Arithmetic for Computers

Ming-Hwa Wang, Ph.D.
COEN 210 Computer Architecture
Department of Computer Engineering
Santa Clara University

Signed and Unsigned Numbers

- computers perform operations on numbers whose precision is finite and fixed, the finite-precision numbers; not close under operations and not follow algebra rules (i.e., the order of operations is important); may cause overflow, underflow, or error
- radix number systems or weighted number systems
  - the radix is the base for exponentiation; a radix *k* number system requires *k* different symbols to represent the digits 0 to *k* - 1
  - in any number base, the value of i^{th} digit d_i is d_i * base^i, where i starts at 0 and increases from right to left, the least significant bit is the rightmost one, and most significant bit is the leftmost; the value of n-digit number = $\sum_{i=0}^{n-1} d_i$ * base^i
  - humans prefer base 10, computers use base 2; the first commercial computer did offer decimal (represented by several binary digits) arithmetic, which is very inefficient; now computers are all binary, and only convert to decimal for input/output
  - if we represent numbers as string of ASCII digits, waste space
  - the MIPS words is 32 bits long, it can represent 2^{32} different 32-bit patterns form 0 to 2^{32} –1 (4,294,967,295)
  - save reading and writing long binary numbers by using a higher base than binary; since almost all computer data sizes are multiple of 4, hexadecimal (base 16) are popular
- number systems conversions
  - binary to decimal conversion, decimal to binary conversion
  - binary to octal, octal to binary, binary to hexadecimal, hexadecimal to binary: by replacing each group of 3/4 binary digits by a single octal/hexadecimal digit, and vice versa
  - decimal to octal, octal to decimal, decimal to hexadecimal, hexadecimal to decimal: by converting to binary first
- negative numbers
  - sign and magnitude: sign bit and magnitude bits (has both positive and negative 0s, not arithmetic)
  - one's complement: invert every 0 to 1 and every 1 to 0 (leading 0s mean positive and leading 1s mean negative; hardware tests the most significant bit for sign, same problems as sign and magnitude)
  - two's complement: one's complement plus 1, since $x + \sim x \equiv -1$, therefore, $\sim x + 1 = -x$ (different singularity or unsymmetric: positive from 0 to $2^{31} - 1$ or 2,147,483,647, negative from -2^{31} to –1, and the value of a number x is $\sum_{i=0}^{30} x_i * 2^i + x_{31} * -2^{31}$, where x_i means the i^{th} bit of x); the name comes from the negative of n-bit value x is $2^n - x$
  - biased notation or excess 2^{m-1} for floating point's exponent: 00..000 for most negative value, 11…111 for most positive number, and 10..000 for zero or bias (the number plus the bias is nonnegative); this system is identical to two's complement with the sign bit reversed
- sign extension: the function of a signed load is to copy the sign repeatedly to fill the rest of the register (unsigned load simply fill with 0s to the left of the data), when loading a 32-bit word into a 32-bit register, signed and unsigned load are identical
- Memory addresses are unsigned from 0 to the largest address
- High-level languages have integer and unsigned integer types
- MIPS offers **lb** (load byte), **lbu** (load byte unsigned), slt (set on less than), slti (set on less than immediate), sltu (set on less than unsigned), sltiu (set on less than immediate unsigned)
- if a number cannot be represented by those bits (for singed numbers, when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits to the left), overflow occurs, and it's up to the OS and program to determine what to do
- computer arithmetic has become largely standardized, greatly enhancing the portability of programs

Addition and Subtraction

- digits are added bit by bit from right to left, with carries passed to the next digit to the left; subtraction uses addition, just negate before add
- when adding 2 operands with different signs or subtracting 2 operands with same signs, overflow cannot occur; when adding 2 positive numbers and the sum is negative, or when subtracting a negative/positive number from a positive/negative number and get a positive/negative result, overflow occurs
- a simple check for overflow during addition is to see if the *CarryIn* to the most significant bit is not the same as the *CarryOut*
- unsigned integers are commonly used for memory addresses where overflows are ignored
- the MIPS has **add**, **addi** and **sub** instructions which cause exceptions; has **addu**, **addiu** and **subu** instructions which do not cause exceptions on overflow
- because C ignores overflow, the MIPS C compiler always generate the unsigned version of the arithmetic instructions; Ada and Fortran require the programmer or the programming environment to decide what to do when overflow occurs
- the MIPS detects overflow with an exception/interrupt (an unscheduled procedure call); the exception program counter (EPC) to contain the address of the instruction that cause the exception; the **mfc0** (move from system control) instruction is used to copy EPC into a general-purpose register

- the MIPS reserves registers $k0 and $k1 for the OS, and these registers are not restored on exceptions; exception routines place the return address in one of these registers and then use **jr** to restore the instruction address

Logic Operations

- logic instructions operate on fields of bits within a word or even on individual bits
- shift instructions move all the bits in a word to the left or right (with shift amount in the *shamt* field in the R-format), filling the emptied bits with 0s: **sll** (shift left logic), **srl** (sift right logic)
- bit-wise/bit-by-bit operations
  - **and** and **andi** (and immediate) instructions leave a 1 in the result only if both bits of the operands are 1; it can be used to apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern (a mask to conceal bits)
  - **or** and **ori** (or immediate) instructions place a 1 in the result if either operand bit is a 1
  - **not** (inversion or one's complement)
  - **nor** (not OR) and **xor** (exclusive OR)
- C allows right-aligned bit fields or fields (unsigned integers) to be defined within words and all fields must fit or packed within a single word; C compilers insert and extract fields using logical operations
  - a shift left of 32 – (*n* + *m*) followed by a shift right by 32 – *n* will isolate any *n*-bit field whose least significant bit is in bit *m*
  - since **addi** and **slti** are intended for signed numbers, their immediate fields are sign-extended before use
  - **addiu** and **sltiu** also sign-extend their immediates
  - **andi** and **ori** treats their immediates as unsigned integers

Constructing an Arithmetic Logic Unit

- the 4 hardware building blocks:
  1. AND gate: $c = a \bullet b$
  2. OR gate: $c = a + b$
  3. Inverter: $c = \sim c$
  4. multiplexor: if $s == 0$, $c = a$; else $c = b$
- an exclusive OR gate is true if the two operands disagree; in some technologies, exclusive OR is more efficient than two levels of AND and OR gates: $Sum = a \oplus b \oplus CarryIn$
- computers are designed today in CMOS transistors, which are basically switches
- the arithmetic logic unit (ALU) performs the arithmetic or logic operations, a 32-bit ALU is constructed by connecting 32 1-bit ALUs
  - a 1-bit full adder or a (3,2) adder: 2 inputs for the operands and 1 output for the *Sum*, a second output *CarryOut*, and a third input *CarryIn* (a half adder or a (2,2) adder without the *CarryIn*)
    - since we know what addition is supposed to do, we can specify the outputs of this "black box" based on its inputs
    - $CarryOut = (b \bullet CarryIn) + (a \bullet CarryIn) + (a \bullet b) + (a \bullet b \bullet CarryIn) = (b \bullet CarryIn) + (a \bullet CarryIn) + (a \bullet b)$
    - $Sum = (a \bullet \sim b \bullet \sim CarryIn) + (\sim a \bullet b \bullet \sim CarryIn) + (\sim a \bullet \sim b \bullet CarryIn) + (a \bullet b \bullet CarryIn)$
  - a 32-bit ripple carry adder is created by directly linking the carries of the 32 1-bit adders: a single carry out of the least significant bit ($Result_0$) can ripple all the way through the adder, causing a carry out of the most significant bit ($Result_{31}$)
  - a 1-bit ALU: merge adder, AND gate and OR gate; and use a multiplexor to choose the desired operation
  - a 32-bit ALU is created by connecting adjacent black boxes, using x_i to mean the i^{th} bit of x
  - the universal symbol for a complete ALU:
- subtraction is the same as adding the negative version of an operand by inverting (setting the *Binvert* to 1) each bit of the number and setting the least significant *CarryIn* to 1 (or combine *Binvert* and the least significant *CarryIn* to form *Bnagate*)
  - $a + \sim b + 1 = a + (\sim b + 1) = a - b$
- the **slt** command returns 1 if $a < b$ and returns 0 otherwise, this is done by connecting 0 to the *Less* input for the upper 31 bits of ALU, and connect the sign bit (which if 1 if negative) from the adder output to the least significant bit; thus we need a new 1-bit ALU for the most significant bit that has an extra output bit (the adder output *Set*)
- the **beq** command returns 1 if $a == b$ (i.e., $a - b = 0$) and returns 0 otherwise; $Zero = \sim(OR_{i=0}^{31} Result_i)$
- the sequential evaluation of all 32 1-bit adders is too slow to be used in time-critical hardware; unlike software, hardware executes in parallel whenever input change; thus we use carry lookahead by many more gates to anticipate the proper carry with O(lg *n*)
  - fast carry using infinite hardware (abbreviation of c_i for $CarryIn_i$)
    - $c_1 = (b_0 \bullet c_0) + (a_0 \bullet c_0) + (a_0 \bullet b_0)$
    - $c_2 = (b_1 \bullet c_1) + (a_1 \bullet c_1) + (a_1 \bullet b_1) = (b_1 \bullet a_0 \bullet b_0) + (b_1 \bullet a_0 \bullet c_0) + (b_1 \bullet b_0 \bullet c_0) + (a_1 \bullet a_0 \bullet b_0) + (a_1 \bullet a_0 \bullet c_0) + (a_1 \bullet b_0 \bullet c_0) + (a_1 \bullet b_1)$
    - $O(2^n)$, too expensive for wide adders
  - fast carry (carry-lookahead adder) using the first level of abstraction: propagate and generate
    - generate $g_i = a_i \bullet b_i$ and propagate $p_i = a_i + b_i$
    - $c_{i+1} = (b_i \bullet c_i) + (a_i \bullet c_i) + (a_i \bullet b_i) = (a_i \bullet b_i) + (a_i + b_i) \bullet c_i = g_i + p_i \bullet c_i$
    - the adder generates a $CarryOut(c_{i+1})$ independent of the value of $CarryIn(c_i)$: suppose $g_i = 1$, then $c_{i+1} = 1 + p_i \bullet c_i = 1$
    - the adder propagates *CarryIn* to a *CarryOut*: suppose $g_i = 0$ and $p_i = 1$, then $c_{i+1} = 0 + 1 \bullet c_i = c_i$

- a carry out can be make true by a generate far away provided all the propagates between them are true
- fast carry using the second level of abstraction
  - consider this 4-bit adder with its carry-lookahead logic as a single building block, and connect them in ripple carry fashion to form a 16-bit adder
    - the super propagate signal for the 4-bit abstraction (P_i) is true only if each of the bits in the group will propagate a carry: $P_0 = p_3 \bullet p_2 \bullet p_1 \bullet p_0$, $P_1 = p_7 \bullet p_6 \bullet p_5 \bullet p_4$, $P_2 = p_{11} \bullet p_{10} \bullet p_9 \bullet p_8$, $P_4 = p_{15} \bullet p_{14} \bullet p_{13} \bullet p_{12}$
    - the super generate signal (G_i) occurs if generate is true for that most significant bit, it also occurs if an earlier generate is true and all the intermediate propagate, including that of the most significant bit, are also true: if each of the bits in the group will propagate a carry; $G_0 = g_3 + (p_3 \bullet g_2) + (p_3 \bullet p_2 \bullet g_1) + (p_3 \bullet p_2 \bullet p_1 \bullet g_0)$, $G_1 = g_7 + (p_7 \bullet g_6) + (p_7 \bullet p_6 \bullet g_5) + (p_7 \bullet p_6 \bullet p_5 \bullet g_4)$, $G_2 = g_{11} + (p_{11} \bullet g_{10}) + (p_{11} \bullet p_{10} \bullet g_9) + (p_{11} \bullet p_{10} \bullet p_9 \bullet g_8)$, $G_3 = g_{15} + (p_{15} \bullet g_{14}) + (p_{15} \bullet p_{14} \bullet g_{13}) + (p_{15} \bullet p_{14} \bullet p_{13} \bullet g_{12})$
    - the super carry signal (C_i); $C_1 = G_0 + (P_0 \bullet c_0)$, $C_2 = G_1 + (P_1 \bullet G_0) + (P_1 \bullet P_0 \bullet c_0)$, $C_3 = G_2 + (P_2 \bullet G_1) + (P_2 \bullet P_1 \bullet G_0) + (P_2 \bullet P_1 \bullet P_0 \bullet c_0)$, $C_4 = G_3 + (P_3 \bullet G_2) + (P_3 \bullet P_2 \bullet G_1) + (P_3 \bullet P_2 \bullet P_1 \bullet G_0) + (P_3 \bullet P_2 \bullet P_1 \bullet P_0 \bullet c_0)$
  - to add a collection of numbers together, use carry save adders: the adder can add 3 inputs together (a_i, b_i, c_i) and produce two output (s, c_{i+1})
  - one simple way to model time for logic is assume each AND or OR gate takes the same time for a signal to pass through it; time is estimated by simply counting the number of gates along the longest (or critical) path through a piece of logic
- a barrel shifter can shift from 1 to 31 bits, and shifting is normally done outside of ALU

Multiplication

- for multiplication, the first operand is a n-bit multiplicand, the second is an m-bit multiplier, and the final result is a product with $n + m$ bits
- for binary multiplication, starting from right to left of the multiplier, just place a copy of the multiplicand in the proper place if the digit is 1, or place 0 in the proper place if the digit is 0
- the multiplication algorithm and hardware
  - the multiplicand register, ALU, and product register are all 64 bits wide, with only the multiplier register containing 32 bits; the product is initialized to 0, the multiplicand starts in the right half of the multiplicand register, and is shifted left 1 bit on each step; the multiplier shifts in the opposite direction at each step; a control is used to decide when to shift the multiplicand and multiplier and when to write new values into the product register; these steps repeated 32 times to obtain the product
    - half of the bits of the multiplicand were always 0
  - the multiplicand register, ALU, and multiplier register are all 32 bits wide, with only the product register left at 64 bits; the product register is initialized to 0, the multiplicand always add to the left half of the 64-bit product, the product and multiplier shift right 1 bit on each step
    - the product register always wastes 32 bits
  - eliminate the multiplier register by combining the right most half of the product with the multiplier; it starts by assigning the multiplier to the right half of the product register, placing 0 in the upper half; the product shifts right 1 bit on each step
- signed number multiplication
  - convert the multiplier and multiplicand to positive numbers (remember the origin signs), get the product and then negate the product only if the original signs disagree
  - Booth's algorithm:
    - an ALU could add or subtract to get the same result in more than one way
    - in machines of Booth's era, shifting was faster than addition
    - classify group of bits into the beginning, the middle, or the end of a run of 1s, replace a string of 1s in the multiplier with an initial subtract when we first see a 1 and then later add when we see the bit after the last 1
    - look at 2 bits of the multiplier, assume the pair of bits examined consists of the current bit and the bit to the right
      - 00: middle of a string of 0s, no arithmetic operation
      - 01: end of a string of 1s, add the multiplicand to the left half of the product
      - 10: beginning of a string of 1s, subtract the multiplicand from the left half of the product
      - 11: middle of a string of 1s, no arithmetic operation
    - when shifting the product right, must extend the sign of the intermediate result (an arithmetic right shift)
    - let a be the multiplier and b be the multiplicand and use a_i to refer to bit i of a; the value of expression $a_{i-1} - a_i$ are:
      - 0: do nothing
      - +1: add b
      - -1: subtract b
    - since $-a_i * 2^i + a_i * 2^{i+1} = a_i * 2^i$, the product = $(0 - a_0) * b * 2^0 + (a_0 - a_1) * b * 2^1 + .. + (a_{30} - a_{31}) * b * 2^{31} = b * (a_{31} * -2^{31} + \sum_{i=0}^{30}(a_i * 2^i)) = b * a$
    - Booth's algorithm is sensitive to particular bit pattern, the isolated 1s (i.e., alternate 0 and 1) cause the hardware to add or subtract at each step

- strength reduction: compiler substitute a left shift for a multiply by a power of 2
- MIPS provides a separate pair of 32-bit registers to contain the 64-bit product: *Hi* and *Lo*
  - instructions: **mult** (multiply), **multu** (multiply unsigned), **mflo** (move from *Lo*), **mfhi** (move from *Hi*)
  - both MIPS multiply instructions ignore overflow, to avoid overflow, *Hi* must be 0 for **multu** or must be the replicated sign of *Lo* for **mult**

Division

- division is an operation that is even less frequent, also it has the opportunity to perform mathematically invalid operation: divide by 0
- dividend = quotient * divisor + remainder, where remainder < divisor
- the basic algorithm to figure out how many times the divisor goes into the portion of the dividend
- division algorithm and hardware:
  - the divisor register, ALU, and remainder register are all 64 bits wide, with only the quotient register being 32 bits; the 32-bit divisor starts in the left half of the divisor register and is shifted right 1 bit on each step; the quotient is initialized to 0, and is shift left 1 bit on each step; the remainder is initialized with the dividend; control decides when to shift the divisor and quotient registers and when to write the new value into the remainder register
    - restoring division: the computer can't know in advance whether the divisor is smaller than the dividend; it must first subtract the divisor from the remainder and perform the **slt** instruction; if the result is positive, generate a 1 in the quotient, otherwise, restore the original value by adding the divisor back to the remainder and generate a 0 in the quotient; therefore, these steps need to be repeated for 33 times
    - non-restoring division: 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 $(r + d) * 2 - d = r * 2 + d$
    - at most half of the divisor has useful information
  - the divisor, ALU, and quotient registers are all 32 bits wide, with only the remainder register left at 64 bits; shifting the remainder to the left instead of shifting the divisor to the right
    - the first step cannot produce a 1 in the quotient bit, if it did, then the quotient would be too large for the register; by switching the order of the operations to shift and then subtract, it only need iterated 32 times
    - the remainder register always wastes 32 bits
  - the quotient register could be eliminated by shifting the bits of the quotient into the remainder instead of shifting in 0s; the remainder register shifts both the remainder in the left half and the quotient in the right half; the remainder will be shifted left one time too many, thus the final correction step must shift back only the remainder in the left half of the register
- signed division
  - the simplest solution is to remember the signs of the divisor and dividend, and then negate the quotient if the signs disagree
  - the dividend and the remainder must have the same signs, no matter what the signs of the divisor and quotient
- MIPS uses the same hardware for both multiply and divide, it uses *Hi* to contain the remainder and *Lo* to contain the quotient
  - instructions: **div** (divide) and **divu** (divide unsigned)
  - MIPS software need to check division by 0 and determine if the quotient too large (since MIPS divide ignore overflow)

Floating Point

- real numbers: infinite numbers with fractions and form a continuum; the real line is divided up into 7 regions:
  - large negative numbers (negative overflow)
  - expressible negative numbers
  - small negative numbers (negative underflow)
  - zero
  - small positive numbers (positive underflow)
  - expressible positive numbers
  - large positive numbers (positive overflow)

  underflow error is less serious than overflow error, because 0 is often a satisfactory approximation to small numbers
- floating-point: the decimal/binary point is not fixed, and not form a continuum; use rounding to the nearest number that can be expressed; the space between adjacent expressible numbers is not constant but the relative error introduced by rounding is approximately the same for small numbers as large numbers
- scientific notation has a single digit to the left of the decimal point
- normalized numbers (without leading 0s) have 3 advantages:
  - simplifies exchange of data
  - simplifies the floating-point arithmetic algorithms
  - increase the accuracy of the numbers

  there is only one normalized form, whereas there are many unnormalized forms
- many machines dedicate hardware to run floating-point operations
- many compilers do constant folding to avoid runtime computation
- code commonly found in scientific programs are floating-point operations on matrices
- the accuracy of floating-point calculations depends a great deal on the accuracy of rounding
- floating-point representation

- the trade-off between accuracy and range (design principle #3): since a fixed size, increasing the size of significand (fraction or mantissa) enhance the accuracy, while increasing the size of exponent increase the range of numbers that can be represented
- sign and magnitude representation: a floating-point number is usually a multiple of the size of a word, and contains a sign bit *S*, a exponent field *E*, and a significand field *F*; the value of the floating-point number is $(-1)^S * F * 2^E$
  - single precision floating-point: 1 bit for *S*, 8 bits for *E* and 23 bits for *F*, it can represent from $2.0 * 10^{-38}$ to $2.0 * 10^{38}$
  - double precision floating-point: 1 bit for *S*, 11 bit for *E* and 52 bits for *F*
  - extended precision floating-point: 80 bits intended to reduce roundoff errors
  - overflow: the exponent is too large to be represented in the exponent field
  - underflow: the nonzero fraction they are calculating has become so small that it cannot be represented, and thus the negative exponent is too large to fit in the exponent field
  - IEEE 754 (William Kahan) makes the leading 1 bit of normalized binary number implicit to pack even more bits into the significand; hence, the significand is actually 14 bits long for single precision, and 53 bits long for double precision floating-point, the value of the floating-point number is $(-1)^S * (F + 1) * 2^E$
  - IEEE 754 makes the floating-point representation could be easily processed by integer comparisons; it uses biased notation for the exponent field (127 for single precision); the value of the floating-point number is $(-1)^S * (F + 1) * 2^{(E - bias)}$
  - IEEE 754 has 5 numerical types:
    - normalized: 0 < exponent < maxExponent
    - denormalized: exponent = 0 and the implicit 1 bit to the left of the binary point becomes a 0; a graceful underflow by giving up significance
    - zero: all bits are zeros except the sign bit (2 zeros: +0 and -0); the bit to the left of the binary point is implicitly 0
    - infinity: exponent = maxExponent and significand = 0
    - not a number (NaN): reserves the pattern of all one bits for the exponent indicating values outside the scope of normal floating-point numbers
  - IBM 360/370 attempted to increase range without removing bits from the significand by using base 16 for the exponent field, but the normalized base 16 numbers can have up to 3 leading 0s, which leads to surprising problem in the accuracy of floating-point arithmetic
- floating-point addition: add numbers in scientific notation
  1. align the decimal point of the number that has the smaller exponent (since there are multiple representations of an un-normalized scientific notation) by shifting the significand to the right until its corrected exponent matches that of the number with the larger exponent]
  2. add the significands
  3. convert the sum into normalized form, also check for overflow or underflow
  4. round the result using the rule that 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; if we have bad luck on rounding, we need to perform the step 3 again
- floating-point multiplication
  1. calculate the exponent of the product by simply adding the exponents of the operands together; this can be done with the biased exponents (when adding biased numbers, we must subtract the bias from the sum)
  2. multiply the significands
  3. convert to normalized form and check overflow/underflow
  4. round the product (may need more normalization)
  5. if both operands have the same sign, the sign of the product is positive, otherwise, negative
- floating-point instructions in MIPS
  - floating-point addition: single (**add.s**) and double (**add.d**)
  - floating-point subtraction: single (**sub.s**) and double (**sub.d**)
  - floating-point multiplication: single (**mul.s**) and double (**mul.d**)
  - floating-point division: single (**div.s**) and double (**div.d**)
  - floating-point comparison: single (**c.x.s**) and double (**c.x.d**), where **x** can be **eq**, **neq**, **lt**, **le**, **gt**, **ge**
  - floating-point branch: true (**bclt**) and false (**bclf**)
  - separate floating-point registers: $f0, $f1, $f2, .., $f31 (unlike integer registers, $f0 can contain a nonzero number), a double precision register is really an even-odd pair of single precision registers, the single precision does not use the odd numbered register, both using the even register number as the name
  - separate loads (**lwc1**) and stores (**swc1**) for floating-pointer registers; early microprocessor didn't have enough transistors to put the floating-point unit on the same chip as the integer unit, hence the floating-pointer unit and registers are put in the coprocessor 1 (coprocessor 0 deals with virtual memory)
  - pseudo-instructions: **li** for load constant, **l.d** for **lwc1**, and **s.d** for **swc1**
  - hardware for multiplication is fast, but floating-point division is more difficult to make fast (except on parallel machines) and accurate; Newton's iteration technique performs divide by recasting as finding the zero of a function to find the reciprocal 1/*x*, which is then multiply by the other operand
- accurate arithmetic

- numerical precision is the very soul of science
- computer numbers have limited size, hence limited precision; computer arithmetic is finite and thus can disagree with natural arithmetic
- speed gets you nowhere if you're headed the wrong way
- integers can be represented exactly, but floating-point numbers are normally approximations for a number that they can't really represent (because there are an infinite real numbers exists between 0 and 1, but no more then 2^{53} can be represented exactly in double precision)
- rounding accurately requires the hardware to include extra bits
- if every intermediate result had to be truncated to the exact number of digits, there would be no opportunity to round
- there are 4 rounding modes:
  - always round up (toward $+\infty$)
  - always round down (toward $-\infty$)
  - truncate
  - round to nearest even
- IEEE 754 always keep 2 extra bits (*guard* and *round*) on the right during intermediate calculation, which truncate values 0 to 49, and round up values 50 to 99 (multiply can need 2 bits when the normalizing shift the *guard* bit into the least significant bit of the product, leaving the *round* bit to help accurately round)
- an extra third bit (*sticky*) allows the machine to get the same results as if the intermediate results were calculated to infinite precision; it is set whenever there are nonzero bits to the right of the *round* bit, and allows the computer to distinguish between 0.50..00 and 0.50…01 when rounding
- accuracy in floating-point is measured in terms of number of bits in error in the least significant place (or *ulp*); provide there is no overflow, underflow, or invalid operation exceptions, IEEE 754 guarantees that the computer uses the number that is within one-half *ulp*
- IEEE 754 has special symbols to represent unusual events
  - infinite: the largest exponent is reserved for this symbol
  - NaN (not a number) for invalid operation (e.g., 0/0, $\infty - \infty$): allow programmers to postpone some tests and decisions to a later time in the program when it is convenient; uses ordered and unordered as options for compares
  - gradual underflow: rather than having a gap between 0 and the smallest normalized number, IEEE allows denormalized number (denorms or subnormals), they have the same exponent as zero but a nonzero significant to allow a number to degrade in significance until it becomes 0; this cause headaches to hardware designers to make it fast, hence many computers cause an exception and let software to complete the operation (may not portable)
- numerical analysis: the study of imprecision and limited representation of floating-point

Floating Point in PowerPC and Intel 80x86

- PowerPC has a fused multiply-add instruction: a single instruction reads 3 operands, multiplies 2 operands and adds the 3rd to the product and round the sum and writes the sum in the result operand
- Intel 8087 floating-point coprocessor uses a stack architecture (a register-memory model), it uses ST to indicate the top of stack, and ST(i) to represent the i^{th} register below the top of stack; the operands are wider in the register stack (80 bits) than they are stored in memory (double extended precision); the floating-point operations are (curly brackets to show optional variation of the basic operations, {I} for integer, {P} for pop, and {R} for reverse order):
  - data movement instructions: F{I}LD mem/ST(i), F{I}ST{P} mem/ST(i), FLDPI, FDL1, FLDZ, etc.
  - arithmetic instructions: F{I}ADD{P} mem/ST(i), F{I}SUB{R}{P} mem/ST(i), F{I}MUL{P} mem/ST(i), F{I}DIV{R}{P} mem/ST(i), FSORT, FABS, FRNDINT, etc.
  - comparison instructions: send the result to the integer processor by FSTSW instruction followed by an SAHF instruction to set the condition code so that the integer processor can branch, F{I}COM{P}{P}, F{I}UCOM{P}{P}, FSTSW AX/mem, etc.
  - transcendental instructions: FPATAN, F2XM1, FCOS, FPTAN, FPREM, FSIN, FYL2X, etc.

Fallacies and Pitfalls

- ❖ fallacy: floating-point addition is associative: $x + (y + z) = (x + y) + z$
  - ✓ when adding 2 large numbers of opposite signs plus a small number, the result is not associative and depends on the order of floating-point addition
- ❖ fallacy: just as a left shift instruction can replace an integer multiply by a power of 2, a right shift is the same as an integer division by a power of 2
  - ✓ this is only true for unsigned integers
  - ✓ to make signed integers working, the PowerPC has a shift right algebraic instruction
- ❖ pitfall: the MIPS instruction add immediate unsigned **addiu** sign-extends its 16-bit immediate field
  - ✓ MIPS has no subtract immediate instruction and negative numbers need sign extension, so the MIPS architecture decided to sign-extend the immediate field
- ❖ fallacy: only theoretical mathematicians care about floating-point accuracy
  - July 1994 Intel discovers the bug in the Pentium, the recall in December 1994 cost Intel $300 million; in April 1997 another floating-point bug was revealed in the Pentium Pro and Pentium II, Intel

publicly acknowledged the bug and offer a software patch to get around it

Historical perspective

- scale factors: multiplication constants which serve to keep numbers within the limits of the machine
- floating factors: 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; accuracy was sacrificed because the least significant bits had to be lost on the right to accommodate leading 0s
- true floating-point hardware became popular because it was useful by 1957
- portable numerical software (e.g., Linpack and Eispack) is distributed as source code to be compiled and executed on practically any commercial machines, but it was too expensive (>$100/line)
- a theorem that single precision then (and now) honored: if $1/2 \leq x/y \leq 2$, then no rounding error can occur when x–y is computed
- IBM 360 is the successor to the 7094 series, but 360 had narrower single precision word (32 bits) and without the *guard* digit in double precision; reasonable implementation of approximate arithmetic were not appreciated until they were lost