Number Representation

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Unsigned and signed integers
- Overflow in integer operations
- Floating-point numbers
- Overflow in floating-point operations
- Underflow in floating-point operations
- Rounding modes

Unsigned and signed integers

- Numbers in a computer (or a processor) are represented in binary
  - Digital designs are realized using devices such as transistors which can be made to operate at two discrete voltage levels representing 0 and 1 conveniently
  - Number of bits devoted to represent a number is fixed by the design and cannot be extended after the design is finished
    - Puts an upper bound on the value of the largest representable number in a computer
    - If a computation generates a number that cannot fit within these bits, an overflow is said to have occurred

Unsigned and signed integers

- Non-negative integers (often called unsigned integers) can be represented using standard binary
- Negative integers require the sign to be encoded appropriately
  - Four possibilities have been tried: sign magnitude, two's complement, one's complement, biased
  - Sign magnitude is the simplest: reserve the most significant bit to represent the sign of the integer
    - Ambiguous representation of zero (00…0 and 100…0)
    - Addition requires extra logic to set the result's sign

Representing negative integers

- Sign magnitude representation
  - With n bits to represent a number, the representable range is $-2^{n-1}+1$ to $2^{n-1}-1$
- Two's complement representation
  - Non-negative integers are represented as in sign magnitude representation
    - MSB is 0 followed by the magnitude in n-1 bits
  - An integer x and its negative add up to 2^n
    - $x+(-x) = (2^n - 1) + 1$ or $(-x) = (2^n - 1) - x + 1$
    - Observe: $(2^n - 1) - x$ is same as bitwise inversion of x
    - Offers an easy implementation to derive –x from x
      - Invert bits of x to get y; add 1 to y; ignore carry out of MSB

Representing negative integers

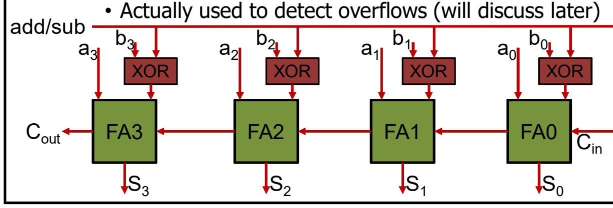
- Two's complement representation
  - Unambiguous representation of zero (00…0)
  - No special treatment needed by the sign bit
    - To convert a two's complement binary number to decimal use -2^{n-1} as the coefficient for the MSB and the rest is same as regular binary numbers
      - Proof outline below:
      - Consider an n-bit negative integer x in two's complement representation: $x=1b_{n-2}b_{n-3}\ldots b_1b_0$
      - Therefore, -x in two's complement representation is $0a_{n-2}a_{n-3}\ldots a_1a_0 + 1$ where $a_i = 1 - b_i$ (follows from the definition of two's complement)
      - The decimal equivalent of –x is $1+\sum_{i=0}^{n-2} a_i 2^i = 1+\sum_{i=0}^{n-2}(1-bi)2^i$
      - Therefore, the decimal equivalent of x is $-1-\sum_{i=0}^{n-2}(1-bi)2^i = -\sum_{i=0}^{n-2} 2^i - 1 + \sum_{i=0}^{n-2} b_i 2^i = -2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$

Representing negative integers

- Two's complement representation
  - No special hardware or step required in binary addition and subtraction (A+B or A-B)
    - A and B are in two's complement representation
    - An extra array of xor gates to implement A-B
    - Ignore carry out of MSB position
    - Basic binary addition algorithm remains unchanged
  - Only drawback is an imbalanced range on positive and negative sides
    - Representable range: -2^{n-1} to $2^{n-1}-1$
- Two's complement representation is used in all computers today for handling integers

Representing negative integers

- Two's complement adder/subtractor
  - For A+B, the add/sub input wire is set to 0 and for A-B, the add/sub input wire is set to 1
  - Both A and B must be in two's complement representation (assumes n=4)
  - C_{out} is usually ignored
    - Actually used to detect overflows (will discuss later)



Representing negative integers

- One's complement representation
  - An integer x and its negative add up to $2^n - 1$
    - $x+(-x) = 2^n - 1$ or $(-x) = 2^n - 1 - x$
    - Just bitwise inversion of x is –x
  - Ambiguous representation of zero
    - 00…0 and 11…1
  - How easy is addition or subtraction?
    - Various cases have different treatments
    - Not as easy as two's complement addition/subtraction
  - Balanced ranges on positive and negative sides
    - $-2^{n-1}+1$ to $2^{n-1} - 1$

Representing negative integers

- Biased representation
  - Representation of an integer x is x+bias
    - x+bias is represented as a regular binary number with no sign bit
    - bias is a constant fixed by the representation
  - The goal is to avoid negative numbers all together i.e., $x+bias \geq 0$
    - Smallest representable number is -bias
  - A typical bias is 2^{n-1} for n-bit representation
    - An integer x is represented as $x+2^{n-1}$ using unsigned binary representation
  - Imbalanced range on positive and negative sides
    - -bias to $2^n - 1 - bias$
    - -2^{n-1} to $2^{n-1} - 1$ when bias is 2^{n-1}

Representing negative integers

- Use of two's complement representation
  - All computers use this representation
  - The following C code can be used to verify whether a computer is using two's complement representation for 32-bit integers ("int" type)

```
int x = -10;     // Any value can be used
unsigned bits = *((unsigned*)&x);
printf ("bits = %#x\n", bits);
```

  - Similarly, 64-bit integers ("long long" type) can be verified

```
long long x = -10;
unsigned long long bits = *((unsigned long
long*)&x); printf ("bits = %#llx\n", bits);
```

Overflow in integer operations

- Overflow occurs if the result of a computation cannot fit within the given number of bits
  - In usual binary representation, if there is a carry-out in the MSB position of an addition, overflow is said to occur
  - In two's complement representation, overflow occurs if and only if the carry in to the MSB is different from carry out from the MSB
    - Proof?
    - Let the carry in to the MSB be c_{n-1} and carry out from the MSB be c_n when adding two n-bit numbers

Overflow in integer operations

- Overflow in two's complement addition
  - Suppose that we are adding A ($a_{n-1}a_{n-2}\ldots a_0$) and B ($b_{n-1}b_{n-2}\ldots b_0$) both in two's complement representation and let the sum be S ($s_{n-1}s_{n-2}\ldots s_0$)
  - Case I: $c_{n-1}=0$, $c_n=1$
    - Therefore, $a_{n-1}=b_{n-1}=1$ and $s_{n-1}=0$
    - This is an overflow condition because adding two negative numbers cannot yield a positive number
  - Case II: $c_{n-1}=1$, $c_n=0$
    - Therefore, $a_{n-1}=b_{n-1}=0$ and $s_{n-1}=1$
    - This is an overflow condition because adding two positive numbers cannot yield a negative number

Overflow in integer operations

- Overflow in two's complement addition
  - Consider adding A ($a_{n-1}a_{n-2}\ldots a_0$) and B ($b_{n-1}b_{n-2}\ldots b_0$) both in two's complement representation and let the sum be S ($s_{n-1}s_{n-2}\ldots s_0$)
  - Case III: $c_{n-1}=0$, $c_n=0$
    - Therefore, at most one of a_{n-1} and b_{n-1} is 1
    - Case IIIA: $a_{n-1}=b_{n-1}=0$
      - Since c_{n-1} is 0, the magnitudes of two positive numbers are added without any overflow
      - Therefore, there is no overflow
    - Case IIIB: $a_{n-1}=0$, $b_{n-1}=1$ ($a_{n-1}=1$, $b_{n-1}=0$ is similar)
      - Therefore, A+B = $-2^{n-1} + \sum_{i=0}^{n-2}(a_i + bi).2^i$
      - Since c_{n-1} is 0, there is no overflow in adding lower n-1 bits of A and B; hence, $s_{n-2}s_{n-3}\ldots s_0 = \sum_{i=0}^{n-2}(a_i + bi).2^i$
      - So, A+B = $-2^{n-1} + s_{n-2}s_{n-3}\ldots s_0 = 1s_{n-2}s_{n-3}\ldots s_0$ (no overflow)

Overflow in integer operations

- Overflow in two's complement addition
  - Consider adding A ($a_{n-1}a_{n-2}\ldots a_0$) and B ($b_{n-1}b_{n-2}\ldots b_0$) both in two's complement representation and let the sum be S ($s_{n-1}s_{n-2}\ldots s_0$)
  - Case IV: $c_{n-1}=1$, $c_n=1$
    - Therefore, at most one of a_{n-1} and b_{n-1} is 0
    - Case IVA: $a_{n-1}=1$, $b_{n-1}=0$ ($a_{n-1}=0$, $b_{n-1}=1$ is similar)
      - Therefore, A+B = $-2^{n-1} + \sum_{i=0}^{n-2}(a_i + bi).2^i$
      - Since c_{n-1} is 1, the result of adding lower n-1 bits of A and B is $1s_{n-2}s_{n-3}\ldots s_0 = \sum_{i=0}^{n-2}(a_i + bi).2^i$, where $1s_{n-2}s_{n-3}\ldots s_0$ is treated as a positive value i.e., $2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$
      - So, A+B = $-2^{n-1} + 1s_{n-2}s_{n-3}\ldots s_0 = -2^{n-1} + 2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$
        $= 0s_{n-2}s_{n-3}\ldots s_0$
      - Since in this case, $a_{n-1}+b_{n-1}+c_{n-1} = c_n0$, correct result is obtained by ignoring c_n (hence, no overflow)

Overflow in integer operations

- Overflow in two's complement addition
  - Consider adding A ($a_{n-1}a_{n-2}\ldots a_0$) and B ($b_{n-1}b_{n-2}\ldots b_0$) both in two's complement representation and let the sum be S ($s_{n-1}s_{n-2}\ldots s_0$)
  - Case IV: $c_{n-1}=1$, $c_n=1$
    - Therefore, at most one of a_{n-1} and b_{n-1} is 0
    - Case IVB: $a_{n-1}=1$, $b_{n-1}=1$
      - Therefore, A+B = $-2^n + \sum_{i=0}^{n-2}(a_i + bi).2^i$
      - Since c_{n-1} is 1, the result of adding lower n-1 bits of A and B is $1s_{n-2}s_{n-3}\ldots s_0 = \sum_{i=0}^{n-2}(a_i + bi).2^i$, where $1s_{n-2}s_{n-3}\ldots s_0$ is treated as a positive value i.e., $2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$
      - So, A+B = $-2^n + 1s_{n-2}s_{n-3}\ldots s_0 = -2^n + 2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i$
        $= -2^{n-1} + \sum_{i=0}^{n-2} s_i 2^i = 1s_{n-2}s_{n-3}\ldots s_0$ in two's complement
      - Since in this case, $a_{n-1}+b_{n-1}+c_{n-1} = c_n1$, correct result is obtained by ignoring c_n (hence, no overflow)

Floating-point numbers

- Real numbers are represented using scientific notation
  - a.b x 10^c
  - Said to be normalized if there is only one non-zero digit to the left of the decimal point
    - $1 \leq a \leq 9$
    - Always possible to normalize a real number
- Digital computers represent real numbers as floating-point binary numbers using normalized scientific notation
  - 1.b x 2^c where b and c are binary numbers
  - Binary point as opposed to decimal point

Floating-point numbers

- Normalized scientific representation
  - Floating-point because the binary point can be moved by adjusting the exponent
- How to convert a decimal real number to normalized scientific binary notation?
  - Procedure for integer part: continued division
  - Procedure for fraction part: continued multiplication
- A representation of normalized floating-point number fixes the number of bits needed for b and c, given a fixed total number of bits
  - Trade-off between range and precision
  - If b has more bits then c has less

Floating-point numbers

- Normalized scientific representation has three advantages
  - Simplifies exchange of floating-point data between computers due to a standard representation
  - Simplifies floating-point arithmetic algorithms because all operands are in the same format
  - Compacts the representation by discarding leading zeros on the left hand side
    - 0.0000000010101 is same as 1.0101 x 2^{-9} and it is enough to store b=0101 and c=-9
    - b is referred to as the fraction (or mantissa) and c is referred to as the exponent

Floating-point numbers

- Normalized scientific representation
  - IEEE 754 standard dictates the number of bits reserved for mantissa and exponent
    - It is a standard format for representing floating-point numbers
    - All computers are expected to follow this format
  - A single-precision floating-point number is represented using 32 bits
    - Sign magnitude representation
    - MSB is sign bit
    - Least significant 23 bits represent the mantissa
      - 24 bits of (1 + mantissa) represent the significand
    - Middle eight bits represent the exponent

Floating-point numbers

- IEEE 754 single-precision format
  - Exponent field needs to encode both positive and negative exponents
  - Necessary to choose an encoding such that the entire 31-bit magnitude field is monotonic in the value of the floating-point number
    - A larger magnitude floating-point number should have a larger 31-bit magnitude field compared to a smaller magnitude floating-point number
    - Helps to sort numbers quickly by magnitude
    - Mantissa is already monotonic in the value of the fraction
    - Cannot use two's complement encoding for exponent
      - Negative exponents will be represented by large numbers

Floating-point numbers

- IEEE 754 single-precision format
  - Biased encoding of exponent with bias set to 127
    - Encoded exponent = 127 + actual exponent
    - Actual exponent is allowed to range from -126 to 127 i.e., the encoded exponent can range from 00000001 to 11111110
    - Encoded exponent cannot be 00000000 or 11111111 because these encodings are reserved to represent some special numbers
    - Notice that larger encoded exponent now represents larger actual exponent because biased encoding preserves monotonicity
      - Two's complement encoding does not preserve monotonicity

Floating-point numbers

- IEEE 754 single-precision format
  - Largest non-negative number

0	11111110	11111111111111111111111

    - $+1.111\ldots1 \times 2^{127} = (2 - 2^{-23}) \times 2^{127}$
  - Smallest positive number

0	00000001	00000000000000000000000

    - $+1.000\ldots0 \times 2^{-126}$
  - Negative number with smallest magnitude

1	00000001	00000000000000000000000

    - $-1.000\ldots0 \times 2^{-126}$
  - Negative number with largest magnitude

1	11111110	11111111111111111111111

    - $-1.111\ldots1 \times 2^{127} = -(2 - 2^{-23}) \times 2^{127}$

Floating-point numbers

- IEEE 754 single-precision format
  - Special numbers
    - Encoding of zero (two possible representations):

X	00000000	00000000000000000000000

    - Encoding of +infinity:

0	11111111	00000000000000000000000

    - Encoding of –infinity:

1	11111111	00000000000000000000000

    - Encoding of NaN (result of 0/0, sqrt(-n), 0*inf, etc.):

X	11111111	Anything non-zero

Floating-point numbers

- IEEE 754 single-precision format
  - What about the numbers between 0 and ±N where N is the smallest representable magnitude ($1.000…0 \times 2^{-126}$)?
    - These are called denormalized numbers because they are of the form $\pm 0.b \times 2^c$ where c=-126
    - Largest non-negative denormalized number:

0	00000000	11111111111111111111111

      - $+0.111…1 \times 2^{-126} = (1 - 2^{-23}) \times 2^{-126}$
    - Smallest positive denormalized number:

0	00000000	00000000000000000000001

      - $+0.000…01 \times 2^{-126} = 2^{-149}$
    - Negative denormalized range is similar

Floating-point numbers

- IEEE 754 single-precision format
  - Representable magnitudes:
    - 0
    - Denormalized: 2^{-149} to $(1 - 2^{-23}) \times 2^{-126}$
    - Normalized: 2^{-126} to $(2 - 2^{-23}) \times 2^{127}$
    - Infinity
      - Anything with an exponent bigger than 127
      - Note: a mantissa that is larger than the largest representable mantissa (i.e., 1111…1) does not make the magnitude infinity (will discuss this soon)
    - NaN
- Use of IEEE 754 single-precision format
  - The C data type "float" translates to 32-bit single-precision format

Floating-point numbers

- Use of IEEE 754 single-precision format
  - You can verify that the computer uses IEEE 754 single-precision format for float through the following code

```
float x = -3.75;  // Can use any float value
// -3.75 = -11.11 = -1.111 x 2^1 in binary
// biased exponent = 128, mantissa = 111000…0

unsigned int sbit = ((*((unsigned int*)&x)) >> 31) & 1;
unsigned int exponent = ((*((unsigned int*)&x)) >> 23)
                                                & 0xff;
unsigned int fraction = (*((unsigned int*)&x)) & 0x7fffff;
printf("Sign bit: %u, biased exponent: %u, actual
exponent: %d, mantissa: %#x\n", sbit, exponent,
exponent – 127, fraction);
```

Floating-point numbers

- IEEE 754 double-precision format
  - Improves both precision and range over the single-precision format
  - 64-bit representation
    - MSB is sign bit
    - Least significant 52 bits represent mantissa
    - The middle 11 bits represent biased exponent with a bias of 1023
      - Actual exponent can range from -1022 to 1023
  - Largest representable normalized magnitude
    - $1.111…1 \times 2^{1023} = (2 - 2^{-52}) \times 2^{1023}$
  - Smallest representable normalized magnitude
    - $1.000…0 \times 2^{-1022} = 2^{-1022}$

Floating-point numbers

- IEEE 754 double-precision format
  - Largest representable denormalized magnitude
    - $0.111\ldots1 \times 2^{-1022} = (1 - 2^{-52}) \times 2^{-1022}$
  - Smallest representable denormalized magnitude
    - $0.000\ldots01 \times 2^{-1022} = 2^{-1074}$
  - Zero
    - Exponent = 0, Mantissa = 0
  - Infinity
    - Exponent = 11111111111, Mantissa = 0
  - NaN
    - Exponent = 11111111111, Mantissa = non-zero
- Use of IEEE 754 double-precision format
  - The C data type "double" translates to 64-bit double-precision format

Floating-point numbers

- Use of IEEE 754 double-precision format
  - You can verify that the computer uses IEEE 754 double-precision format for double through the following code

```
double x = -3.75;  // Can use any float value
unsigned int sbit = ((*((unsigned long long*)&x)) >> 63) & 1;
unsigned int exponent = ((*((unsigned long long*)&x)) >> 52) & 0x7ff;
unsigned long long fraction = (*((unsigned long long*)&x)) & 0xfffffffffffff;
printf("Sign bit: %u, biased exponent: %u, actual exponent: %d, mantissa: %#llx\n", sbit, exponent, exponent – 1023, fraction);
```

Floating-point numbers

- IEEE 754 half-precision format
  - Lower range and precision compared to single-precision for computers having narrow busses/operands
  - 16-bit representation
    - MSB is sign bit
    - Least significant ten bits represent mantissa
    - The middle five bits represent the biased exponent with a bias of 15
      - Actual exponent can range from -14 to 15
  - Largest representable normalized magnitude
    - $1.1111111111 \times 2^{15} = (2 - 1/1024) \times 2^{15}$
  - Smallest representable normalized magnitude
    - $1.0000000000 \times 2^{-14} = 2^{-14}$

Floating-point numbers

- IEEE 754 half-precision format
  - Largest representable denormalized magnitude
    - $0.1111111111 \times 2^{-14} = (1 - 1/1024) \times 2^{-14}$
  - Smallest representable denormalized magnitude
    - $0.000\ldots01 \times 2^{-14} = 2^{-24}$
  - Zero
    - Exponent = 0, Mantissa = 0
  - Infinity
    - Exponent = 11111, Mantissa = 0
  - NaN
    - Exponent = 11111, Mantissa = non-zero

Floating-point numbers

- IEEE 754 quadruple-precision format
  - Much higher range and precision compared to double-precision
  - No computer supports it yet
  - 128-bit representation
    - MSB is sign bit
    - Least significant 112 bits represent mantissa
    - The middle 15 bits represent the biased exponent with a bias of 16383
      - Actual exponent can range from -16382 to 16383
  - Largest representable normalized magnitude
    - $1.111\ldots1 \times 2^{16383} = (2 - 2^{-112}) \times 2^{16383}$
  - Smallest representable normalized magnitude
    - $1.000\ldots0 \times 2^{-16382} = 2^{-16382}$

Floating-point numbers

- IEEE 754 quadruple-precision format
  - Largest representable denormalized magnitude
    - $0.111\ldots1 \times 2^{-16382} = (1 - 2^{-112}) \times 2^{-16382}$
  - Smallest representable denormalized magnitude
    - $0.000\ldots01 \times 2^{-16382} = 2^{-16494}$
  - Zero
    - Exponent = 0, Mantissa = 0
  - Infinity
    - Exponent = 111…1, Mantissa = 0
  - NaN
    - Exponent = 111…1, Mantissa = non-zero

Overflow in floating-point ops

- What happens if the magnitude of a floating-point number exceeds the largest representable by the corresponding format?
  - E.g., float $x > (2 - 2^{-23}) \times 2^{127}$
  - May cause an overflow if exponent is larger than maximum allowed (e.g., 127 in single-precision)
    - The number is treated as +infinity or –infinity depending on the sign of the number
  - What if the fraction cannot fit within the mantissa field, but the exponent is within range?
    - Not an overflow
    - Handled by rounding the mantissa (default is round to nearest and round to nearest even for halfway round)
    - Loss of precision

Underflow in floating-point ops

- What happens if the magnitude of a floating-point number is less than the smallest representable by the corresponding format?
  - E.g., float $x < 2^{-149}$
  - May cause an underflow if an exponent smaller than the minimum (e.g., -126 in single-precision) is required to represent the number
    - E.g., 2^{-150} in single-precision or 2^{-1075} in double-precision
    - The number is treated as zero in this case
    - Denormalized numbers are said to undergo gradual underflow as more and more leading zeros appear on the right side of the binary point ultimately becoming zero below 2^{-149}

Rounding modes

- IEEE 754 rounding modes
  - Round to nearest (default behavior)
    - Round to nearest even for halfway rounding
      - After rounding, the least significant representable mantissa bit should be even (see following single-precision examples)
    - Example: 1.1111…1 (has 24 1s in mantissa) is rounded to 2.0 (in decimal) i.e., 1.0 x 2^1 in binary
    - Example: 1.111…101 (has 22 1s followed by a 0 and a 1 in mantissa) is rounded to 1.111…10 (has 22 1s followed by a 0 in mantissa)
  - Round toward zero
  - Round toward +infinity
  - Round toward –infinity
- A computer can choose one of the modes

Rounding modes

- IEEE 754 rounding modes
  - Round to nearest (default behavior)
    - How to recognize halfway/non-halfway rounding?
      - Consider *n* = 1.111…1001 (has 22 1s in mantissa followed by 001)
      - Lower nearest neighbor (LNN) is found by zeroing out the overflown bits of the mantissa i.e., LNN = 1.111…1000 (has 22 1s in mantissa followed by 000)
      - Upper nearest neighbor (UNN) is found by adding 0.000…01 (22 0s followed by a 1; pad zeros to the right if needed) to LNN i.e., UNN = 1.111…1100 (has 23 1s in mantissa followed by 00)
      - The halfway mark is LNN + (UNN – LNN)/2 i.e., 1.111…1010 (has 22 1s in mantissa followed by 010)
      - Since the given number *n* is below the halfway mark, it will be rounded to LNN i.e., 1.111…10 (has 22 1s in mantissa followed by a 0)

Rounding modes

- IEEE 754 rounding modes
  - Round to nearest (default behavior)
    - How to recognize halfway/non-halfway rounding?
      - Consider *n* = 1.111…1010 (has 22 1s in mantissa followed by 010)
      - Lower nearest neighbor (LNN) is found by zeroing out the overflown bits of the mantissa i.e., LNN = 1.111…1000 (has 22 1s in mantissa followed by 000)
      - Upper nearest neighbor (UNN) is found by adding 0.000…01 (22 0s followed by a 1; pad zeros to the right if needed) to LNN i.e., UNN = 1.111…1100 (has 23 1s in mantissa followed by 00)
      - The halfway mark is LNN + (UNN – LNN)/2 i.e., 1.111…1010 (has 22 1s in mantissa followed by 010)
      - Since the given number *n* is equal to the halfway mark, it will be rounded to LNN i.e., 1.111…10 as the least significant mantissa bit that can be accommodated is even in LNN