18613 - Computer Systems

optimization point

related area

Programs and Data

data

- architecture and compilers
- data representation
  - data in difference base → transfer number in base to base 10
    - e.g. 1001 (base-x) = 1*x^3 + 1*x^0 = 1*x^3 + 1*x^0 = 8 (base-10)
  - range bound → overflow → go over the range again
    - add a sign bit
  - sign representation
    - negative design logic → require the -n = ~n, where n is any natural number
      - so,could not add a sign 1 in the front of a n make it a n, because n + (-n) = ~n(~ means of range)
      - so in machine, -n = ~n + 1 → where ~n is the complement of n
  - byte ordering
    - Big Endian → Least significant byte has highest address
      - used in Sun JAVA, IPC Mac, Internet
    - Little Endian → Least significant byte has lowest address
      - used in x86, ARM
    - e.g.
      - 0x1234abcd
        - 0x100 0x101 0x102 0x103
        - 0x100 0x101 0x102 0x103
  - integer
    - representation
      - unsigned
        - if given a w bits binary number x_3, x_2, ..., x_w-1
          - $B2U(x) = \sum_{i=0}^{w-1} x^i$
        - range → [0, 2^w -1]
      - signed, nbz two's complement
        - $B2T(x) = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-1} x^i$
        - range → [-2^(w-1), 2^(w-1) -1]
    - conversion & casting
      - bit pattern is kept, but reinterpreted
      - signed and unsigned in single expression, signed values implicitly cast to unsigned
        - unsigned → pad 0
        - signed → pad 1
    - expanding & truncating
      - expanding → signed → pad 1
        - always yield expected result
      - truncating → bit pattern is kept, but reinterpreted
        - bits dong need
        - result sometimes unexpected
    - calculation
      - adding → if overflow, discard carry bits
      - minusing → mind the range too
      - multiplication
      - division → **efficient mode / divisions**
        - using shift
        - for multiply → **$* \Leftrightarrow << p$ pass $+ \Leftrightarrow P$**
          - logic shift
          - unsigned → **$>> p$ pass $| \Leftrightarrow / 2^p$**
        - for divide
          - arithmetic shift
          - signed → rounding toward left → **$+ \Leftrightarrow << p$ pass $| \Leftrightarrow / 2^p$**
            - **Common x / 2 <=>(x<0) ? x+1 : x**
            - **Common x % 2 <=>(x<0) ? -x : x**
            - **Base default round 0**
            - rounding toward 0 → add 0.5 in base 10 → lead to modify according to the nominator part
  - float
    - scientific notation → **$3.1415926535898 \times 10^{1}$**
    - numerical form → $(-1)^S \cdot M \cdot 2^E$
      - Sign bit s → determines whether number is negative or positive
      - significand M → normally a fractional value in range [1.0,2.0). Sometimes, normally leading a implicit leading 1 = 1.M
      - Exponent E → weights value by power of two
        - E need to minus a bias
        - not a signed number actually → because when this bit equals 1, the exponent of 0x ffed is scientific notation is positive
        - the bias is for distinguishing whether the exponent of the float is normally notation is positive or negative
    - IEEE 754
      - floating point standard
      - simple precision 32bits
        - 1 bit s + 8 bits exp + 23 bits frac
        - can represent 7 decimal digits, 10^6-1M
      - double precision 64bits
        - 1 bit s + 11 bits exp + 52 bits frac
        - can represent 16 decimal digits, 10^16-1M
      - types regrading to E
        - normalized
          - $exp = 00...00$
          - $\mathbf{s = (-1)^S M 2^E}$
            - M = 0.frac
            - E = 1-bias
          - when $exp = 00...01$, $frac = 00...00$ → represent 0
          - when $exp = 00...01$, $frac = 00...00$ → represent number closest to 0
          - $exp = 0$ & $exp = 11...11$
            - $\mathbf{s = (-1)^S M 2^E}$ / $\mathbf{E = exp - Bias}$
              - M = 1.frac
              - E = exp - bias
            - construction examples
              - **Exponent**
                - **Fraction**
                - **Value**
                - **Comment**
                - **0 00000000 00000000000000000000000 = +0.0 = 0.0 × 2^(-126) Positive zero**
                - **1 00000000 00000000000000000000000 = -0.0 = -0.0 × 2^(-126) Negative zero**
                - **0 00000000 00000000000000000000001 = +0.00000012 = 0.00000012 × 2^(-126) = 2^(-149) Smallest positive subnormal**
                - **0 00000000 00000000000000000000010 = +0.00000024 = 0.00000024 × 2^(-126) = 2^(-148) Second smallest positive subnormal**
                - **0 00000000 00000000000000000000011 = +0.00000036 = 0.00000036 × 2^(-126) = 2^(-148)+2^(-149) Third smallest positive subnormal**
                - **0 00000000 00000000000000000000100 = +0.00000048 = 0.00000048 × 2^(-126) = 2^(-147) Fourth smallest positive subnormal**
                - **0 00000000 00000000000000000000101 = +0.00000060 = 0.00000060 × 2^(-126) = 2^(-147)+2^(-149) Fifth smallest positive subnormal**
                - **0 00000000 00000000000000000000110 = +0.00000072 = 0.00000072 × 2^(-126) = 2^(-147)+2^(-148) Sixth smallest positive subnormal**
                - **0 00000000 00000000000000000000111 = +0.00000084 = 0.00000084 × 2^(-126) = 2^(-147)+2^(-148)+2^(-149) Seventh smallest positive subnormal**
                - **0 00000000 00000000000000000000000 = +0.00000000 = 0.00000000 × 2^(-126) = 0 Positive zero**
                - **1 00000000 00000000000000000000000 = -0.00000000 = -0.00000000 × 2^(-126) = 0