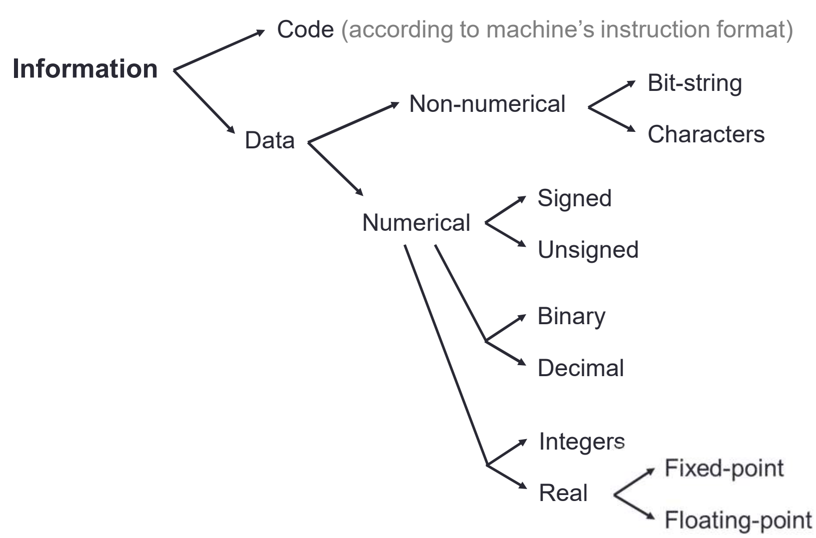
Computer Structure and Language

Hamid Sarbazi-Azad
Department of Computer Engineering
Sharif University of Technology (SUT)
Tehran, Iran

Data Representation



Binary Number Systems

- Data are represented in computers as a sequence of bits (*b*inary dig*its*).
- Other units
  - Byte: 8 bits
  - Nibble: 4 bits (rarely used now)
  - Word: Multiple of bytes (e.g. 2 bytes, 4 bytes, etc.) depending on the computer architecture
- In Binary system *n* bits can represent up to 2^n values
  - 2 bits can represent up to 4 values (00, 01, 10, 11)
  - 4 bits can represent up to 16 values (0000, 0001, 0010, …., 1111)
- To represent *M* values, $\lceil \log_2 M \rceil$ bits required
  - 32 values require 5 bits; 1000 values require 10 bits

Decimal (base-10) Number System

- A weighted-positional number system
- Base (also called radix) is 10
- Symbols/digits = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
- Each position has a weight of power of 10

Example:

$$(7594.36)_{10} = (7 \times 10^3) + (5 \times 10^2) + (9 \times 10^1) + (4 \times 10^0) + (3 \times 10^{-1}) + (6 \times 10^{-2})$$

For a decimal number with (n+m) digits (n digits for integer part and m digits for fractional part), we can write:

$$(a_{n-1}a_{n-2}\ldots a_0 . f_1f_2 \ldots f_m)_{10} = (a_{n-1} \times 10^{n-1}) + (a_{n-2} \times 10^{n-2}) + \ldots + (a_0 \times 10^0) + (f_1 \times 10^{-1}) + (f_2 \times 10^{-2}) + \ldots + (f_m \times 10^{-m})$$

Other Number Systems

- Octal (base 8)

  Weights in powers of 8; Octal digits: **0, 1, 2, 3, 4, 5, 6, 7**.

- Hexadecimal (base 16)

  Weights in powers of 16; Hexadecimal digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**.

- Base/radix *R*

  Weights in powers of *R;* R-base digits: **0, 1, 2, …, *R*-2, *R*-1**

- In some languages/software, special notations are used to represent numbers in certain bases. For example:
  - C language: prefix 0x for hexadecimal (e.g.: 0xF2 represents hexadecimal number $(F2)_{16}$.
  - Verilog: 8'b11110000, 8'hF0, 8'd240, all represent a an 8-bit binary value 11110000.

Base-*R* to Decimal Conversion

Easy! For number Base-R number $a_{n-1} \ldots a_1 a_0 . a_{-1} a_{-2} \ldots a_{-m}$

just calculate $\sum_{k=-m}^{n-1} R^k a_k$.

Examples:

- ❑ $1101.101_2 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} = 13.625_{10}$
- ❑ $572.6_8 = 5\times8^2 + 7\times8^1 + 2\times8^0 + 6\times8^{-1} = 378.75$
- ❑ $2A.8_{16} = 2\times16^1 + 10\times16^0 + 8\times16^{-1} = 42.5$
- ❑ $341.24_5 = 3\times5^2 + 4\times5^1 + 1\times5^0 + 2\times5^{-1} + 4\times5^{-2} = 96.56$

Decimal to Base-R Conversion

- For whole numbers
  - Repeated Division-by-R Method
- For fractions
  - Repeated Multiplication-by-R Method

Repeated Divison-by-2

To convert a non-fractional number to binary, use successive division by 2 until the quotient is 0. The remainders form the answer, with the first remainder as the *least significant bit (LSB)* and the last as the *most significant bit (MSB)*.

$(43)_{10} = (\ 101011\ )_2$

2	43	
2	21 rem 1	← LSB
2	10 rem 1	
2	5 rem 0	
2	2 rem 1	
2	1 rem 0	
	0 rem 1	← MSB

(c) Hamid Sarbazi-Azad Computer Structure and Language -- Lecture #4: Data Representation 9

Repeated Multiplication-by-2

To convert a fractional number to binary, repeated multiplication by 2 is used, until the fractional product is 0 (or until the desired number of decimal places). The carried digits, or *carries*, produce the answer, with the first carry as the MSB, and the last as the LSB.

$(0.3125)_{10} = (\ .0101\ )_2$

	Carry	
0.3125×2=0.625	0	←MSB
0.625×2=1.25	1	
0.25×2=0.50	0	
0.5×2=1.00	1	←LSB

(c) Hamid Sarbazi-Azad Computer Structure and Language -- Lecture #4: Data Representation 10

Conversion Between Bases

In general, conversion between bases can be done as: Base-i → Decimal → Base-j

Shortcuts for conversion between bases of powers of 2 (e.g. 2, 4, 8, 16): Base-2^i → Base-2^j

Step 1. Write down digits of the input number each in i bits.

Step 2. From right side, partition the bits in groups of *j* bits.

Step 3. Write down the equivalent of each group in Base-j notation.

Example: Convert $(7452)_8$ to hexadecimal.

7452_8 → 111 100 101 010 →

111 100 101 010 →

$F2A_{16}$

Negative Numbers

Unsigned numbers: only non-negative values

Signed numbers: include all values (positive and negative)

There are 4 representations for signed binary numbers:

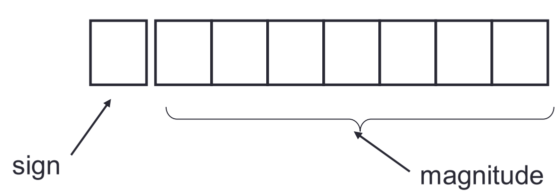
- Sign-Magnitude
- 1s-Complement
- 2s-Complement
- Excess-Number (not much popular but used for some cases)

Sign-Magnitude

The sign is represented by a 'sign bit'

- 0 for +
- 1 for -

An 8-bit (1-bit sign and 7-bit magnitude) format.



Example:

00110100 ➔ $+110100_2 = +52_{10}$

10010011 ➔ $-10011_2 = -19_{10}$

Sign-Magnitude

- Largest value (8 bits): 01111111 = $+127_{10}$
- Smallest value (8 bits): 11111111 = -127_{10}
- Zeros (8 bits): 00000000 = $+0_{10}$
  10000000 = -0_{10}
- Range (8-bit): -127_{10} to $+127_{10}$

Question: For an *n*-bit sign-magnitude representation, what is the range of values that can be represented?

Answer: $-(2^{n-1}-1)$ to $2^{n-1}-1$

Sign-Magnitude

To negate a number, just invert the sign bit.

Examples:

- How to negate 00100001_{sm} (decimal 33)?
  Answer: 10100001_{sm} (decimal -33)
- How to negate 10000101_{sm} (decimal -5)?
  Answer: 00000101_{sm} (decimal +5)

Sign-Magnitude Computation

Addition: To compute S = A + B where A (A_s | A_m) and B (B_s | B_m), represented in sign-magnitude format, we need to consider different conditions:

if A_s=B_s then return S (A_s | A_m+B_m)
else
    if A_s=1 & A_m<B_m then return S (0 | B_m- A_m)
    if A_s=1 & A_m>B_m then return S (1 | A_m- B_m)
    if A_s=0 & A_m<B_m then return S (1 | B_m- A_m)
    if A_s=0 & A_m>B_m then return S (0 | A_m- B_m)

Sign-Magnitude Computation

Subtraction: To compute S = A - B where A (A_s | A_m) and B (B_s | B_m), represented in sign-magnitude format, we need to consider different conditions:

if $A_s \neq$ B_s then return S (A_s | A_m+B_m)
else
    if A_s=1 & A_m<B_m then return S (0 | B_m- A_m)
    if A_s=1 & A_m>B_m then return S (1 | A_m- B_m)
    if A_s=0 & A_m<B_m then return S (1 | B_m- A_m)
    if A_s=0 & A_m>B_m then return S (0 | A_m- B_m)

Sign-Magnitude Computation

Multiplication: To compute S = A * B where A (A_s | A_m) and B (B_s | B_m), represented in sign-magnitude format, we have:

S (A_s <u>xor</u> B_s | A_m*B_m)

Division: To compute S = A / B where A (A_s | A_m) and B (B_s | B_m), represented in sign-magnitude format, we have:

S (A_s <u>xor</u> B_s | A_m/B_m)

Sign-magnitude computation (specifically addition/ subtraction) is complex and requires more hardware with respect to others (1s- and 2s-complement)!

1s-Complement

Given a number ***x*** which can be expressed as an *n*-bit binary number, its <u>negated value</u> can be obtained in **1s-Complement** representation as:

$$-x = 2^n - x - 1$$

Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 1s-complement is:

-00001100_2 $= 2^8 - 12 - 1$ (calculation done in decimal)
= 243
= 11110011_2

This means that -12_{10} is written as 11110011 in 8-bit 1s-complement representation.

1s-Complement

- Technique to negate a value: invert all the bits.
- Largest value (8 bits): 01111111 = $+127_{10}$
- Smallest value(8 bits): 10000000 = -127_{10}
- Zeros (8 bits): 00000000 = $+0_{10}$
  11111111 = -0_{10}
- Range (8 bits): -127_{10} to $+127_{10}$
- Range (n bits): $-(2^{n-1} – 1)$ to $2^{n-1} – 1$

The most significant bit (MSB) still represents the sign, i.e. 0 for +, 1 for -.

1s-Complement for Addition/Subtraction

Algorithm for addition of integers, A + B:

1. Perform binary addition on the two numbers.
2. Add the carry out of the MSB to the result.
3. Check for overflow. Overflow occurs if result is opposite sign of A and B.

- If the result of addition/subtraction goes beyond this range, an **overflow** occurs.
- Overflow can be easily detected:
  - *positive* + *positive* → *negative*
  - *negative* + *negative* → *positive*

Algorithm for subtraction of integers, A – B:

A – B = A + (-B) = A + 1s-Complement (B)

1. Take 1s-complement of B.
2. Add the 1s-complement of B to A.

1s Complement Addition

Examples: 4-bit system

```
  +3        0011
+ +4      + 0100
----      -------
  +7        0111
```

No overflow

```
  +5        0101
+ -5      + 1010
----      -------
  -0        1111
```

No overflow

```
  -2        1101
+ -5      + 1010
----      -------
  -7       10111
          +     1
          -------
            1000
```

No overflow

```
  -3        1100
+ -7      + 1000
----      -------
 -10       10100
          +     1
          -------
            0101
```

Overflow!

Any overflow?

2s-Complement

Given a number ***x*** which can be expressed as an *n*-bit binary number, its negated value can be obtained in **2s-complement** representation using:

$$-x = 2^n - x = \text{1s-complement}(x) + 1$$

Example: With an 8-bit number 00001100 (or 12_{10}), its negated value expressed in 2s-complement is:

$-00001100_2 = 2^8 - 12$ (calculation done in decimal)
$= 244$
$= 11110100$

This means that -12_{10} is written as 11110100 in 8-bit 2s-complement representation.

2s-Complement

Technique to negate a value: invert all the bits, then add 1. Alternatively, from right to left, write down the bits until the first 1 bit, then invert the remaining bits.

- Largest value (8 bits): 01111111 = $+127_{10}$
- Smallest value (8 bits): 10000000 = -128_{10}
- Zero (8 bits): 00000000 = $+0_{10}$
- Range (8 bits): -128_{10} to $+127_{10}$
- Range (*n* bits): -2^{n-1} to $2^{n-1} - 1$

The most significant bit (MSB) still represents the sign: 0 for +, 1 for -.

2s-Complement for Addition/Subtraction

Algorithm for addition, A + B:

1. Perform binary addition on the two numbers.
2. Ignore the final carry out of the MSB.
3. Check for overflow. Overflow occurs if the ‘carry in’ and ‘carry out’ of the MSB are different, or if result has the opposite sign of both A and B.

Algorithm for subtraction, A – B:

A – B = A + (-B) = A + 2s-Complement (B)

1. Take 2s-complement of B.
2. Add the result to A.

2s-Complement Addition

Examples: 4-bit system

```
   +3        0011
 + +4      + 0100
 ----      -------
   +7        0111
```

No overflow

```
   -2        1110
 + -6      + 1010
 ----      -------
   -8       11000
```

No overflow

```
   +6        0110
 + -3      + 1101
 ----      -------
   +3       10011
```

No overflow

```
   +4        0100
 + -7      + 1001
 ----      -------
   -3        1101
```

No overflow

```
   -3        1101
 + -6      + 1010
 ----      -------
   -9       10111
```

Overflow!

```
   +5        0101
 + +6      + 0110
 ----      -------
  +11        1011
```

Overflow!

Which of the above is/are overflow(s)?

2s-Complement Subtraction

Examples: 4-bit system

- 4 – 7
- Convert it to 4 + (-7)

```
   +4        0100
 + -7      + 1001
 ----      -------
   -3        1101
```

No overflow

- 6 – 1
- Convert it to 6 + (-1)

```
   +6        0110
 + -1      + 1111
 ----      -------
   +5       10101
```

No overflow

- -5 – 4
- Convert it to -5 + (-4)

```
   -5        1011
 + -4      + 1100
 ----      -------
   -9       10111
```

Overflow!

Which of the above is/are overflow(s)?

Comparisons

For 4-bit numbers

Positive values

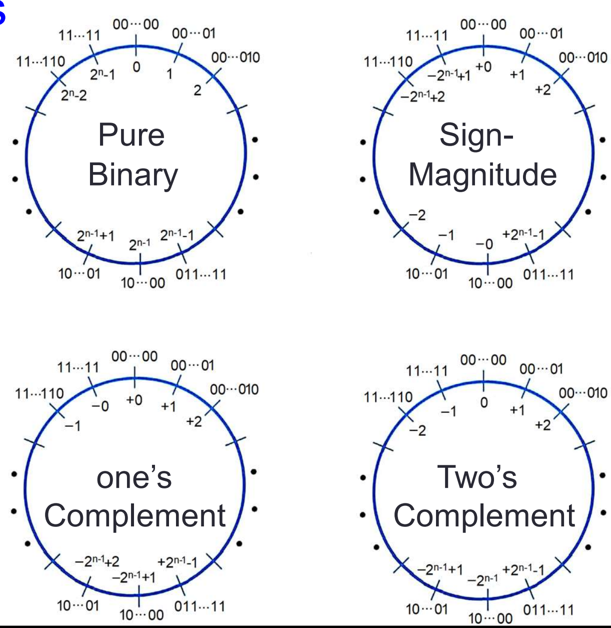
Value	Sign-and-Magnitude	1s Comp.	2s Comp.
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000

Negative values

Value	Sign-and-Magnitude	1s Comp.	2s Comp.
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Comparisons

Modular presentation of n-bit binary numbers



Homework #6

1. Prove that adding two A and B in 1s-complement system using a binary adder and then adding the generated carry (round around carry) to the result, will produce A+B in 1s-complement system.

Note: Consider all possible combinations of A and B and then show that the result is correct.

Example: A<0, B<0 ➔ $2^n-|A|-1 + 2^n-|B|-1 = 2^n+2^n-(|A|+|B|)-2$.

Now, by adding the round around carry (2^n) to the result we have

$2^n-(|A|+|B|)-2+1 = 2^n-(|A|+|B|)-1$ which is OK.

2. Prove that adding two A and B in 2s-complement system using a binary adder will produce A+B in 2s-complement system.

Note: Consider all possible combinations of A and B and then show that the result is correct.

Example: A<0, B<0 ➔ $2^n-|A| + 2^n-|B| = 2^n+2^n-(|A|+|B|)$ which is OK.

Complement on Fractions

We can extend the idea of complement on fraction part too.

Examples:

- Negate 01011.011 in Sign-Magnitude
  Answer: 11011.011
- Negate 11000.010 in 1s-complement
  Answer: 00111.101
- Negate 01001.011 in 2s-complement
  Answer: 10110.101

Excess Representation

Beside sign-magnitude and complement schemes, the **excess representation** is another scheme.

It allows the range of values to be distributed evenly between the positive and negative values, by a simple level transition (addition or subtraction).

Example:

Excess-4 representation on 3-bit numbers.

Excess-4 Representation	Value
000	-4
001	-3
010	-2
011	-1
100	0
101	1
110	2
111	3

Excess Representation

In general, for **evenly** distribution of code values, we may use Excess-2^{n-1} or Excess-$(2^{n-1}-1)$ codes.

Example:

For 4-bit numbers, we may use Excess-7 or Excess-8 codes.

Excess-7 is shown here.

Excess-7 Representation	*Value*
0000	-7
0001	-6
0010	-5
0011	-4
0100	-3
0101	-2
0110	-1
0111	0
1000	1
1001	2
1010	3
1011	4
1100	5
1101	6
1110	7
1111	8

Excess Representation

In general, for **evenly** distribution of code values, we may use Excess-2^{n-1} or Excess-(2^{n-1}-1) codes.

Other Excess values will result in **unevenly** code distribution.

Example:

For 4-bit numbers, we may use Excess-7 or Excess-8 codes for **evenly** code distribution.

Excess-5 is an **unevenly** distributed code.

Representation			Value
Excess-8	*Excess-7*	*Excess-5*	
0000	—	—	-8
0001	0000	—	-7
0010	0001	—	-6
0011	0010	0000	-5
0100	0011	0001	-4
0101	0100	0010	-3
0110	0101	0011	-2
0111	0110	0100	-1
1000	0111	0101	0
1001	1000	0110	1
1010	1001	0111	2
1011	1010	1000	3
1100	1011	1001	4
1101	1100	1010	5
1110	1101	1011	6
1111	1110	1100	7
—	1111	1101	8
—	—	1110	9
—	—	1111	10

Real Numbers

- Many applications involve computations not only on integers but also on real numbers.
- How are real numbers represented in a computer system?

  Due to the finite number of bits, real number are often represented in their approximate values.
- Real numbers can be represented in two forms:

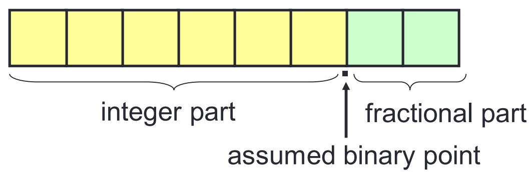
  - Fixed-point

  - Floating-point

Fixed-Point Representation

In fixed-point representation, the number of bits allocated for the integer part and fractional part are fixed.

Example: Given an 8-bit representation (6 bits for integer part and 2 bits for fractional part), we have:



If 2s-complement is used, we can represent values like:

$011010.11 \quad = 26.75_{10}$

$111110.11 \quad = -000001.01_2 \quad = -1.25_{10}$

Floating-Point Representation

Fixed-point representation has limited range.

Alternative: Floating-point numbers allow us to represent very large and very small numbers.

Examples:

$+ 0.23 \times 10^{23}$ (very large positive number)

$+ 0.5 \times 10^{-37}$ (very small positive number)

$- 0.2397 \times 10^{-18}$ (very small negative number)

Floating-Point Representation

A floating-point number comprises of 2 parts: mantissa (fraction) and exponent.

mantissa	exponent

value = mantissa x $b^{Exponent}$

- In most systems, the base (radix) is assumed to be b=2 (IBM360/370 uses b=16).
- Although we can use any representation for the Manitissa and Exponent parts, the widely-used and accepted format use: Mantissa of m bits in Sign-Magnitude format and
  Exponent of e bits in Excess-2^{e-1} (Biased) format.

Sign bit

S	M	E
	mantissa	exponent

Value = $(-1)^S$ x 0.M x 2^{E-Bias}

Floating-Point Representation

Example: Consider a 12-bit floating-point number format.

S	M	E
1 bit	7 bits	4 bits

Value = $(-1)^S$ x 0.M x 2^{E-8}

Largest number = 011111111111 = +0.1111111x2^{15-8} = 1111111_2 = 127

Smallest number = 111111111111 = - 0.1111111x2^{15-8} = -127

Largest negative number = 100000010000 = -2^{-7}X2^{0-8} = -2^{-15} = -0.0000305176

Smallest positive number = 000000010000 = 2^{-7}X2^{0-8} = 2^{-15} = 0.0000305176

Number -3.625 = -11.101_2 = 111101001010 or 101110101011
Normalized or 100111011100

Number 25.375 = +11001.01$\boxed{1}$ = 011001011101 which shows 25.25 !!!

Note that the LSB of Mantissa (valued 0.125 here) is truncated.

Floating-Point Addition/Subtraction

S	M	E
1 bit	7 bits	4 bits

Value = $(-1)^S \times 0.M \times 2^{E-8}$

In above representation system, add 11.5_{10} and 23.25_{10}.

11.5_{10} = 010111001100 23.25_{10} = 010111011101

In order to add the two mantissas, we need to align the two numbers by equalizing the exponents of the two numbers. To save MSB of 23.35, we shift mantissa of 11.5 one bit to the right (that divides it by 2) and hence add 1 to its exponent (to multiply it by 2, so keeping the value of number 11.5 unchanged). Hence, we have 11.5 = 001011101101.

Now, the exponents of the two number are equal and we can add their mantissas. So, we have: 0101110 + 1011101 = 10001011

To fit the resulted mantissa in 7 bits we have to truncate the LSB (=1 here) and add 1 to the exponent 1101. So, we have the exponent of 1110.

Therefore, final result is 010001011110 that is $+34.5_{10}$ and NOT $+34.75_{10}$!

Floating-Point Addition/Subtraction

S	M	E
1 bit	7 bits	4 bits

Value = $(-1)^S \times 0.M \times 2^{E-8}$

Example: Add floating-point numbers 0.6015625 and 12.375.
Using pen-and-paper, we have: 0.6015625 + 12.375 = 12.9765625.

Let us add them in our floating-point system:

0.6015625 = 010011010111
12.375 = 011000111100

We need to align numbers:

0.6015625 = 010011010111 → 000000101100
+ 011000111100
011001011100 = 12.625 !!!

In floating-point add/sub, smaller number is sacrificed for larger number (the penalty is bigger if the exponents difference is higher).

Floating-Point Addition/Subtraction

S	M	E
1 bit	7 bits	4 bits

Value = $(-1)^S \times 0.M \times 2^{E-8}$

Example: We have four registers R0 (=10.75), R1 (= -12.25), R2(= -23.0) and R3 (= 0.248046875) and the following code is executed:

```
FSUB  R0,R1
FADD  R2,R0
FADD  R3,R2
```

R0	R1	R2	R3
010101101100	111000101100	110111001101	011111110110
$=10.75_{10}$	$= -12.25_{10}$	$= -23.0_{10}$	$=0.248046875_{10}$

FSUB R0,R1 → (R0) = 010111001101 = 23.0_{10}
FADD R2,R0 → (R2) = 000000001101 = 0.0_{10}
FADD R3,R2 → (R3) = 000000001101 = 0.0_{10} !!!

Floating-Point Addition/Subtraction

S	M	E
1 bit	7 bits	4 bits

Value = $(-1)^S \times 0.M \times 2^{E-8}$

Example: We have four registers R0 (=10.75), R1 (= -12.25), R2(= -23.0) and R3 (= 0.248046875) and the following code is executed:

```
FSUB  R0
FADD  R2
FADD  R3
```

R0
010101101100
$=10.75_{10}$

```
FSUB  R0
FADD  R2
FADD  R3
```

To resolve this issue, we give the smallest possible exponent to floating-point zero, i.e.

000000000000 = 0.0×2^{-8}

So, when added with any number, mantissa of zero is shifted → **no error** ☺

Floating-Point Representation

Question 1: Normalization

What are the benefits of normalizing floating-point numbers?

1. Best using the allocated bits for mantissa
2. Making the representation of each number unique.

Question 2: Smallest exponent for 0.0

Why should we consider the smallest possible exponent for floating-point zero?

To minimize computations error.

Floating-Point Representation

Question 3: Biased exponent

What are the benefits of having biased exponent?

1. Having a pattern of zero bits to represent float-point number 0.0
2. Exponents comparison will be simpler (less complexity)
3. Checking for a floating-point number being zero/non-zero can be done by the circuit used for integer numbers
4. No need to have additional opcodes for conditional jump on zero/not-zero instructions for floating-point numbers

What are the negative points of biased exponent?

1. Additional bias in floating-point multiplication
2. Removed bias in floating-point division

IEEE 754 Floating-Point Representation

- 3 fields: **sign**, **exponent** and **fraction** (mantissa's magnitude)

S	E	F

Value = $(-1)^S \times 1.F \times 2^{E-Bias}$

- Two formats:
  - Single-precision (32 bits)
    1-bit sign, 8-bit exponent (bias=127) and 23-bit fraction
  - Double-precision (64 bits)
    1-bit sign, 11-bit exponent (bias= 1023) and 52-bit fraction
- Mantissa is **normalized** with an implicit leading bit 1.
  - 110.1_2 → normalized → $1.101_2 \times 2^2$ → only **101** is stored in the fraction field
  - 0.00101101_2 → normalized → $1.01101_2 \times 2^{-3}$ → only **01101** is stored in the fraction field

IEEE 754 Floating-Point Representation

Example: How is -6.5_{10} represented in IEEE 754 single-precision floating-point format?

$$-6.5_{10} = -110.1_2 = -1.\boxed{101}_2 \times 2^{2}$$

Exponent = 2 + 127 = 129 = 10000001_2

1	10000001	10100000000000000000000
sign	exponent (excess-127)	fraction

We may write the 32-bit representation in hexadecimal as:

$1\ 10000001\ 10100000000000000000000_2$ = $C0D00000_{16}$

IEEE 754 Floating-Point Representation

Floating Point Range

	Denormalized	Normalized	Approximate Decimal
Single Precision	$\pm 2^{-149}$ to $(1-2^{-23})\times 2^{-126}$	$\pm 2^{-126}$ to $(2-2^{-23})\times 2^{127}$	$\pm \approx 10^{-44.85}$ to $\approx 10^{38.53}$
Double Precision	$\pm 2^{-1074}$ to $(1-2^{-52})\times 2^{-1022}$	$\pm 2^{-1022}$ to $(2-2^{-52})\times 2^{1023}$	$\pm \approx 10^{-323.3}$ to $\approx 10^{308.3}$

Special values: when exponent field's bits are all 0s or all 1s.

1. **Denormalized number**. When the exponent is all 0s, then the value is denormalized, and the value of the FP number is: $(-1)^s$x 0.f x 2^{-126} in single precision format and $(-1)^s$x 0.f x 2^{-1022} in double precision format.
2. **Zero**. A denormalized number where f is all 0s. We have +0 and -0 which compare equal.

IEEE 754 Floating-Point Representation

Special values: when exponent field's bits are all 0s or all 1s.

1. **Denormalized number**. When the exponent is all 0s, then the value is denormalized, and the value of the FP number is:
   $(-1)^s$x 0.f x 2^{-126} in single precision format and
   $(-1)^s$x 0.f x 2^{-1022} in double precision format.
2. **Zero**. A denormalized number where fraction is all 0s. We have +0 and -0 which compare equal.
3. **Infinity**. The values $+\infty$ and $-\infty$ are denoted with an exponent of all 1s and a fraction of all 0s.
4. **Not A Number (NaN)**. The value NaN is used to represent a value that does not represent a real number. NaN's are represented by a bit pattern with an exponent of all 1s and a non-zero fraction.

(c) Hamid Sarbazi-Azad — Computer Structure and Language -- Lecture #4: Data Representation — 49

IEEE 754 FP Representation

Arithmetic operations with special values

Operation	Result
n ÷ ±∞	0
±∞ × ±∞	±∞
±*nonZero* ÷ ±0	±∞
±*finite* × ±∞	±∞
∞ + ∞ ∞ − −∞	+∞
−∞ − ∞ −∞ + −∞	−∞
∞ − ∞ −∞ + ∞	*NaN*
±0 ÷ ±0	*NaN*
±∞ ÷ ±∞	*NaN*
±∞ × 0	*NaN*
NaN == *NaN*	*false*

(c) Hamid Sarbazi-Azad — Computer Structure and Language -- Lecture #4: Data Representation — 50

IEEE 754 FP Representation

Summary

Sign	Exponent (e)	Fraction (*f*)	Value
0	00⋯00	00⋯00	+0
0	00⋯00	00⋯01 ⋮ 11⋯11	Positive Denormalized Real $0.f \times 2^{(-b+1)}$
0	00⋯01 ⋮ 11⋯10	XX⋯XX	Positive Normalized Real $1.f \times 2^{(e-b)}$
0	11⋯11	00⋯00	+∞
0	11⋯11	00⋯01 ⋮ 11⋯11	NaN
			ed Real
1	⋮ 11⋯10	XX⋯XX	Negative Normalized Real $-1.f \times 2^{(e-b)}$
1	11⋯11	00⋯00	−∞
1	11⋯11	00⋯01 ⋮ 11⋯11	NaN

For more information see:
http://steve.hollasch.net/cgindex/coding/ieeefloat.html

Decimal Numbers

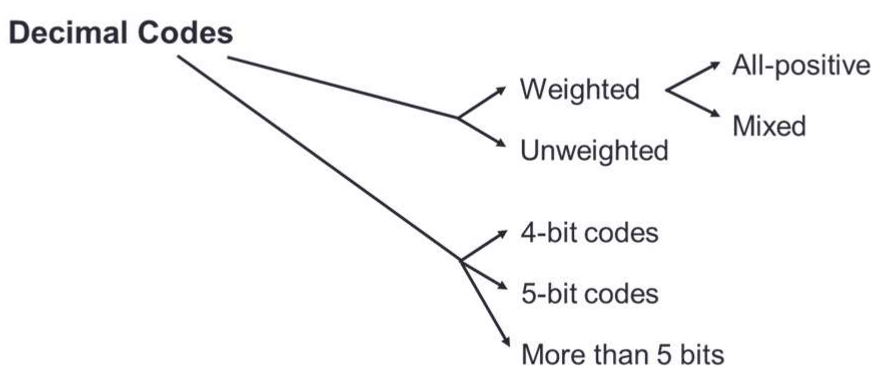
The famous decimal number representation format is called BCD (Binary Coded Decimal) format, where binary equivalent of each digit is used.

Example: What is the BCD format for number 7089652?

7089652

0111 0000 1000 1001 0110 0101 0010

Decimal Number Representation



Decimal Number Representation

Some 4-bit decimal codes

Decimal Digit	8421 (BCD)	Excess 3	5421	84-2-1	7421	5311	631-1	2421
0	0000	0011	0000	0000	0000	0000	0011	0000
1	0001	0100	0001	0111	0001	0001	0010	0001
2	0010	0101	0010	0110	0010	0011	0101	0010
3	0011	0110	0011	0101	0011	0100	0111	0011
4	0100	0111	0100	0100	0100	0101	0110	0100
5	0101	1000	0101	1011	0101	1000	1001	1011
6	0110	1001	0110	1010	0110	1001	1000	1100
7	0111	1010	0111	1001	1000	1011	1010	1101
8	1000	1011	1011	1000	1001	1100	1101	1110
9	1001	1100	1100	1111	1010	1101	1100	1111

Self-complement code: where 9s-complement is resulted by bit-wise not!
Excess-3, 631-1 and 2421 codes are self-complement.

Decimal Number Representation

Some 5-bit decimal codes

Decimal Digit	2-out-of-5	63210	Shift-counter	86421	51111
0	00011	00001	00000	00000	00000
1	00101	00011	00001	00001	00001
2	00110	00101	00011	00010	00011
3	01001	01001	00111	00011	00111
4	01010	01010	01111	00100	01111
5	01100	01100	11111	00101	10000
6	10001	10001	11110	01000	11000
7	10010	10010	11100	01001	11100
8	10100	10100	11000	10000	11110
9	11000	11000	10000	10001	11111

51111 code is self-complement.

Decimal Number Representation

Some more than 5 bits decimal codes

Decimal Digit	50 43210	543210	9876543210
0	01 00001	000001	0000000001
1	01 00010	000010	0000000010
2	01 00100	000100	0000000100
3	01 01000	001000	0000001000
4	01 10000	010000	0000010000
5	10 00001	100001	0000100000
6	10 00010	100010	0001000000
7	10 00100	100100	0010000000
8	10 01000	101000	0100000000
9	10 10000	110000	1000000000

Decimal Number Representation

Gray code

Decimal Digit	BCD 8421	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101

Character Codes

Used to represent and store alphabetical data.

ASCII (7-bit) code table

	MSBs							
LSBs	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC$_1$	!	1	A	Q	a	q
0010	STX	DC$_2$	“	2	B	R	b	r
0011	ETX	DC$_3$	#	3	C	S	c	s
0100	EOT	DC$_4$	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	‘	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	O	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

‘A’: 1000001 (or 65_{10})

Character Codes

Famous character codes:

ASCII (American Standard Code for Information Interchange): 8-bit code (7 bits + 1 bit parity) that is popular in personal computers

EBCDIC (Extended Binary Coded Decimal Interchange Code): 8-bit IBM standard character code

Unicode (Universal code): uses 8 bits (UFT-8), 16 bits (UFT-16) or 32 bits (UFT-32) to show all characters in all languages. See https://en.wikipedia.org/wiki/Unicode.

END OF SLIDES