/underflow limitations. Another component was the software engineering effort required to achieve and confirm portability to the diverse computers commercially significant at the time; this component grew more onerous as ever more diverse floating-point arithmetics blossomed in the 1970s. And yet scarcely anybody realized how much that diversity inflated the cost of such software packages.

PARTICIPATION ACTIVITY | 3.11.1: Floating-point: Inclusion and portability.

1) Floating-point hardware was included in the first computers.

○ True

○ False

2) Floating-point hardware became standardized by the late 1950s.

○ True

○ False

3) Numerical software in the late 1950s was portable because the software consists entirely of computer-independent mathematical formulas.

○ True

○ False

A backward step

Early evidence that somewhat different arithmetics could engender grossly different software development costs was presented in 1964. It happened at a meeting of SHARE, the IBM mainframe users' group, at which IBM announced System/360, the successor to the 7094 series. One of the speakers described the tricks he had been forced to devise to achieve a level of quality for the S/360 library that was not quite so high as he had previously achieved for the 7094.

Part of the trouble could have been foretold by von Neumann, had he still been alive. In 1948, he and Goldstine had published a lengthy error analysis so difficult and so pessimistic that hardly anybody paid attention to it. It did predict correctly, however, that computations with larger arrays of data would probably fall prey to roundoff more often. IBM S/360s had bigger memories than 7094s, so data arrays could grow bigger, and they did. To make matters worse, the S/360s had narrower single precision words (32 bits versus 36) and used a cruder arithmetic (hexadecimal or base 16 versus binary or base 2) with consequently poorer worst-case precision (21 significant bits versus 27) than the old 7094s. Consequently, software that had almost always provided (barely) satisfactory accuracy on 7094s too often produced inaccurate results when run on S/360s. The quickest way to recover adequate accuracy was to replace old codes' single precision declarations with double precision before recompilation for the S/360. This practice exercised S/360 double precision far more than had been expected.

The early S/360's worst troubles were caused by lack of a guard digit in double precision. This lack showed up in multiplication as a failure of identities like 1.0 * $x = x$ because multiplying x by 1.0 dropped x's last hexadecimal digit (4 bits). Similarly, if x and y were very close but had different exponents, subtraction dropped off the last digit of the smaller operand before computing $x - y$. This last aberration in double precision undermined a precious theorem that single precision then (and now) honored: If $1/2 \le x/y \le 2$, then no rounding error can occur when $x - y$ is computed; it must be computed exactly.

Innumerable computations had benefited from this minor theorem, most often unwittingly, for several decades before its first formal announcement and proof. We had been taking all this stuff for granted.

The identities and theorems about exact relationships that persisted, despite roundoff, with reasonable implementations of approximate arithmetic were not appreciated until they were lost.

Previously, it had been thought that the things that matter were precision (how many significant digits were carried) and range (the spread between over-/underflow thresholds). Since the S/360's double precision had more precision and wider range than the 7094's, software was expected to continue to work at least as well as before. But it didn't.

Programmers who had matured into program managers were appalled at the cost of converting 7094 software to run on S/360s. A small subcommittee of SHARE proposed improvements to the S/360 floating point. This committee was surprised and grateful to get a fair part of what they asked for from IBM, including all-important guard digits. By 1968, these had been retrofitted to S/360s in the field at considerable expense; worse than that was customers' loss of faith in IBM's infallibility (a lesson learned by Intel 30 years later; see COD Figure 3.23). IBM employees who can remember the incident still shudder.

The people who built the bombs

Seymour Cray was associated for decades with the CDC and Cray computers that were, when he built them, the world's biggest and fastest. He always understood what his customers wanted most: *speed*. And he gave it to them even if, in so doing, he also gave them arithmetics more "interesting" than anyone else's. Among his customers have been the great government laboratories like those at Livermore and Los Alamos, where nuclear weapons were designed. The challenges of "interesting" arithmetics were pretty tame to people who had to overcome Mother Nature's challenges.

Perhaps all of us could learn to live with arithmetic idiosyncrasy if only one computer's idiosyncrasies had to be endured. Instead, when accumulating different computers' different anomalies, software dies the Death of a Thousand Cuts. Here is an example from Cray's computers:

```
if (x == 0.0) y = 17.0 else y = z/x
```

Could this statement be stopped by a divide-by-zero error? On a CDC 6600 it could. The reason was a conflict between the 6600's adder, where x was compared with 0.0, and the multiplier and divider. The adder's comparison examined x's leading 13 bits, which sufficed to distinguish zero from normal nonzero floating-point numbers x. The multiplier and divider examined only 12 leading bits. Consequently, tiny numbers existed that were nonzero to the adder but zero to the multiplier and divider! To avoid disasters with these tiny numbers, programmers learned to replace statements like the one above with

```
if (1.0 * x == 0.0) y = 17.0 else y = z/x
```

But this statement is unsafe to use in would-be portable software because it malfunctions obscurely on other computers designed by Cray, the ones marketed by Cray Research, Inc. If x was so huge that 2.0 * x would overflow, then 1.0 * x might overflow too! Overflow happens because Cray computers check the product's exponent before the product's exponent has been normalized, just to save the delay of a single AND gate.

Rounding error anomalies that are far worse than the over-/underflow anomaly just discussed also affect Cray computers. The worst error came from the lack of a guard digit in add/subtract, an affliction of IBM S/360s. Further bad luck for software is occasioned by the way Cray economized his multiplier; about one-third of the bits that normal multiplier arrays generate have been left out of his multipliers, because they would contribute less than a unit to the last place of the final Cray-rounded product. Consequently, a Cray multiplier errs by almost a bit more than might have been expected. This error is compounded when division takes three multiplications to improve an approximate reciprocal of the divisor and then multiply the numerator by it. Square root compounds a few more multiplication errors.

The fast way drove out the slow, even though the fast was occasionally slightly wrong.

PARTICIPATION ACTIVITY | 3.11.2: Floating-point: User-side challenges.

1) In an effort to achieve similar floating-point accuracy, programmers converting code from the IBM 7094 to the S/360 replaced single precision declarations with double precision declarations.

○ True

○ False

2) Reduced floating-point accuracy and the high cost of converting code from the IBM 7094 to the S/360 led to the formation of a subcommittee of IBM mainframe users to propose improvements to the S/360 floating point.

○ True

○ False

3) The following statements produce the same results on the CDC 6600 computer:

```
 if (x == 0.0) y = 17.0
else y = z/x
```

```
 if (1.0 * x == 0.0) y =
17.0 else y = z/x
```

○ True

○ False

Making the world safe for floating point, or vice versa

William Kahan was an undergraduate at the University of Toronto in 1953 when he learned to program its Ferranti-Manchester Mark-I computer. Because he entered the field early, Kahan became acquainted with a wide range of devices and a large proportion of the personalities active in computing; the numbers of both were small at that time. He has performed computations on slide rules, desktop mechanical calculators, tabletop analog differential analyzers, and so on; he has used all but the earliest electronic computers and calculators mentioned in this book.

Kahan's desire to deliver reliable software led to an interest in error analysis that intensified during two years of postdoctoral study in England, where he became acquainted with Wilkinson. In 1960, he resumed teaching at Toronto, where an IBM 7090 had been acquired, and was granted free rein to tinker with its operating system, Fortran compiler, and runtime library. (He denies that he ever came near the 7090 hardware with a soldering iron but admits asking to do so.) One story from that time illuminates how misconceptions and numerical anomalies in computer systems can incur awesome hidden costs.

A graduate student in aeronautical engineering used the 7090 to simulate the wings he was designing for short takeoffs and landings. He knew such a wing would be difficult to control if its characteristics included an abrupt onset of stall, but he thought he could avoid that. His simulations were telling him otherwise. Just to be sure that roundoff was not interfering, he had repeated many of his calculations in double precision and gotten results much like those in single; his wings had stalled abruptly in both precisions. Disheartened, the student gave up.

Meanwhile Kahan replaced IBM's logarithm program (ALOG) with one of his own, which he hoped would provide better accuracy. While testing it, Kahan reran programs using the new version of ALOG. The student's results changed significantly; Kahan approached him to find out what had happened.

The student was puzzled. Much as the student preferred the results produced with the new ALOG—they predicted a gradual stall—he knew they must be wrong because they disagreed with his double precision results. The discrepancy between single and double precision results disappeared a few days later when a new release of IBM's double precision arithmetic software for the 7090 arrived. (The 7090 had no double precision hardware.) He went on to write a thesis about it and to build the wings; they performed as predicted. But that is not the end of the story.

In 1963, the 7090 was replaced by a faster 7094 with double precision floating-point hardware but with otherwise practically the same instruction set as the 7090. Only in double precision and only when using the new hardware did the wing stall abruptly again. A lot of time was spent to find out why. The 7094 hardware turned out, like the superseded 7090 software and the subsequent early S/360s, to lack a guard bit in double precision. Like so many programmers on those computers and on Cray's, the student discovered a trick to compensate for the lack of a guard digit; he wrote the expression `(0.5 - x) + 0.5` in place of `1.0 - x`. Nowadays we would blush if we had to explain why such a trick might be necessary, but it solved the student's problem.

Meanwhile the lure of California was working on Kahan and his family; they came to Berkeley and he to the University of California. An opportunity presented itself in 1974 when accuracy questions induced Hewlett-Packard's calculator designers to call in a consultant. The consultant was Kahan, and his work dramatically improved the accuracy of HP calculators, but that is another story. Fruitful collaboration with congenial coworkers, however, fortified him for the next and crucial opportunity.

It came in 1976, when John F. Palmer at Intel was empowered to specify the "best possible" floating-point arithmetic for all of Intel's product line, as Moore's Law made it now possible to create a whole floating point unit on a single chip. The floating-point standard was originally started for the iAPX-432, but when it was late, Intel started the 8086 as a short term emergency stand-in until the iAPX-432 was ready. The iAPX-432 never became popular, so the emergency stand-in became the standard-bearer for Intel. The 8087 floating-point coprocessor for the 8086 was contemplated. (A ***coprocessor*** is simply an additional chip that accelerates a portion of the work of a processor; in this case, it accelerated floating-point computation.)

Palmer had obtained his Ph.D. at Stanford a few years before and knew whom to call for counsel of perfection—Kahan. They put together a design that obviously would have been impossible only a few years earlier and looked not quite possible at the time. But a new Israeli team of Intel employees led by Rafi Navé felt challenged to prove their prowess to Americans and leaped at an opportunity to put something impossible on a chip—the 8087.

By now, floating-point arithmetics that had been merely diverse among mainframes had become chaotic among microprocessors, one of which might be host to a dozen varieties of arithmetic in ROM firmware or software. Robert G. Stewart, an engineer prominent in IEEE activities, got fed up with this anarchy and proposed that the IEEE draft a decent floating-point standard. Simultaneously, word leaked out in Silicon Valley that Intel was going to put on one chip some awesome floating point well beyond anything its competitors had in mind. The competition had to find a way to slow Intel down, so they formed a committee to do what Stewart requested.

Meetings of this committee began in late 1977 with a plethora of competing drafts from innumerable sources and dragged on into 1985, when IEEE Standard 754 for Binary Floating Point was made official. The winning draft was very close to one submitted by Kahan, his student Jerome T. Coonen, and Harold S. Stone, a professor visiting Berkeley at the time. Their draft was based on the Intel design, with Intel's permission, of course, as simplified by Coonen. Their harmonious combination of features, almost none of them new, had at the outset attracted more support within the committee and from outside experts like Wilkinson than any other draft, but they had to win nearly unanimous support within the committee to win official IEEE endorsement, and that took time.

PARTICIPATION ACTIVITY 3.11.3: A need to standardize floating-point.

1) A graduate student in aeronautical engineering used the IBM 7090 to simulate a new _____. After many simulations, the student was disheartened because the simulation predicted an abrupt onset of stall.

○ wing design

○ logarithm program

2) The IEEE formed a committee in _____ to begin drafting a floating-point standard.

○ 1976

○ 1977

○ 1985

The first IEEE 754 chips

In 1980, Intel became tired of waiting and released the 8087 for use in the IBM PC. The floating-point architecture of the companion 8087 had to be retrofitted into the 8086 opcode space, making it inconvenient to offer two operands per instruction as found in the rest of the 8086. Hence the decision for one operand per instruction using a stack: "The designer's task was to make a Virtue of this Necessity." (Kahan's [1990] history of the stack architecture selection for the 8087 is entertaining reading.)

Rather than the classical stack architecture, which has no provision for avoiding common subexpressions from being pushed and popped from memory into the top of the stack found in registers, Intel tried to combine a flat register file with a stack. The reasoning was that the restriction of the top of stack as one operand was not so bad since it only required the execution of an `FXCH` instruction (which swapped registers) to get the same result as a two-operand instruction, and `FXCH` was much faster than the floating-point operations of the 8087.

Since floating-point expressions are not that complex, Kahan reasoned that eight registers meant that the stack would rarely overflow. Hence, he urged that the 8087 use this hybrid scheme with the provision that stack overflow or stack underflow would interrupt the 8086 so that interrupt software could give the illusion to the compiler writer of an unlimited stack for floating-point data.

The Intel 8087 was implemented in Israel, and 7500 miles and 10 time zones made communication from California difficult. According to Palmer and Morse (*The 8087 Primer*, J. Wiley, New York, 1984, p. 93):

> “ Unfortunately, nobody tried to write a software stack manager until after the 8087 was built, and by then it was too late; what was too complicated to perform in hardware turned out to be even worse in software. One thing found lacking is the ability to conveniently determine if an invalid operation is indeed due to a stack overflow.… Also lacking is the ability to restart the instruction that caused the stack overflow …

The result is that the stack exceptions are too slow to handle in software. As Kahan [1990] says:

> “ Consequently, almost all higher-level languages' compilers emit inefficient code for the 80x87 family, degrading the chip's performance by typically 50% with spurious stores and loads necessary simply to preclude stack over/under-flow.…
> I still regret that the 8087's stack implementation was not quite so neat as my original intention.… If the original design had been realized, compilers today would use the 80x87 and its descendents more efficiently, and Intel's competitors could more easily market faster but compatible 80x87 imitations.

In 1982, Motorola announced its 68881, which found a place in Sun 3s and Macintosh IIs; Apple had been a supporter of the proposal from the beginning. Another Berkeley graduate student, George S. Taylor, had soon designed a high-speed implementation of the proposed standard for an early superminicomputer (ELXSI 6400). The standard was becoming de facto before its final draft's ink was dry.

An early rush of adoptions gave the computing industry the false impression that IEEE 754, like so many other standards, could be implemented easily by following a standard recipe. Not true. Only the enthusiasm and ingenuity of its early implementors made it look easy.

In fact, to implement IEEE 754 correctly demands extraordinarily diligent attention to detail; to make it run fast demands extraordinarily competent ingenuity of design. Had the industry's engineering managers realized this, they might not have been so quick to affirm that, as a matter of policy, "We conform to all applicable standards."

IEEE 754 today

Unfortunately, the compiler-writing community was not represented adequately in the wrangling, and some of the features didn't balance language and compiler issues against other points. That community has been slow to make IEEE 754's unusual features available to the applications programmer. Humane exception handling is one such unusual feature; directed rounding another. Without compiler support, these features have atrophied.

The successful parts of IEEE 754 are that it is a widely implemented standard with a common floating-point format, that it requires minimum accuracy to one-half ulp in the least significant bit, and that operations must be commutative.

The IEEE 754/854 have been implemented to a considerable degree of fidelity in at least part of the product line of every North American computer manufacturer. The only significant exceptions were the DEC VAX, IBM S/370 descendants, and Cray Research vector supercomputers, and all three have been replaced by compliant computers.

In 1989, the Association for Computing Machinery, acknowledging the benefits conferred upon the computing industry by IEEE 754, honored Kahan with the Turing Award. On accepting it, he thanked his many associates for their diligent support, and his adversaries for their blunders. So … not all errors are bad.

IEEE rules ask that a standard be revisited periodically for updating. A committee started in 2000, and drafts of the revised standards were circulated for voting, and these were approved in 2008. The revised standard, IEEE Std 754-2008 [2008], includes several new types: 16-bit floating point, called *half precision*; 128-bit floating point, called *quad precision*; and three decimal types, matching the length of the 32-bit, 64-bit, and 128-bit binary formats. IEEE Std 754-2019 made minor changes to the standard. The plan is to revisit it every 10 years.

PARTICIPATION ACTIVITY 3.11.4: IEEE 754 binary floating-point.

1) The release of the IEEE 754 standard made the implementation of floating point hardware simple for manufacturers.

○ True

○ False

2) Kahan was honored with the Turing Award in 1989 for the benefits conferred upon the computing industry through standardizing floating point.

○ True

○ False

3) IEEE 754, released in 1985, is the most recent floating point standard.

○ True

○ False

Further reading

If you are interested in learning more about floating point, two publications by David Goldberg [1991, 2002] are good starting points; they abound with pointers to further reading. Several of the stories told on the CD come from Kahan [1972, 1983]. The latest word on the state of the art in computer arithmetic is often found in the *Proceedings* of the latest IEEE-sponsored Symposium on Computer Arithmetic, held every two years; the 16th was held in 2003.

Burks, A. W., H. H. Goldstine, and J. von Neumann [1946]. "Preliminary discussion of the logical design of an electronic computing instrument," *Report to the U.S. Army Ordnance Dept.*, p. 1; also in *Papers of John von Neumann*, W. Aspray and A. Burks (Eds.), MIT Press, Cambridge, MA, and Tomash Publishers, Los Angeles, 1987, 97—146.
This classic paper includes arguments against floating-point hardware.

Goldberg, D. [2002]. "Computer arithmetic," Appendix A of *Computer Architecture: A Quantitative Approach*, third edition, J. L. Hennessy and D. A. Patterson, Morgan Kaufmann Publishers, San Francisco.
A more advanced introduction to integer and floating-point arithmetic, with emphasis on hardware. It covers Sections 3.4—3.6 of this book in just 10 pages, leaving another 45 pages for advanced topics.

Goldberg, D. [1991]. "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys* 23(1) 5—48.
Another good introduction to floating-point arithmetic by the same author, this time with emphasis on software.

Kahan, W. [1972]. "A survey of error-analysis," in *Info. Processing* 71 (Proc. IFIP Congress 71 in Ljubljana), Vol. 2, North-Holland Publishing, Amsterdam, 1214—1239.
This survey is a source of stories on the importance of accurate arithmetic.

Kahan, W. [1983]. "Mathematics written in sand," *Proc. Amer. Stat. Assoc. Joint Summer Meetings of 1983, Statistical Computing Section*, 12—26.
The title refers to silicon and is another source of stories illustrating the importance of accurate arithmetic.

Kahan, W. [1990]. "On the advantage of the 8087's stack," unpublished course notes, Computer Science Division, University of California, Berkeley.
What the 8087 floating-point architecture could have been.

Kahan, W. [1997]. Available via a link to Kahan's homepage at *www.mkp.com/books_catalog/ cod/links.htm*.
A collection of memos related to floating point, including "Beastly numbers" (another less famous Pentium bug), "Notes on the IEEE floating point arithmetic" (including comments on how some features are atrophying), and "The baleful effects of computing benchmarks" (on the unhealthy preoccupation on speed versus correctness, accuracy, ease of use, flexibility, ...).

Koren, I. [2002]. *Computer Arithmetic Algorithms*, second edition, A. K. Peters, Natick, MA.
A textbook aimed at seniors and first-year graduate students that explains fundamental principles of basic arithmetic, as well as complex operations such as logarithmic and trigonometric functions.

Wilkes, M. V. [1985]. *Memoirs of a Computer Pioneer*, MIT Press, Cambridge, MA.
This computer pioneer's recollections include the derivation of the standard hardware for multiply and divide developed by von Neumann.

3.12 Self study

Data can be anything. In the Self Study section at the end of COD Chapter 2, we saw the binary bit pattern $\mathbf{00000001010010110100100000100000_{two}}$ in hexadecimal, decimal, and as a MIPS assembly language instruction. What IEEE 754 Floating Point number does it represent?

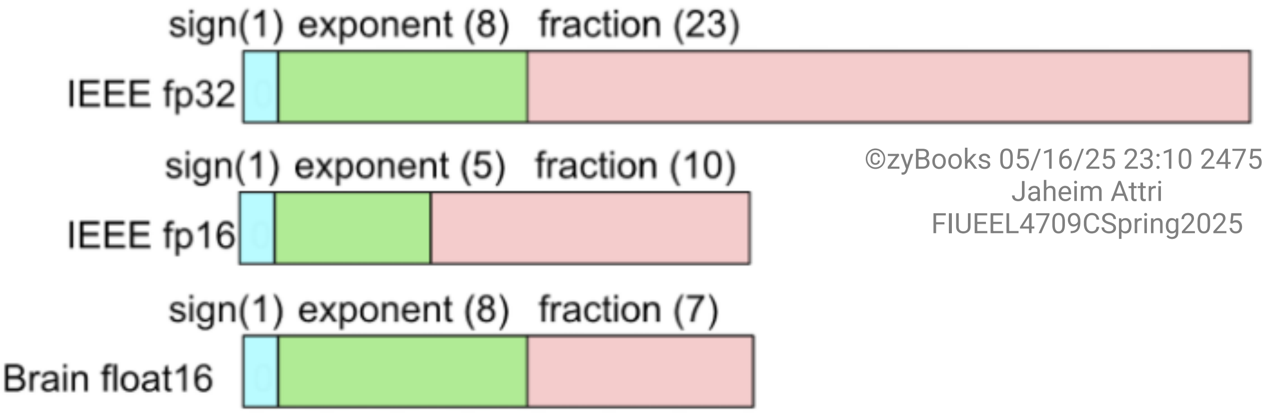
Big numbers. What is the largest, positive two's complement 32-bit integer? Can you represent it exactly in IEEE 754 single precision floating point? If not, how close can you get? What about IEEE 754 half precision floating point?

Brainy Arithmetic. Machine learning is starting to work so well that it is revolutionizing many industries (See COD Section 6.6.5 of COD Chapter 6). It uses floating point numbers to learn, but unlike scientific programming, it doesn't need lots of precision. Double precision floating point, the standard bearer of scientific programming, is overkill as 32 bits is sufficient. Ideally it could use half-precision (16 bits), since that is much more efficient in computation and memory. However, machine learning training often deals with very small numbers, so the range is important.

These observations about the needs of machine learning led to a new format that is not part of the IEEE standard called Brain Float 16 (named after Google's Brain division, which invented the format). The figure below shows the three formats.

Figure 3.12.1: Floating point format for IEEE 754 single precision (fp32), IEEE 754 half precision (fp16), and Brain float 16. Google's TPUv3 hardware uses Brain float 16 (see COD Section 6.11) (COD Figure 3.27).

Assume Brain float16 follows the same conventions as IEEE 754, just with different sizes of fields. What is the smallest non-zero, positive number that you can represent in the three formats? How much smaller is that number for Brain float 16 than for IEEE fp32? Than fp16? (If you know about subnormals or denorms, ignore them for this question.)


sign(1) exponent (8) fraction (23)
IEEE fp32
sign(1) exponent (5) fraction (10)
IEEE fp16
sign(1) exponent (8) fraction (7)
Brain float16

Brainy Area and Energy. A common operation in machine learning is multiply and accumulate like we see in DGEMM, with the multiply occupying most of the silicon area and using most of the energy. If we have fast multipliers like in COD Figure 3.7, they are primarily a function of the square of size of the inputs. What are the correct ratios of area/energy of the three formats for multiplies?

1. 32^2 vs. 16^2 vs. 16^2 for fp32, fp16, and Brain float, respectively
2. 8^2 vs. 5^2 vs. 8^2
3. 23^2 vs. 10^2 vs. 7^2
4. 24^2 vs. 11^2 vs. 8^2

Brainy Programming. Can you think of software benefits of IEEE fp32 and Brain float16 having the same sized exponents?

Brainy Choices. For the domain for machine learning, which of the following are true about Brain float 16 arithmetic versus IEEE 754 half precision floating point?

1. Brain float 16 multipliers take much less hardware than IEEE 754 half precision.
2. Brain float 16 multiplies take much less energy than IEEE 754 half precision.
3. Brain float 16 is easier for software than IEEE 754 half precision when converting from IEEE 754 full precision software.
4. All of the above.

Self Study Answers

Data can be anything. Mapping the binary number into the IEEE 754 floating-point format:

Table 3.12.1

Sign (1)	Exponent (8)	Fraction (23)
0	00000010_{two}	$10010110100100000100000_{two}$
+	2_{ten}	$4,933,66_{ten}$

Since the exponent bias for single precision floating point is 127, the exponent is actually 2-127 or -125. The fraction can be thought of as $4,933,664_{ten} /(2^{23}-1) = 4,933,664_{ten} /8,388,607_{ten} = 0.58813865043_{ten}$. The actual significand adds the implicit 1, so the real number the binary pattern represents is $1.58813865043_{ten} \text{ x } 2^{-125}$ or about $3.7336959_{ten} \text{ x } 10^{-38}$.

Once again, this exercise demonstrates that there is no inherent meaning in a bit pattern, it depends solely on how software interprets it.

Big numbers. The largest, positive two's complement 32-bit integer is 2^{31} - 1 = 2,147,483,647 You cannot represent it exactly in IEEE 754 single precision floating point.

Table 3.12.2

Sign (1)	Exponent (8)	Fraction (23)
0	00000010_{two}	$00000000000000000000000_{two}$
+	158_{ten}	0_{ten}

=1.0 *$2^{(158-127)}$=1.0 *2^{31}= 2,147,483,648, so off by 1 from 2^{31} - 1.

The largest number you can represent in IEEE 754 half precision is

Table 3.12.3

Sign (1)	Exponent (5)	Fraction (10)
0	11110_{two}	1111111111_{two}
+	30_{ten}	1023_{ten}

= (1 + 1023/1024) *$2^{(30-15)}$ = 1.999 * 2^{15} = 65,504, so it's off by many orders of magnitude. Converting integers to IEEE Half precision float can cause overflow.(The 5-bit exponent 11111_{two} is reserved for infinites and NaNs in half precision, like the exponent 11111111_{two} is reserved for single precision.)

Brainy Arithmetic. Smallest non-zero positive numbers per format:

- IEEE fp32 1.0 * 2^{-126}
- IEEE fp16 1.0 * 2^{-14}
- Brain float16 1.0 * 2^{-126}

Since IEEE fp32 and Brian float 16 have the same sized exponents, the can represent the same smallest non-zero positive number. The smallest number they can represent is 2112 times smaller than that of IEEE fp16, or about 5 x10^{33}.

Brainy Area and Energy. The exponent and sign fields are not involved in the multiplications, so the answer is a function of the size of significands. As there is an implicit 1 and followed by the fraction in these formats, the correct answer is number 4: 24^2 vs. 11^2 vs. 8^2. That makes IEEE fp16 multiplier about twice (121/64) the size or energy of Brain float16 and IEEE fp32 about nine (576/64) times larger.

Brainy Programming. Since the exponents are the same, that means software will have the same behavior for underflows and overflows, Not a Numbers (NaNs), infinities, and so on, which means software using brain float16 to replace IEEE fp32 in some calculations will likely have fewer compatibility problems than switching to IEEE fp16.

Brainy Choices. The answer is 4, all of the above. Remarkably, for machine learning applications, Brain float 16 is both easier for hardware designers and for software programmers. Not surprisingly, Brain float 16 is very popular for machine learning, and Googles TPUv2 and TPUv3 were the first processors to implement it (See Section 6.11)

3.13 Exercises

> “ Never give in, never give in, never, never, never—in nothing, great or small, large or petty—never give in.
> *Winston Churchill, address at Harrow School, 1941*

Because this interactive zyBook version may have been re-ordered and hence sections renumbered, section numbers below labeled with COD refer to the original hardcopy book's section numbers.

EXERCISE 3.13.1: [5] <COD §3.2>.

(a) What is 5ED4 - 07A4 when these values represent unsigned 16-bit hexadecimal numbers? The result should be written in hexadecimal. Show your work.

Solution

EXERCISE 3.13.2: [5] <COD §3.2>.

(a) What is 5ED4 — 07A4 when these values represent signed 16-bit hexadecimal numbers stored in sign-magnitude format? The result should be written in hexadecimal. Show your work.

Solution ⌄

EXERCISE 3.13.3: [10] <COD §3.2>. ?

(a) Convert 5ED4 into a binary number. What makes base 16 (hexadecimal) an attractive numbering system for representing values in computers?

Solution ⌄

EXERCISE 3.13.4: [5] <COD §3.2>. ?

(a) What is 4365 — 3412 when these values represent unsigned 12-bit octal numbers? The result should be written in octal. Show your work.

Solution ⌄

EXERCISE 3.13.5: [5] <COD §3.2>. ?

(a) What is 4365 — 3412 when these values represent signed 12-bit octal numbers stored in sign-magnitude format? The result should be written in octal. Show your work.

Solution ⌄

EXERCISE 3.13.6: [5] <COD §3.2>. ?

(a) Assume 185 (1011 1001) and 122 (0111 1010) are unsigned 8-bit decimal integers. Calculate 185 — 122. Is there overflow, underflow, or neither?

Solution ⌄

EXERCISE 3.13.7: [5] <COD §3.2>.

(a) Assume 185 and 122 are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 + 122. Is there overflow, underflow, or neither?

Solution

EXERCISE 3.13.8: [5] <COD §3.2>.

(a) Assume 185 (1011 1001) and 122 (0111 1010) are signed 8-bit decimal integers stored in sign-magnitude format. Calculate 185 — 122. Is there overflow, underflow, or neither?

Solution

EXERCISE 3.13.9: [10] <COD §3.2>.

(a) Assume 151 (1001 0111) and 214 (1101 0110) are signed 8-bit decimal integers stored in two's complement format. Calculate 151 + 214 using saturating arithmetic. The result should be written in decimal. Show your work.

Solution

EXERCISE 3.13.10: [10] <COD §3.2>.

(a) Assume 151 (1001 0111) and 214 (1101 0110) are signed 8-bit decimal integers stored in two's complement format. Calculate 151 — 214 using saturating arithmetic. The result should be written in decimal. Show your work.

Solution

EXERCISE 3.13.11: [10] <COD §3.2>.

(a) Assume 151 (1001 0111) and 214 (1101 0110) are unsigned 8-bit integers. Calculate 151 + 214 using saturating arithmetic. The result should be written in decimal. Show your work.

Solution

EXERCISE 3.13.12: [20] <COD §3.3>.

(a) Using a table similar to that shown in COD Figure 3.6 (Multiply example using algorithm in COD Figure 3.4), calculate the product of the octal unsigned 6-bit integers 62 and 12 using the hardware described in COD Figure 3.3 (First version of the multiplication hardware). You should show the contents of each register on each step.

Solution

EXERCISE 3.13.13: [20] <COD §3.3>.

(a) Using a table similar to that shown in COD Figure 3.6 (Multiply example using algorithm in COD Figure 3.4), calculate the product of the octal unsigned 8-bit integers 62 and 12 using the hardware described in COD Figure 3.5 (Refined version of the multiplication hardware). You should show the contents of each register on each step.

EXERCISE 3.13.14: [10] <COD §3.3>.

(a) Calculate the time necessary to perform a multiply using the approach given in COD Figure 3.3 (First version of the multiplication hardware) and 3.4 (The first multiplication algorithm, using the hardware shown in COD Figure 3.3) if an integer is 8 bits wide and each step of the operation takes 4 time units. Assume that in step 1a an addition is always performed—either the multiplicand will be added, or a zero will be. Also assume that the registers have already been initialized (you are just counting how long it takes to do the multiplication loop itself). If this is being done in hardware, the shifts of the multiplicand and multiplier can be done simultaneously. If this is being done in software, they will have to be done one after the other. Solve for each case.

EXERCISE 3.13.15: [10] <COD §3.3>.

(a) Calculate the time necessary to perform a multiply using the approach described in the text (31 adders stacked vertically) if an integer is 8 bits wide and an adder takes 4 time units.

EXERCISE 3.13.16: [20] <COD §3.3>.

(a) Calculate the time necessary to perform a multiply using the approach given in COD Figure 3.7 (Fast multiplication hardware) if an integer is 8 bits wide and an adder takes 4 time units.

EXERCISE 3.13.17: [20] <COD §3.3>.

(a) As discussed in the text, one possible performance enhancement is to do a shift and add instead of an actual multiplication. Since 9×6, for example, can be written $(2 \times 2 \times 2 + 1) \times 6$, we can calculate 9×6 by shifting 6 to the left 3 times and then adding 6 to that result. Show the best way to calculate 0x33 × 0x55 using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

EXERCISE 3.13.18: [20] <COD §3.4>.

(a) Using a table similar to that shown in COD Figure 3.10 (Division example using the algorithm in COD Figure 3.9), calculate 60 divided by 17 using the hardware described in COD Figure 3.8 (First version of the division hardware). You should show the contents of each register on each step. Assume both inputs are unsigned 6-bit integers.

EXERCISE 3.13.19: [30] <COD §3.4>.

(a) Using a table similar to that shown in COD Figure 3.10 (Division example using the algorithm in COD Figure 3.9), calculate 60 divided by 17 using the hardware described in COD Figure 3.11 (An improved version of the division hardware). You should show the contents of each register on each step. Assume A and B are unsigned 6-bit integers. This algorithm requires a slightly different approach than that shown in COD Figure 3.9 (A division algorithm, using the hardware in COD Figure 3.8). You will want to think hard about this, do an experiment or two, or else go to the web to figure out how to make this work correctly. (Hint: one possible solution involves using the fact that COD Figure 3.11 (An improved version of the division hardware) implies the remainder register can be shifted either direction.)

EXERCISE 3.13.20: [5] <COD §3.5>.

(a) What decimal number does the bit pattern `0×0C000000` represent if it is a two's complement integer? An unsigned integer?

Solution

EXERCISE 3.13.21: [10] <COD §3.5>.

(a) If the bit pattern `0×0C000000` is placed into the Instruction Register, what MIPS instruction will be executed?

Solution

EXERCISE 3.13.22: [10] <COD §3.5>.

(a) What decimal number does the bit pattern `0×0C000000` represent if it is a floating point number? Use the IEEE 754 standard

Solution

EXERCISE 3.13.23: [10] <COD §3.5>.

(a) Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 single precision format.

Solution

EXERCISE 3.13.24: [10] <COD §3.5>.

(a) Write down the binary representation of the decimal number 63.25 assuming the IEEE 754 double precision format.

Solution

EXERCISE 3.13.25: [10] <COD §3.5>.

(a) Write down the binary representation of the decimal number 63.25 assuming it was stored using the single precision IBM format (base 16, instead of base 2, with 7 bits of exponent).

EXERCISE 3.13.26: [20] <COD §3.5>.

(a) Write down the binary bit pattern to represent -1.5625×10^{-1} assuming a format similar to that employed by the DEC PDP-8 (the leftmost 12 bits are the exponent stored as a two's complement number, and the rightmost 24 bits are the fraction stored as a two's complement number). No hidden 1 is used. Comment on how the range and accuracy of this 36-bit pattern compares to the single and double precision IEEE 754 standards.

EXERCISE 3.13.27: [20] <COD §3.5>.

(a) IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed. Write down the bit pattern to represent -1.5625×10^{-1} assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

EXERCISE 3.13.28: [20] <COD §3.5>.

(a) The Hewlett-Packard 2114, 2115, and 2116 used a format with the leftmost 16 bits being the fraction stored in two's complement format, followed by another 16-bit field which had the leftmost 8 bits as an extension of the fraction (making the fraction 24 bits long), and the rightmost 8 bits representing the exponent. However, in an interesting twist, the exponent was stored in sign-magnitude format with the sign bit on the far right! Write down the bit pattern to represent -1.5625×10^{-1} assuming this format. No hidden 1 is used. Comment on how the range and accuracy of this 32-bit pattern compares to the single precision IEEE 754 standard.

EXERCISE 3.13.29: [20] <COD §3.5>.

(a) Calculate the sum of 2.6125×10^1 and $4.150390625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision described in COD Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps.

EXERCISE 3.13.30: [30] <COD §3.5>.

(a) Calculate the product of -8.0546875×10^0 and $-1.79931640625 \times 10^{-1}$ by hand, assuming A and B are stored in the 16-bit half precision format described in COD Exercise 3.27. Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps; however, as is done in the example in the text, you can do the multiplication in human-readable format instead of using the techniques described in COD Exercises 3.12 through 3.14. Indicate if there is overflow or underflow. Write your answer in both the 16-bit floating point format described in COD Exercise 3.27 and also as a decimal number. How accurate is your result? How does it compare to the number you get if you do the multiplication on a calculator?

EXERCISE 3.13.31: [30] <COD §3.5>.

(a) Calculate by hand 8.625×10^1 divided by -4.875×10^0. Show all the steps necessary to achieve your answer. Assume there is a guard, a round bit, and a sticky bit, and use them if necessary. Write the final answer in both the 16-bit floating point format described in COD Exercises 3.27 and in decimal and compare the decimal result to that which you get if you use a calculator.

EXERCISE 3.13.32: [20] <COD §3.10>.

(a) <COD §3.9> Calculate $\left(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}\right) + 1.771 \times 10^{3}$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.33: [20] <COD §3.10>.

(a) Calculate $3.984375 \times 10^{-1} + \left(3.4375 \times 10^{-1} + 1.771 \times 10^{3}\right)$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.34: [10] <COD §3.10>.

(a) Based on your answers to COD Exercises 3.32 and 3.33, does $\left(3.984375 \times 10^{-1} + 3.4375 \times 10^{-1}\right) + 1.771 \times 10^{3} = 3.984375 \times 10^{-1} + (3.437$
?

EXERCISE 3.13.35: [30] <COD §3.10>.

(a) Calculate $\left(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}\right) \times 1.05625 \times 10^{2}$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.36: [30] <COD §3.10>.

(a) Calculate $\mathbf{3.41796875 \times 10^{-3} \times \left(6.34765625 \times 10^{-3} \times 1.05625 \times 10^{2}\right)}$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.37: [10] <COD §3.10>.

(a) Based on your answers to COD Exercises 3.35 and 3.36, does $\mathbf{\left(3.41796875 \times 10^{-3} \times 6.34765625 \times 10^{-3}\right) \times 1.05625 \times 10^{2} = 3.41796875 \times 1}$?

EXERCISE 3.13.38: [30] <COD §3.10>.

(a) Calculate $\mathbf{1.666015625 \times 10^{0} \times \left(1.9760 \times 10^{4} + -1.9744 \times 10^{4}\right)}$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercise 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.39: [30] <COD §3.10>.

(a) Calculate $\mathbf{\left(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}\right) + \left(1.666015625 \times 10^{0} \times -1.9744 \times 10^{4}\right)}$ by hand, assuming each of the values are stored in the 16-bit half precision format described in COD Exercises 3.27 (and also described in the text). Assume 1 guard, 1 round bit, and 1 sticky bit, and round to the nearest even. Show all the steps, and write your answer in both the 16-bit floating point format and in decimal.

EXERCISE 3.13.40: [10] <COD §3.10>.

(a) Based on your answers to COD Exercises 3.38 and 3.39, does $\mathbf{\left(1.666015625 \times 10^{0} \times 1.9760 \times 10^{4}\right) + \left(1.666015625 \times 10^{0} \times -1.9744 \times 10^{4}\right)}$

?

EXERCISE 3.13.41: [10] <COD §3.5>.

(a) Using the IEEE 754 floating point format, write down the bit pattern that would represent $-1/4$. Can you represent $-1/4$ exactly?

Solution

EXERCISE 3.13.42: [10] <COD §3.5>.

(a) What do you get if you add -1/4 to itself 4 times? What is -1/4 × 4? Are they the same? What should they be?

EXERCISE 3.13.43: [10] <COD §3.5>.

(a) Write down the bit pattern in the fraction of value $1/3$ assuming a floating point format that uses binary numbers in the fraction. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

Solution

EXERCISE 3.13.44: [10] <COD §3.5>.

(a) Write down the bit pattern in the fraction assuming a floating point format that uses Binary Coded Decimal (base 10) numbers in the fraction instead of base 2. Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

EXERCISE 3.13.45: [10] <COD §3.5>.

(a) Write down the bit pattern assuming that we are using base 15 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0—9 and A—F. Base 15 numbers would use 0—9 and A—E.) Assume there are 24 bits, and you do not need to normalize. Is this representation exact?

EXERCISE 3.13.46: [20] <COD §3.5>.

(a) Write down the bit pattern assuming that we are using base 30 numbers in the fraction instead of base 2. (Base 16 numbers use the symbols 0—9 and A—F. Base 30 numbers would use 0—9 and A—T.) Assume there are 20 bits, and you do not need to normalize. Is this representation exact?

EXERCISE 3.13.47: [45] <COD §§3.6, 3.7>.

(a) The following C code implements a four-tap FIR filter on input array sig_in. Assume that all arrays are 16-bit fixed-point values.

```
for(i = 3; i < 128; i++)
   sig_out[i] = sig_in[i - 3] * f[0] + sig_in[i - 2] * f[1] +
                sig_in[i - 1] * f[2] + sig_in[i] * f[3];
```

Assume you are to write an optimized implementation of this code in assembly language on a processor that has SIMD instructions and 128-bit registers. Without knowing the details of the instruction set, briefly describe how you would implement this code, maximizing the use of sub-word operations and minimizing the amount of data that is transferred between registers and memory. State all your assumptions about the instructions you use.