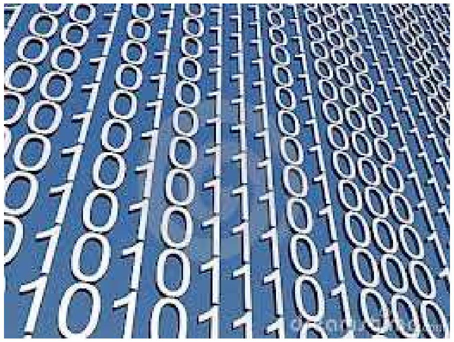
Number Systems and Number Representation



Goals of these Lectures

Help you learn (or refresh your memory) about:

- The binary, hexadecimal, and octal number systems
- Finite representation of unsigned integers
- Finite representation of signed integers
- Finite representation of rational numbers (if time)

Why?

- A power programmer must know number systems and data representation to fully understand C's **primitive data types**

Agenda

Number Systems (Lecture 1)

Finite representation of unsigned integers (Lecture 2)

Finite representation of signed integers (Lecture 3)

Finite representation of rational numbers (Lecture 4)

The Decimal Number System

Name

- “decem” (Latin) => ten

Characteristics

- Ten symbols
  - `0 1 2 3 4 5 6 7 8 9`
- Positional
  - $2945 \neq 2495$
  - $2945 = (2*10^3) + (9*10^2) + (4*10^1) + (5*10^0)$

(Most) people use the decimal number system

The Binary Number System

Name

- "binarius" (Latin) => two

Characteristics

- Two symbols
  - `0 1`
- Positional
  - $1010_B \neq 1100_B$

Most (digital) computers use the binary number system

Terminology

- **Bit**: a binary digit
- **Byte**: (typically) 8 bits

Decimal-Binary Equivalence

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Decimal	Binary
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111
...	...

Decimal-Binary Conversion

Binary to decimal: expand using positional notation

$$
\begin{aligned}
100101_B &= (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0) \\
&= 32 + 0 + 0 + 4 + 0 + 1 \\
&= 37
\end{aligned}
$$

Decimal-Binary Conversion

Decimal to binary: do the reverse

- Determine largest power of 2 ≤ number; write template

$$37 = (?*2^5)+(?*2^4)+(?*2^3)+(?*2^2)+(?*2^1)+(?*2^0)$$

- Fill in template

$$37 = (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$$

```
 37
-32
  5
 -4
  1
 -1
  0
```

100101_B

Decimal-Binary Conversion

Decimal to binary shortcut

- Repeatedly divide by 2, consider remainder

```
37 / 2 = 18 R 1
18 / 2 =  9 R 0
 9 / 2 =  4 R 1
 4 / 2 =  2 R 0
 2 / 2 =  1 R 0
 1 / 2 =  0 R 1
```

Read from bottom to top: 100101_B

The Hexadecimal Number System

Name

- “hexa” (Greek) => six
- “decem” (Latin) => ten

Characteristics

- Sixteen symbols
  - **0 1 2 3 4 5 6 7 8 9 A B C D E F**
- Positional
  - $\mathbf{A13D_H \neq 3DA1_H}$

Computer programmers often use the hexadecimal number system

Decimal-Hexadecimal Equivalence

Decimal	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Decimal	Hex
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F

Decimal	Hex
32	20
33	21
34	22
35	23
36	24
37	25
38	26
39	27
40	28
41	29
42	2A
43	2B
44	2C
45	2D
46	2E
47	2F
...	...

Decimal-Hexadecimal Conversion

Hexadecimal to decimal: expand using positional notation

```
25H = (2*16^1) + (5*16^0)
    =   32     +     5
    =   37
```

Decimal to hexadecimal: use the shortcut

```
37 / 16 = 2 R 5
 2 / 16 = 0 R 2
```

Read from bottom to top: 25_H

Binary-Hexadecimal Conversion

Observation: $16^1 = 2^4$

- Every 1 hexadecimal digit corresponds to 4 binary digits

Binary to hexadecimal

```
1010000100111101B
  A    1    3   DH
```

Digit count in binary number not a multiple of 4 => pad with zeros on left

Hexadecimal to binary

```
  A    1    3   DH
1010000100111101B
```

Discard leading zeros from binary number if appropriate

The Octal Number System

Name

- “octo” (Latin) => eight

Characteristics

- Eight symbols
  - **0 1 2 3 4 5 6 7**
- Positional
  - $1743_O \neq 7314_O$

Computer programmers often use the octal number system

Decimal-Octal Equivalence

Decimal	Octal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17

Decimal	Octal
16	20
17	21
18	22
19	23
20	24
21	25
22	26
23	27
24	30
25	31
26	32
27	33
28	34
29	35
30	36
31	37

Decimal	Octal
32	40
33	41
34	42
35	43
36	44
37	45
38	46
39	47
40	50
41	51
42	52
43	53
44	54
45	55
46	56
47	57
...	...

Decimal-Octal Conversion

Octal to decimal: expand using positional notation

$$37_O = (3*8^1) + (7*8^0)$$
$$= 24 + 7$$
$$= 31$$

Decimal to octal: use the shortcut

```
31 / 8 = 3 R 7
 3 / 8 = 0 R 3
```

Read from bottom to top: 37_O

Binary-Octal Conversion

Observation: $8^1 = 2^3$

- Every 1 octal digit corresponds to 3 binary digits

Binary to octal

```
001010000100111101B
 1  2  0  4  7  5O
```

Digit count in binary number not a multiple of 3 => pad with zeros on left

Octal to binary

```
 1  2  0  4  7  5O
001010000100111101B
```

Discard leading zeros from binary number if appropriate

Agenda

Number Systems (Lecture 1)

Finite representation of unsigned integers (Lecture 2)

Finite representation of signed integers (Lecture 3)

Finite representation of rational numbers (Lecture 4)

Bitwise Operations

Bitwise AND

- Similar to logical AND (`&&`), except it works on a bit-by-bit manner
- Denoted by a single ampersand: `&`

```
(1001 &
 0101)=
 0001
```

Bitwise OR

- Similar to logical OR (||), except it works on a bit-by-bit manner
- Denoted by a single pipe character: |

```
(1001 |
 0101)=
 1101
```

Bitwise XOR

- Exclusive OR, denoted by a carat: `^`
- Similar to bitwise OR, except that if both inputs are `1` or `0` then the result is `0`

```
(1001 ^
 0101)=
 1100
```

Bitwise NOT

- Similar to logical NOT (!), except it works on a bit-by-bit manner
- Denoted by a tilde character: ~

```
~1001 =
 0110
```

Bitwise Operations as Masks

X: it is an unknown binary number and can be either 0 or 1

AND (&) Operation:

X & 0 = 0 & X = **0**
X & 1 = 1 & X = X
X & X = X

OR (|) Operation:

X | 1 = 1 | X = **1**
X | 0 = 0 | X = X
X | X = X

XOR (^) Operation:

X ^ 1 = 1 ^ X = ~X
X ^ 0 = 0 ^ X = X
X ^ X = 0

Mask Example

Specify the mask you would need to isolate bit 0 of the unknown number. The result of the operation should be **0 (0x0000) if bit 0 is 0, and non-zero if bit 0 is 1**. Express it as a 4-digit hexadecimal number.

Answer:

We know that 1 hexadecimal digit = 4 bits in binary

```
                    15...              ...... 3 2 1 0   ← Bit position
                     XXXX  XXXX  XXXX XXXX   ← Unknown number
Operation --> ?      ????  ????  ????  ????   ← Mask
                   ------------------------------------
if bit 0 is 0  →  0000  0000  0000  0000   ← zero (0x0000)
if bit 0 is 1  →  0000  0000  0000   0001   ← nonzero (0x0001)
```

In this case, we can use AND operation (**&**) and then the mask(16 bits) will be as

0000 0000 0000 0001 => 0001 in hexadecimal

Therefore, the answer is answer **& as the operation and 0x0001 as the mask.**

Mask Example

Specify the mask you would need to **set bit 1 of the unknown number to zero**. That is, the result of this operation results in a new number, which the unknown number will be subsequently set to. Express it as a 4-digit hexadecimal number.

Answer:

We know that 1 hexadecimal digit = 4 bits in binary

```
                   15...             ...... 3 2 1 0   ← Bit position
                    XXXX  XXXX  XXXX XXXX    ← Unknown number
     Operation --> ?   ????  ????  ????  ????    ← Mask
                 -----------------------------------
                    XXXX  XXXX  XXXX  XX0X
```

In this case, we can use AND operation (**&**) and then the mask(16 bits) will be as

1111 1111 1111 1101 => FFFD in hexadecimal

Therefore, the answer is **& as the operation and 0xFFFD as the mask.**

Unsigned Data Types: Java vs. C

Java has type

- **`int`**
  - Can represent signed integers

C has type:

- **`signed int`**
  - Can represent signed integers
- **`int`**
  - Same as **`signed int`**
- **`unsigned int`**
  - Can represent only unsigned integers

To understand C, must consider representation of both unsigned and signed integers

Representing Unsigned Integers

Mathematics

- Range is 0 to ∞

Computer programming

- Range limited by computer's **word** size
- Word size is n bits => range is 0 to $2^n - 1$
- Exceed range => **overflow**

Nobel computers with gcc217

- $n = 32$, so range is 0 to $2^{32} - 1$ (4,294,967,295)

Pretend computer

- $n = 4$, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4

- All points generalize to word size = 32, word size = n

Representing Unsigned Integers

On pretend computer

Unsigned Integer	Rep
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Adding Unsigned Integers

Addition

```
                 1
   3           0011B
+ 10         + 1010B
  --           ----
  13           1101B
```

Start at right column

Proceed leftward

Carry 1 when necessary

```
                11
   7           0111B
+ 10         + 1010B
  --           ----
   1          10001B
```

Beware of overflow

Results are mod 2^4

Subtracting Unsigned Integers

Subtraction

```
              12
             0202
  10         1010B
-  7       - 0111B
  --         ----
   3         0011B
```

Start at right column
Proceed leftward
Borrow 2 when necessary

```
             2
   3         0011B
- 10       - 1010B
  --         ----
   9         1001B
```

Beware of overflow

Results are mod 2^4

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001
```

Shift Left

- Move all the bits `N` positions to the left, subbing in `N` `0`s on the right

```
1001 << 2 =
100100
```

Shift Left

- Useful as a restricted form of multiplication
- Question: how?

```
1001 << 2 =
100100
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340
```

Shift Left as Multiplication

- Equivalent decimal operation:

```
234 << 1 =
2340

234 << 2 =
23400
```

Multiplication

- Shifting left `N` positions multiplies by `(base)`^{N}
- Multiplying by 2 or 4 is often necessary (shift left 1 or 2 positions, respectively)
- Often a whooole lot faster than telling the processor to multiply

```
234 << 2 =
23400
```

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` `1`s on the left
  - Two different forms

Shift Right

- Move all the bits `N` positions to the right, subbing in **either** `N` `0`s or `N` (whatever the leftmost bit is)s on the left
  - Two different forms

```
1001 >> 2 =
either 0010 or 1110
```

Shift Right as Division

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

Shift Right as Division

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

234

Shift Right as Division

- Question: If shifting left multiplies, what does shift right do?
  - Answer: divides in a similar way, but truncates result

```
234 >> 1 =
23
```

Shifting Unsigned Integers

Bitwise right shift (>>): fill on left with zeros

```
10 >> 1 => 5
```

1010_B 0101_B

```
10 >> 2 => 2
```

1010_B 0010_B

What is the effect arithmetically? (No fair looking ahead)

Bitwise left shift (<<): fill on right with zeros

```
5 << 1 => 10
```

0101_B 1010_B

```
3 << 2 => 12
```

0011_B 1100_B

What is the effect arithmetically? (No fair looking ahead)

Results are mod 2^4

Other Operations on Unsigned Ints

Bitwise NOT (~)

- Flip each bit

```
~10 => 5
```

1010_B 0101_B

Bitwise AND (&)

- Logical AND corresponding bits

```
 10        1010B
& 7      & 0111B
 --        ----
  2        0010B
```

Useful for setting selected bits to 0

Other Operations on Unsigned Ints

Bitwise OR: (|)

- Logical OR corresponding bits

```
  10        1010B
|  1      | 0001B
  --        ----
  11        1011B
```

Useful for setting selected bits to 1

Bitwise exclusive OR (^)

- Logical exclusive OR corresponding bits

```
  10        1010B
^ 10      ^ 1010B
  --        ----
   0        0000B
```

x ^ x sets all bits to 0

The binary **XOR** operation will always produce a **1** output if either of its inputs is **1** and will produce a **0** output if both of its inputs are **0** or **1**.

Aside: Using Bitwise Ops for Arith

Can use <<, >>, and & to do some arithmetic efficiently

x * 2^y == x << y

- 3*4 = 3*2² = 3 << 2 => 12

  0011_B 1100_B

Fast way to **multiply** by a power of 2

x / 2^y == x >> y

- 13/4 = 13/2² = 13 >> 2 => 3

  1101_B 0011_B

Fast way to **divide** by a power of 2

x % 2^y == x & (2^y-1)

- 13%4 = 13%2² = 13&(2²-1)
  = 13&3 => 1

Fast way to **mod** by a power of 2

```
 13         1101B
& 3       & 0011B
--          ----
  1         0001B
```

Two Forms of Shift Right

- Subbing in `0`s makes sense
- What about subbing in the leftmost bit?
  - And why is this called "arithmetic" shift right?

```
1100 (arithmetic)>> 1 =
1110
```

Answer... Sort of

- Arithmetic form is intended for numbers in *two's complement* (next lecture), whereas the non-arithmetic form is intended for *unsigned* numbers

Agenda

Number Systems (Lecture 1)

Finite representation of unsigned integers (Lecture 2)

Finite representation of signed integers (Lecture 3)

Finite representation of rational numbers (Lecture 4)

Signed Magnitude

Integer	Rep
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit indicates sign

- 0 => positive
- 1 => negative

Remaining bits indicate magnitude

$1101_B = -101_B = -5$

$0101_B = 101_B = 5$

Signed Magnitude (cont.)

Integer	Rep
-7	1111
-6	1110
-5	1101
-4	1100
-3	1011
-2	1010
-1	1001
-0	1000
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Computing negative

neg(x) = flip high order bit of x

$\mathtt{neg}(0101_B) = 1101_B$

$\mathtt{neg}(1101_B) = 0101_B$

Pros and cons

+ easy for people to understand
+ symmetric
- two reps of zero
- one of the bit patterns is wasted.
- addition doesn't work the way we want it to.

Signed Magnitude (cont.)

Problem #1: "The Case of the Missing Bit Pattern":

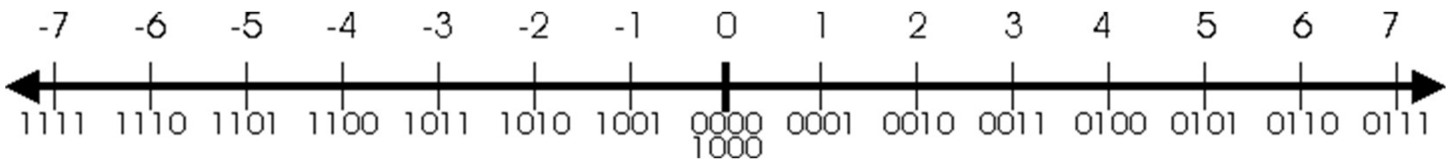
How many possible bit patterns can be created with 4 bits?

Easy, we know that's 16. In unsigned representation, we were able to represent 16 numbers: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15.

But with signed magnitude, we are only able to represent 15 numbers: -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, and 7.

There's still 16 bit patterns, but one of them is either not being used or is duplicating a number. That bit pattern is '1000B'.

When we interpret this pattern, we get '-0' which is both nonsensical (negative zero? come on!) and redundant (we already have '0000B' to represent 0).



Signed Magnitude (cont.)

Problem #2: "Requires Special Care and Feeding": Remember we wanted to have negative binary numbers so we could use our binary addition algorithm to simulate binary subtraction. How does signed magnitude fare with addition? To test it, let's try subtracting 2 from 5 by adding 5 and -2. A positive 5 would be represented with the bit pattern '0101B' and -2 with '1010B'. Let's add these two numbers and see what the result is:

```
 0101
+1010
----------
 1111
```

Now we interpret the result as a signed magnitude number. The sign is ‘1’ (negative) and the magnitude is '7'. So the answer is a negative 7. But, wait a minute, 5-2=3! This obviously didn't work.

Conclusion: signed magnitude doesn't work with regular binary addition algorithms.

One's Complement

Integer	Rep
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Definition

High-order bit has weight -7 ($- 2^n + 1$)

$$1010_B = (1*-7)+(0*4)+(1*2)+(0*1) = -5$$

$$0010_B = (0*-7)+(0*4)+(1*2)+(0*1) = 2$$

One's Complement (cont.)

Integer	Rep
-7	1000
-6	1001
-5	1010
-4	1011
-3	1100
-2	1101
-1	1110
-0	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110

Computing negative

neg(x) = ~x

$\mathtt{neg}(0101_B) = 1010_B$

$\mathtt{neg}(1010_B) = 0101_B$

Computing negative (alternative)

neg(x) = 1111_B - x

$\mathtt{neg}(0101_B) = 1111_B - 0101_B$
$= 1010_B$

$\mathtt{neg}(1010_B) = 1111_B - 1010_B$
$= 0101_B$

Pros and cons

\+ symmetric

\- two reps of zero

Two's Complement

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

Definition

High-order bit has weight -8 (-2^n)

$$1010_B = (1*-8)+(0*4)+(1*2)+(0*1)$$
$$= -6$$

$$0010_B = (0*-8)+(0*4)+(1*2)+(0*1)$$
$$= 2$$

Two's Complement (cont.)

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

Computing negative

neg(x) = ~x + 1

neg(x) = onescomp(x) + 1

neg $(0101_B) = 1010_B + 1 = 1011_B$

neg $(1011_B) = 0100_B + 1 = 0101_B$

Pros and cons

- not symmetric

+ one rep of zero

Two's Complement (cont.)

Almost all computers use two's complement to represent signed integers

Why?

- Arithmetic is easy
- Will become clear soon

Hereafter, assume two's complement representation of signed integers

Two's Complement

- Way to represent positive integers, negative integers, and zero
- If `1` is in the *most significant bit* (generally leftmost bit in this class), then it is negative

Decimal to Two's Complement

- Example: -5 decimal to binary (twos complement)

http://sandbox.mc.edu/~bennet/cs110/tc/dtotc.html

Decimal to Two's Complement

- Example: -5 decimal to binary (twos complement)
- First, convert the magnitude to an unsigned representation

Decimal to Two's Complement

- Example: -5 decimal to binary (two's complement)
- First, convert the magnitude to an unsigned representation

`5` **(decimal) = `0101` (binary)**

Decimal to Two's Complement

- Then, take the bits, and negate them

Decimal to Two's Complement

- Then, take the bits, and negate them

0101

Decimal to Two's Complement

- Then, take the bits, and negate them

```
~0101 =
 1010
```

Decimal to Two's Complement

- Finally, add one:

Decimal to Two's Complement

- Finally, add one:

```
1010
```

Decimal to Two's Complement

- Finally, add one:

```
1010 + 1 =
1011
```

Two's Complement to Decimal

- Same operation: negate the bits, and add one

Two's Complement to Decimal

- Same operation: negate the bits, and add one

```
1011
```

Two's Complement to Decimal

- Same operation: negate the bits, and add one

```
~1011 =
 0100
```

Two's Complement to Decimal

- Same operation: negate the bits, and add one

`0100`

Two's Complement to Decimal

- Same operation: negate the bits, and add one

```
0100 + 1 =
0101
```

Two's Complement to Decimal

- Same operation: negate the bits, and add one

```
0100 + 1 =
0101 =
-5
```

We started with `1011` - negative

Addition

http://sandbox.mc.edu/~bennet/cs110/textbook/module3_2.html

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
+123
----
    ?
```

```
  6
+3
--
  ?
```

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

		8	6
		+2	+3
		--	--
		?	9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

	Carry: 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

	1 9 +1 -- ?	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
 986
+123
----
   ?
```

Carry: 1	1 9 +1 -- 1	8 +2 -- 0	6 +3 -- 9

Building Up Addition

- Question: how might we add the following, in decimal?

```
  986
 +123
 ----
    ?
```

1 +0 -- 1	1 9 +1 -- 1	8 +2 -- 0	6 +3 -- 9

Core Concepts

- We have a "primitive" notion of adding single digits, along with an idea of *carrying* digits
- We can build on this notion to add numbers together that are more than one digit long

Now in Binary

- Arguably simpler - fewer one-bit possibilities

0 +0 -- ?	0 +1 -- ?	1 +0 -- ?	1 +1 -- ?

Now in Binary

- Arguably simpler - fewer one-bit possibilities

0 +0 -- 0	0 +1 -- 1	1 +0 -- 1	1 +1 -- 0 Carry: 1

Chaining the Carry

- Also need to account for any input carry

0 0 +0 -- 0	0 0 +1 -- 1	0 1 +0 -- 1	0 1 +1 -- 0 Carry: 1
1 0 +0 -- 1	1 0 +1 -- 0 Carry: 1	1 1 +0 -- 0 Carry: 1	1 1 +1 -- 1 Carry: 1

Adding Multiple Bits

- How might we add the numbers below?

```
  011
 +001
------
```

Adding Multiple Bits

- How might we add the numbers below?

```
     0
   011
  +001
 ------
```

Adding Multiple Bits

- How might we add the numbers below?

```
   10
  011
 +001
 ------
    0
```

Adding Multiple Bits

- How might we add the numbers below?

```
    110
    011
   +001
 ------
     00
```

Adding Multiple Bits

- How might we add the numbers below?

```
 0110
  011
 +001
------
  100
```

Adding Multiple Bits

- How might we add the numbers below?

```
 0110
   011
 +001
------
   100
```

Output Carry Bit

Result Bits

Another Example

```
  111
 +001
------
```

Another Example

```
     0
  111
 +001
------
```

Another Example

```
    10
   111
  +001
------
     0
```

Another Example

```
  110
  111
 +001
------
   00
```

Another Example

```
 1110
  111
 +001
------
  000
```

Output Carry Bit

Result Bits

Output Carry Bit Significance

- For unsigned numbers, it indicates if the result did not fit all the way into the number of bits allotted
- May be an error condition for software

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
   ?
```

Signed Addition

- Question: what is the result of the following operation?

```
 011
+011
----
0110
```

Overflow

- In this situation, *overflow* occurred: this means that both the operands had the same sign, and the result's sign differed

```
 011
+011
----
 110
```

- Possibly a software error

Overflow vs. Carry

- These are **different ideas**
  - Carry is relevant to **unsigned** values
  - Overflow is relevant to **signed** values

111 +001 ---- 000	011 +011 ---- 110	111 +100 ---- 011	001 +001 ---- 010
No Overflow; Carry	Overflow; No Carry	Overflow; Carry	No Overflow; No Carry

Adding Signed Integers

pos + pos

```
              11
   3         0011B
 + 3       + 0011B
  --         ----
   6         0110B
```

pos + pos (overflow)

```
             111
   7         0111B
 + 1       + 0001B
  --         ----
  -8         1000B
```

pos + neg

```
            1111
   3         0011B
 + -1      + 1111B
  --         ----
   2        10010B
```

neg + neg

```
            11
  -3         1101B
 + -2      + 1110B
  --         ----
  -5        11011B
```

neg + neg (overflow)

```
            1 1
  -6         1010B
 + -5      + 1011B
  --         ----
   5        10101B
```

Subtracting Signed Integers

Perform subtraction with borrows

or

Compute two's comp and add

	1
	22
3	0011_B
- 4	- 0100_B
-1	1111_B

3	0011_B
+ -4	+ 1100_B
-1	1111_B

-5	1011_B
- 2	- 0010_B
-7	1001_B

	111
-5	1011
+ -2	+ 1110
-7	11001

Shifting Signed Integers

Bitwise (**logical/arithmetic**) left shift (<<): fill on right with zeros

```
3 << 1 => 6
```

0011_B 0110_B

```
-3 << 1 => -6
```

1101_B 1010_B

Shift by n = multiplying by 2^n

Bitwise **arithmetic** right shift: fill on left **with sign bit**

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => -3
```

1010_B 1101_B

Shift by n = dividing by 2^n and Round-floor

Results are mod 2^4

Shifting Signed Integers (cont.)

Bitwise **logical** right shift: fill on left **with zeros**

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => 5
```

1010_B 0101_B

?

Right shift (>>) could be logical or arithmetic

- Compiler designer decides
- **Logical** shift is ideal for unsigned binary numbers
- **Arithmetic** shift is ideal for signed two’s complement binary numbers

Other Operations on Signed Ints

Bitwise NOT (~)

- Same as with unsigned ints

Bitwise AND (&)

- Same as with unsigned ints

Bitwise OR: (|)

- Same as with unsigned ints

Bitwise exclusive OR (^)

- Same as with unsigned ints

Agenda

Number Systems (Lecture 1)

Finite representation of unsigned integers (Lecture 2)

Finite representation of signed integers (Lecture 3)

Finite representation of rational numbers (Lecture 4)

Number Systems

- So far, we have studied the following integer number systems in computer
  - Unsigned numbers
  - Sign/magnitude numbers
  - Two's complement numbers

- What about rational numbers?

  - A **rational** number is one that can be expressed as the **ratio** of two integers
  - Infinite range and precision
  - For example, 2.5, -10.04, 0.75 etc

Rational Numbers

- Two common notations to represent rational numbers in computer
  - Fixed-point numbers
  - Floating-point numbers

Computer science

- Finite range and precision
- Approximate using **floating point** number
  - Binary point “floats” across bits

Fixed-Point Numbers

- Fixed point notation has an implied binary point between the integer and fraction bits
  - The binary point is not a part of the representation but is implied
  - Example:
    - Fixed-point representation of 6.75 using 4 integer bits and 4 fraction bits:

01101100

0110.1100

$2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

- The number of integer and fraction bits must be agreed upon by those generating and those reading the number
  - There is no way of knowing the existence of the binary point except through agreement of those people interpreting the number

Signed Fixed-Point Numbers

- As with whole numbers, negative fractional numbers can be represented in two ways
  - Sign/magnitude notation
  - Two's complement notation

- Example:
  - -2.375 using 8 bits (4 bits each to represent integer and fractional parts)
    - 2.375 = 0010 . 0110
    - Sign/magnitude notation: 1010 0110
    - Two's complement notation:

1. flip all the bits:	1101 1001
2. add 1:	+ 1
	1101 1010

- Addition and subtraction works easily in computer with 2's complement notation like integer addition and subtraction

Example

- Suppose that we have 8 bits to represent a number
  - 4 bits for integer and 4 bits for fraction

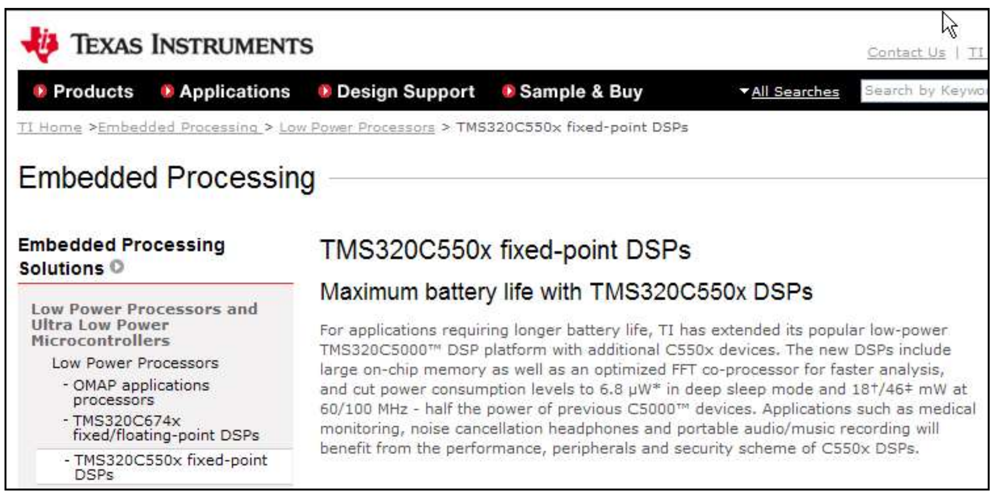
- Compute 0.75 + (-0.625)
  - 0.75 = 0000 1100
  - 0.625 = 0000 1010
  - -0.625 in 2's complement form: 1111 0110

0.75	0000 1100
+ - 0.625	1111 0110
0.125	0000 0010

Fixed-Point Number Systems

- Fixed-point number systems have a limitation of having a constant number of integer and fractional bits

- Some low-end digital signal processors support fixed-point numbers
  - Example: TMS320C550x TI (Texas Instruments) DSPs: www.ti.com



Floating-Point Numbers

- Floating-point number systems circumvent the limitation of having a constant number of integer and fractional bits
  - They allow the representation of very large and very small numbers

- The binary point floats to the right of the most significant 1
  - Similar to decimal scientific notation
  - For example, write 273_{10} in scientific notation:
    - Move the decimal point to the right of the most significant digit and increase the exponent:

$$273 = 2.73 \times 10^2$$

- In general, a number is written in scientific notation as:

$$\pm M \times B^E$$

Where,

- M = mantissa
- B = base
- E = exponent
- In the example, M = 2.73, B = 10, and E = 2 (that is, $+2.73 \times 10^2$)

Floating-Point Numbers

- Floating-point number representation using 32 bits
  - 1 sign bit
  - 8 exponent bits
  - 23 bits for the mantissa.

- The following slides show three versions of floating-point representation with 228_{10} using a 32-bit
  - The final version is called the IEEE 754 floating-point standard

Floating-Point Representation #1

- First, convert the decimal number to binary
  - $228_{10} = 11100100_2 = 1.11001 \times 2^7$
- Next, fill in each field in the 32-bit:
  - The sign bit (1 bit) is positive, so 0
  - The exponent (8 bits) is 7 (111)
  - The mantissa (23 bits) is 1.11001

1 bit	8 bits	23 bits
0	00000111	11 1001 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa / Fraction**

Floating-Point Representation #2

- You may have noticed that the first bit of the mantissa is always 1, since the binary point floats to the right of the most significant 1
  - Example: $228_{10} = 11100100_2 = \mathbf{1}.11001 \times 2^7$
- Thus, storing the most significant 1 (also called the implicit leading 1) is redundant information
- We can store just the fraction parts in the 23-bit field
  - Now, the leading 1 is implied

1 bit	8 bits	23 bits
0	0 0 0 0 0 1 1 1	110 0100 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa / Fraction**

Floating-Point Representation #3

- The exponent needs to represent both positive and negative
- The final change is to use a biased exponent
  - The IEEE 754 standard uses a bias of 127
  - Biased exponent = bias + exponent
    - For example, an exponent of 7 is stored as 127 + 7 = 134 = 10000110_2
- Thus , 228_{10} using the IEEE 754 32-bit floating-point standard is

$228_{10} = 11100100_2 =$ **1**$.11001 \times 2^7$

1 bit	8 bits	23 bits
0	10000110	110 0100 0000 0000 0000 0000
Sign	**Biased Exponent**	**Mantissa / Fraction**

Most general purpose processors (including Intel and AMD processors) provide hardware support for double-precision floating-point numbers and operations

IEEE Floating Point Representation

Common finite representation: **IEEE floating point**

- More precisely: ISO/IEEE 754 standard

Using 32 bits (type float in C):

- 1 bit: sign (0=>positive, 1=>negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form 1.*ddddddddddddddddddddddd*

Using 64 bits (type double in C):

- 1 bit: sign (0=>positive, 1=>negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form 1.*dddddddddddddddddddddddddddddddddddddddddddddddddddd*

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Mantissa / Fraction**

Example

- Represent -58_{10} using the IEEE 754 floating-point standard
  - First, convert the decimal number to binary
    - $58_{10} = 111010_2 = 1.1101 \times 2^5$
  - Next, fill in each field in the 32-bit number
    - The sign bit is negative (1)
    - The 8 exponent bits are $(127 + 5) = 132 = 10000100_{(2)}$
    - The remaining 23 bits are the fraction bits: $11010000...000_{(2)}$

1 bit	8 bits	23 bits
1	10000100	110 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

  - It is 0xC2680000 in the hexadecimal form

Double Precision Example

- Represent -58_{10} using the IEEE 754 double precision
  - First, convert the decimal number to binary
    - $58_{10} = 111010_2 = 1.1101 \times 2^5$
  - Next, fill in each field in the 64-bit number
    - The sign bit is negative (1)
    - The 11 exponent bits are $(1023 + 5) = 1028 = 10000000100_{(2)}$
    - The remaining 52 bits are the fraction bits: $11010000...000_{(2)}$
  - It is 0xC04D000000000000 in the hexadecimal form

Floating-Point Numbers: Special Cases

- The IEEE 754 standard includes special cases for numbers that are difficult to represent, such as 0 because it lacks an implicit leading 1

Number	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
- ∞	1	11111111	00000000000000000000000
NaN	X	11111111	non-zero

NaN is used for numbers that don't exist, such as $\sqrt{-1}$ or log(-5)

Floating Point Example

`11000001110110110000000000000000`

32-bit representation

Sign (1 bit):

- 1 => negative

Exponent (8 bits):

- $10000011_B = 131$
- $131 - 127 = 4$

Fraction (23 bits):

- $1.10110110000000000000000_B$
- $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5})+(1*2^{-6})+(1*2^{-7}) = 1.7109375$

Number:

- $-1.7109375 * 2^4 = -27.375$

Floating Point Example 263.3

2	263	
2	131	1
2	65	1
2	32	1
2	16	0
2	8	0
2	4	0
2	2	0
2	1	0
	0	1

263: 100000111

IEEE754 floating-point standard can't represent some numbers exactly

0.3 * 2	0.6	0
0.6 * 2	1.2	1
0.2 * 2	0.4	0
0.4 * 2	0.8	0
0.8 * 2	1.6	1
0.6 * 2	1.2	1
		0
		0
		1
		1
		0

Stop when it gets 1.0

0.3 : 01001100110011....

Floating Point Example

1) 263.3

100000111.0100110011…

2) Scientific notation:

$$\underbrace{1.00000111010011 0011\ldots}_{\text{Mantissa}} * 2^{8}$$

Sign (1 bit):

- positive => 0

Exponent (8 bits):

- 127 + 8 = 135
- 135 = 10000111_B

Fraction (23 bits):

- 00000111010011001100110

`0100 0011 1000 0011 1010 0110 0110 0110`

32-bit representation

Binary Coded Decimal (BCD)

- Since floating-point number systems can't represent some numbers exactly such as 0.3, some application (calculators) use BCD (Binary coded decimal)
  - BCD numbers encode each decimal digit using 4 bits with a range of 0 to 9

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

BCD fixed-point notation examples

1.7 = 0001 . 0111

4.9 = 0100 . 1001

6.75 = 0110.01110101

- BCD is very common in electronic systems where a numeric value is to be displayed, especially, in systems consisting solely of digital logic (not containing a microprocessor) - Wiki

Examples

1- Convert Decimal to Floating Point (IEEE 754)

https://www.youtube.com/watch?v=8afbTaA-gOQ

2- Convert Floating Point (IEEE 754) to Decimal

https://www.youtube.com/watch?v=LXF-wcoeT0o

Converting Between Decimal and Binary Floating-Point Numbers

https://mebrahimii.github.io/comp122-summer2021/lecture/week_2/floating_point_interconversions.html

Summary

The binary, hexadecimal, and octal number systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers

Essential for proper understanding of

- C primitive data types
- Assembly language
- Machine language

Backup Slides

Floating-Point Numbers: Rounding

- Arithmetic results that fall outside of the available precision must round to a neighboring number
- Rounding modes
  - Round down
  - Round up
  - Round toward zero
  - Round to nearest
- Example
  - Round 1.100101 (1.578125) so that it uses only 3 fraction bits
    - Round down: 1.100
    - Round up: 1.101
    - Round toward zero: 1.100
    - Round to nearest: 1.101
      - 1.625 is closer to 1.578125 than 1.5 is

Floating-Point Addition with the Same Sign

- Addition with floating-point numbers is not as simple as addition with 2's complement numbers

- The steps for adding floating-point numbers with the same sign are as follows
  1. Extract exponent and fraction bits
  2. Prepend leading 1 to form mantissa
  3. Compare exponents
  4. Shift smaller mantissa if necessary
  5. Add mantissas
  6. Normalize mantissa and adjust exponent if necessary
  7. Round result
  8. Assemble exponent and fraction back into floating-point format

Floating-Point Addition Example

Add the following floating-point numbers:

$1.5 + 3.25$

$1.5_{(10)} = 1.1_{(2)} \times 2^0$

$3.25_{(10)} = 11.01_{(2)} = 1.101_{(2)} \times 2^1$

$1.1_{(10)}$ = 0x3FC00000 in IEEE 754 single precision

$3.25_{(10)}$ = 0x40500000 in IEEE 754 single precision

Floating-Point Addition Example

1. Extract exponent and fraction bits

1 bit	8 bits	23 bits
0	01111111	100 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

1 bit	8 bits	23 bits
0	10000000	101 0000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

For first number (N1): S = 0, E = 127, F = .1
For second number (N2): S = 0, E = 128, F = .101

2. Prepend leading 1 to form mantissa

N1: 1.1
N2: 1.101

Floating-Point Addition Example

3. Compare exponents
   127 – 128 = -1, so shift N1 right by 1 bit

4. Shift smaller mantissa if necessary
   shift N1's mantissa: 1.1 >> 1 = 0.11 ($\times 2^1$)

5. Add mantissas

$$
\begin{array}{r}
0.11 \quad \times 2^1 \\
+\ 1.101 \times 2^1 \\
\hline
10.011 \times 2^1
\end{array}
$$

Floating-Point Addition Example

6. Normalize mantissa and adjust exponent if necessary

$$10.011 \times 2^1 = 1.0011 \times 2^2$$

7. Round result

No need (fits in 23 bits)

8. Assemble exponent and fraction back into floating-point format

$$S = 0,\ E = 2 + 127 = 129 = 10000001_2,\ F = 001100..$$

1 bit	8 bits	23 bits
0	10000001	001 1000 0000 0000 0000 0000
Sign	**Exponent**	**Fraction**

$4.75_{(10)}$ = 0x40980000 in the hexadecimal form