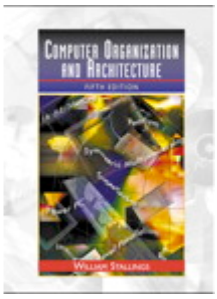
Campus de Gualtar
4710-057 Braga

UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Departamento de
Informática



Computer Organization and Architecture

5th Edition, 2000

by William Stallings

Table of Contents

III. THE CENTRAL PROCESSING UNIT.

8. Computer Arithmetic. (19-Apr-99)

Integer Representation (8.2)

- Sign-Magnitude Representation
  - Leftmost bit is sign bit: 0 for positive, 1 for negative
  - Remaining bits are magnitude
  - Drawbacks
    - Addition and subtraction must consider both the signs and relative magnitudes -- more complex
    - Testing for zero must consider two possible zero representations
- Two's Complement Representation
  - Leftmost bit still indicates sign
  - Positive numbers exactly same as sign-magnitude
  - Zero is only all zeroes (positive)
  - Negative numbers found by taking 2's complement
    - Take complement of positive version
    - Add 1

Integer Arithmetic (8.3)

- 2's complement examples (with 8 bit numbers)
  - Getting -55
    - Start with +55: 0110111
    - Complement that: 1001000
    - Add 1: +0000001
    - Total is -55: 1001001
  - Negating -55
    - Complement -55: 0110110
    - Add 1: +0000001
    - Total is 55 (see top) 0110111
  - Adding -55 + 58
    - Start with -55: 1001001
    - Add 58: +0111010
    - Result is 3: 0000011
    - Overflow into and out-of sign bit is ignored
- Overflow Rule - if two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign
- Converting between different bit lengths
  - Move sign bit to new leftmost position
  - Fill in with copies of the sign bit
  - Examples (8 bit -> 16 bit)
    - +18: 00010010 -> 0000000000010010
    - -18: 11101110 -> 1111111111101110

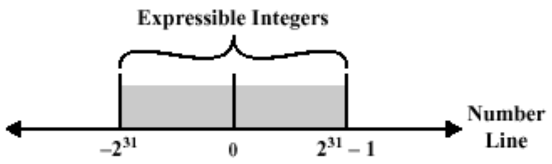
- Multiplication
  - Repeated Addition
  - Unsigned Integers
    - Generating partial products, shifting, and adding
    - Just like longhand multiplication
- Two's Complement Multiplication
  - Straightforward multiplication will not work if either the multiplier or multiplicand are negative
    - multiplicand would have to be padded with sign bit into a 2n-bit partial product, so that the signs would line up
    - in a negative multiplier, the 1's and 0's would no longer correspond to add-shift's and shift-only's
  - Simple solution
    - Convert both multiplier and multiplicand to positive numbers
    - Perform multiplication
    - Take 2's complement of result if and only if the signs of original numbers were different
    - Other methods do not require this final transformation step
- Booth's Algorithm
- Why does Booth's Algorithm work?
  - Consider multiplying some multiplicand M by 30: M * (00011110) which would take 4 shift-adds of M (one for each 1)
  - That is the same as multiplying M by (32 - 2): M * (00100000 - 00000010) = M * (00100000) - M * (00000010) which would take:
    - 1 shift-only on no transition (imagine last bit was 0)
    - 1 shift-subtract on the transition from 0 to 1
    - 3 shift-only's on no transition
    - 1 shift-add on the transition from 1 to 0
    - 2 shift-only's on no transition
  - We can extend this method to any number of blocks of 1's, including blocks of unit length.
  - Consider the smallest number of bits that can hold the 2's complement representation of -6: So we can clearly see that a shift-subtract at the leftmost 1-0 transition will cause 8 to be subtracted from the accumulated total, which is exactly what needs to happen!
  - This will expand to an 8-bit representation: The neat part is that this same (and only) 1-0 transition will also cause -8 to be subtracted from the 8-bit version!
- Division
  - Unsigned integers 00001101 Quotient Divisor 1011 10010011 Dividend 1011 001110 1011 001111 1011 100 Remainder

Floating-Point Representation (8.4)

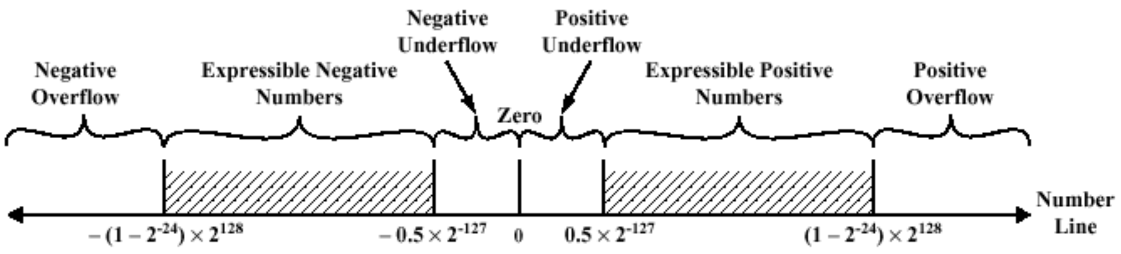
- Principles
  - Using scientific notation, we can store a floating point number in 3 parts ±S * B±E :
    - Sign
    - Significand (or Mantissa)
    - Exponent
    - (The Base stays the same, so need not be stored)

- The sign applies to the significand. Exponents use a biased representation, where a fixed value called the bias is subtracted from the field to get the actual exponent.

- We require that numbers be normalized, so that the decimal in the significand is always in the same place
  - we will choose just to the right of a leading 0
  - format will be ±0.1bbb…b * 2±E
  - thus, it is unnecessary to store either that leading 0, or the next 1, since all numbers will have them
  - for the example following, assume also:
    - An 8 bit sign with a bias of 128
    - A 24-bit significand stored in a 23-bit field
- Example Ranges

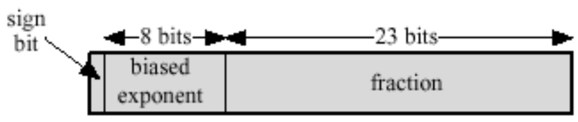


(a) Twos Complement Integers

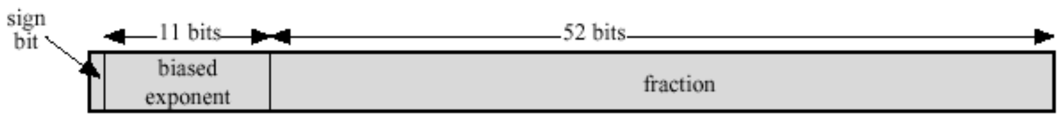


(b) Floating-Point Numbers

  - This compares to a range of –2^31 to 2^31-1 for 2's complement integers for the same 32 bits
  - There is more range, but no more individual values
  - FP numbers are spaced unevenly along the number line
    - More densely close to 0, less densely further from 0
    - Demonstrates tradeoff between range and precision
    - Larger exponent gives more range, larger significand gives more precision
- IEEE Standard for Binary Floating-Point Arithmetic (IEEE 754)
  - Facilitates portability of programs from one processor to another
  - Defines both 32-bit single and 64-bit double formats



(a) Single format



(b) Double format

- Defines some parameters for extending those formats for more range or more precision

- Classes of numbers represented:
  - Positive or Negative Zero - an exponent of 0 together with a fraction of zero (sign bit determines sign).
  - Denormalized Numbers - an exponent of 0 together with a nonzero fraction. The fraction is the portion of the significand to the right of the decimal (0 is assumed to the left). Note that the fraction does NOT have an implied leftmost one.
  - Positive or Negative Infinity - an exponent of all 1's, together with a fraction of zero (sign bit determines sign).
  - NaN (Not a Number) - an exponent of all ones, together with a non-zero fraction. Used to signal various exception conditions.
  - Normalized, Non-Zero Floating Point Numbers - everything else.
- Potential problems:
  - Exponent Overflow - the number is too large to be represented, may be reported as +infinity or -infinity .
  - Exponent Underflow - the number is too small to be represented, may be reported as 0.
  - Significand Underflow - during alignment of significands, digits may flow off the right end, requiring some form of rounding.
  - Significand Overflow - adding two significands with the same sign may result in a carry out of the most significant bit, fixed by realignment.

Floating Point Arithmetic (8.5)

- Addition and Subtraction
  - More complex than multiplication and division
  - 4 basic phases
    - Check for zeros
    - Align the significands
    - Add or subtract the significands
    - Normalize the result
  - The two operands must be transferred to registers in the ALU
    - implicit significand bits must be made explicit
    - exponents and significands usually stored in separate registers
  - If subtraction, change sign of subtrahend
  - If either operand is 0, report other as result
  - Manipulate the numbers so that the two exponents are equal
    - Ex. 123 x 10^0 + 456 x 10^-2 = 123 x 10^0 + 4.56 x 10^0 = 127.56 x 10^0
    - Done by shifting smaller number to the right
      - Simultaneously incrementing the exponent
      - Stops when exponents are equal
      - Digits lost are of relatively small importance
      - If significand becomes 0, report other as result
  - The numbers are added together
    - Signs are taken into account, so zero may result
    - If significand overflows, the significand of the result is shifted right, and the exponent is incremented
    - If exponent then overflows, it is reported and operation halted

  - Normalize the result
    - significand digits are shifted left until the most significant digit is non-zero
    - exponent is decremented for each shift
    - If exponent underflows, report it and halt
  - Round result and store in proper format

- Multiplication
  - The two operands must be transferred to registers in the ALU
  - If either operand is 0, report 0 as result
  - Add the exponents
    - If a biased exponent is being used, subtract the extra bias from the sum
    - Check for exponent overflow or underflow. Report it and halt if it occurs.
  - Multiply the significands
    - Same as for integers, but sign-magnitude representation
    - Product will be double length of multiplier and multiplicand, but extra bits will be lost during rounding
  - Result is normalized and rounded
    - Same as for addition and subtraction
    - Thus, exponent underflow could result

- Division
  - The two operands must be transferred to registers in the ALU
  - Check for 0
    - If divisor is 0, report error and either set result to infinity or halt operation
    - if dividend is 0, report 0 as result
  - Subtract divisor exponent from dividend exponent
    - If a biased exponent is being used, add the bias back in.
    - Check for exponent overflow or underflow. Report it and halt if it occurs.
  - Divide the significands
    - Same as for integers, but sign-magnitude representation
  - Result is normalized and rounded
    - Same as for addition and subtraction
    - Thus, exponent underflow could result

- Precision Considerations
  - Guard Bits
    - Extra bits that pad out the right end of the significand with 0's
    - Used to prevent loss of precision when adding numbers which are very close in value
    - Example without guard bits:
      ```
       1.0000000 * 2^1 +
      -1.1111111 * 2^0 =
       1.0000000 * 2^1 +
      -0.1111111 * 2^1 =
       0.0000001 * 2^1 Normalized to:
       1.0000000 * 2^-7
      ```
    - Example with guard bits:
      ```
       1.00000000 * 2^1+
      -1.11111110 * 2^0 =
       1.00000000 * 2^1 +
      -0.11111111 * 2^1 =
       0.00000001 * 2^1 Normalized to:
       1.00000000 * 2^-8
      ```

        - Notice the order of magnitude difference in results! Difference is worse with base-16 architectures.
    - Rounding
        - 4 alternative approaches with IEEE standard
            - Round to Nearest
                - Result is rounded to nearest representable number
            - Default rounding mode
                - The representable value nearest to the infinitely precise result shall be delivered * if excess bits are less than half of last representable bit, round down * if excess bits are half or more of last representable bit, round up * if they are equally close, use the one with LSB 0
            - Round Toward + infinity and -infinity
                - Result is rounded up toward positive infinity or Result is rounded down toward negative infinity
                - Useful in implementing interval arithmetic * every calculation in a sequence is done twice -- once rounding up, and once rounding down, producing upper and lower bounds on the result. * if resulting range is narrow enough, then answer is sufficiently accurate * useful because hardware limitations cause rounding
            - Round Toward 0
                - Result is rounded toward 0
                - Just simple truncation
                - Result is always less than or equal to the more precise original value, introducing a consistent downward bias

- IEEE Standard for Floating Point Arithmetic
    - Infinity
        - Most operations involving infinity yield infinity
        - Signs obey usual laws
        - -infinity -infinity yields -infinity and +infinity +infinity yields +infinity
    - Quiet and Signaling NaN’s
        - A signaling NaN causes an exception (which may be handled by the program, or may cause an error)
- A quiet NaN propagates through operations in an expression without signaling an exception. They are produced by:
    - Any operation on a signaling NaN
    - Magnitude subtraction of infinities (where you might expect a zero result)
    - 0 x infinity
    - (0 / 0) or (infinity / infinity). Note that x / 0 is always an exception.
    - (x MOD 0) or (infinity MOD y)
    - square-root of x, where x < 0
- Denormalized Numbers
    - Used in case of exponent underflow
    - When the exponent of the result is too small (a negative exponent with too large a magnitude) the result is denormalized
        - right-shifting the fraction
        - incrementing the exponent for each shift, until it is all ones with a final 0
    - Referred to as gradual underflow, use of denormalized numbers:
        - fills the gap between the smallest representable non-zero number and 0
        - reduces the impact of exponent underflow to a level comparable to round off among the normalized numbers