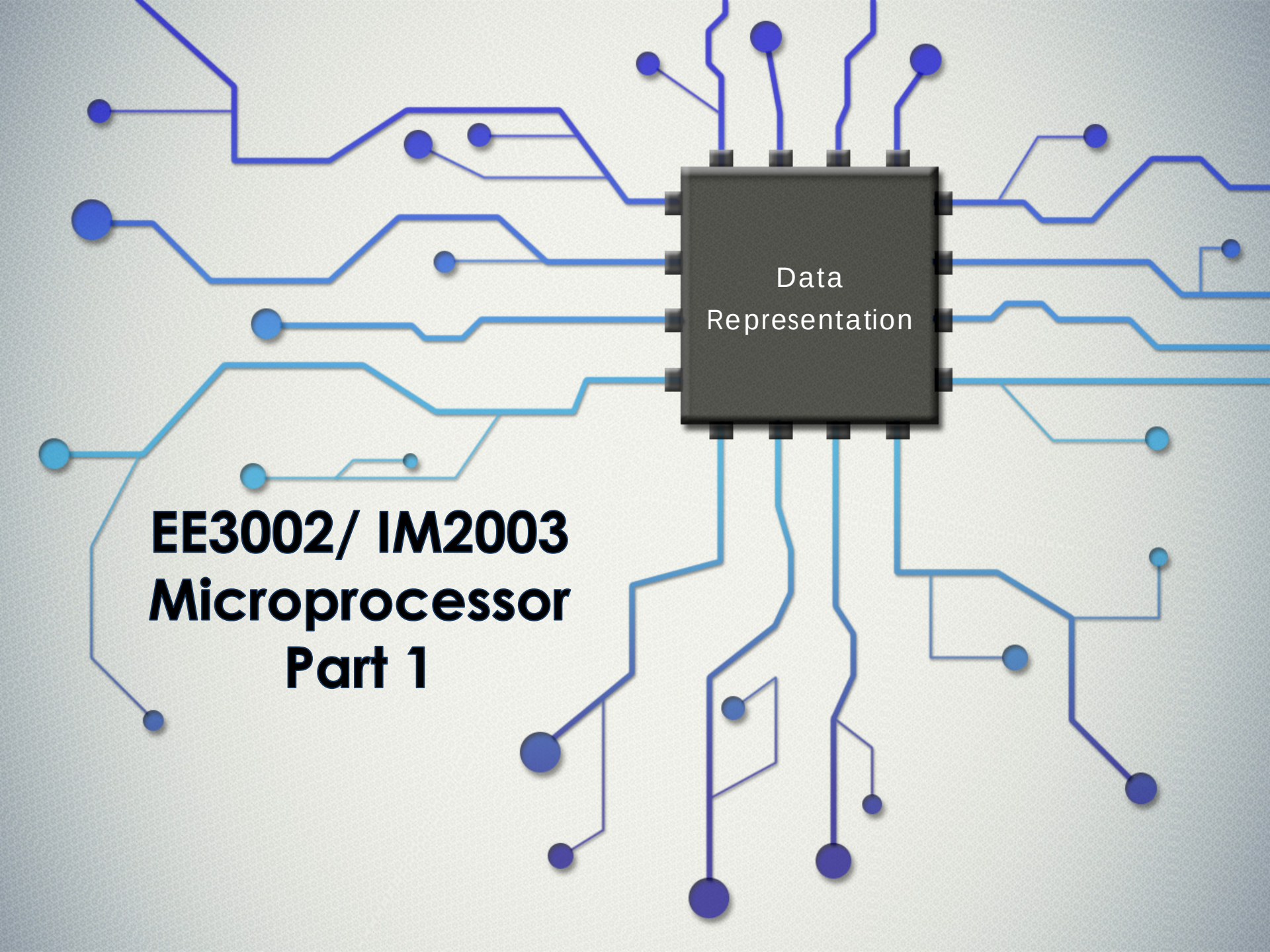


EE3002/ IM2003 Microprocessor Part 1

Kinds Of Data

- Numbers
  - Integers
- Unsigned
- Signed
  - Reals
- Fixed-Point
- Floating-Point
  - Binary-Coded Decimal

- Text
  - ASCII Characters
  - Strings
- Other
  - Graphics
  - Images
  - Video
  - Audio

Numbers Are Different!

- Computers use binary (not decimal) numbers (0's and 1's).
  - Requires more digits to represent the same magnitude.
- Computers store and process numbers using a fixed number of digits ("fixed-precision").
- Computers represent signed numbers using 2's complement instead of the more natural (for humans) "sign-plus-magnitude" representation.

Positional Number Systems

- Numeric values are represented by a *sequence* of digit symbols.

- Symbols represent numeric *values*.
  - Symbols are not limited to '0'-'9'!

- Each symbol's contribution to the total value of the number is *weighted* according to its position in the sequence.

Polynomial Evaluation

Whole Numbers (Radix = 10):

$$1234_{10} = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

With Fractional Part (Radix = 10):

$$36.72_{10} = 3 \times 10^1 + 6 \times 10^0 + 7 \times 10^{-1} + 2 \times 10^{-2}$$

General Case (Radix = R):

$$(S_1 S_0 . S_{-1} S_{-2})_R =$$

$$S_1 \times R^1 + S_0 \times R^0 + S_{-1} \times R^{-1} + S_{-2} \times R^{-2}$$

Converting Radix R *to Decimal*

$$36.72_8 = 3 \times 8^1 + 6 \times 8^0 + 7 \times 8^{-1} + 2 \times 8^{-2}$$

$$= 24 + 6 + 0.875 + 0.03125$$

$$= 30.90625_{10}$$

Important: Polynomial evaluation doesn't work if you try to convert in the *other* direction – I.e., from decimal to something else! Why?

Binary to Decimal Conversion

Converting *to decimal*, so we can use polynomial evaluation:

10110101_2

$= 1\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 128 + 32 + 16 + 4 + 1$

$= 181_{10}$

Variation on Polynomial Evaluation for converting fractional values

Example: Convert 0.437_8 to decimal:

$= 4\times8^{-1} + 3\times8^{-2} + 7\times8^{-3}$

Multiple divisions

$= 4\times0.125 + 3\times0.015625 + 7\times0.001953125$

Adding long decimal fractions

Alternative approach:

$= (4\times8^2 + 3\times8^1 + 7\times8^0) / 8^3$

$= (4\times64 + 3\times8 + 7\times1) / 512$

$= 287 / 512 = 0.56054687510$

Problems: $N_R \rightarrow N_{10}$

110101_2

$.110101_2$

432_5

$.432_5$

137_8

$.137_8$

$3F.4A_{16}$

$F0.0D_{16}$

10_3

$.10_3$

Decimal to Binary Conversion (Fractional Part: Repeated Multiplication)

- Multiply by target radix (2 in this case)
- Whole part of product becomes digit in the new representation (0 <= digit < R)
- Digits produced in left to right order.
- Fractional part of product is used as next multiplicand.
- Stop when the fractional part becomes zero (sometimes it won't).

Decimal to Binary Conversion (Fractional Part: Repeated Multiplication)

$.1 \times 2 \rightarrow 0.2$ (fractional part = .2, whole part = 0)

$.2 \times 2 \rightarrow 0.4$ (fractional part = .4, whole part = 0)

$.4 \times 2 \rightarrow 0.8$ (fractional part = .8, whole part = 0)

$.8 \times 2 \rightarrow 1.6$ (fractional part = .6, whole part = 1)

$.6 \times 2 \rightarrow 1.2$ (fractional part = .2, whole part = 1)

Result = $.0001100110011001_2$.....

(How much should we keep?)

$.1_{10} = .0001100110011001100110011..._2$
How much should we keep?

Mathematician's Answer:

Use the proper notation: **.00011**

Scientist's Answer:

Preserve significant digits and round:

.1 → 1 part out of 10

3 binary digits = 1 out of 8 → need 4 → .0001

Round: 5th digit = 1, thus **.0010**

Engineer's Answer:

Depends on #bits in the variable (**8, 16, 32, 64**)

Moral

- Some fractional numbers have an exact representation in one number system, but not in another! E.g., 1/3rd has no exact representation in decimal, but does in base 3!
- What about 1/10th when represented in binary?
- Can these *representation errors* accumulate?
- What does this imply about *equality comparisons* of real numb?

Problems: $N_{10} \rightarrow N_R$

$27_{10} \rightarrow N_2$

$.27_{10} \rightarrow N_2$

$27_{10} \rightarrow N_5$

$.27_{10} \rightarrow N_5$

$1/3_{10} \rightarrow N_3$

$27_{10} \rightarrow N_8$

$.27_{10} \rightarrow N_8$

$27_{10} \rightarrow N_{16}$

$.27_{10} \rightarrow N_{16}$

Counting

- Principle is the same regardless of radix.
  - Add 1 to the least significant digit.
  - If the result is less than R, write it down and copy all the remaining digits on the left.
  - Otherwise, write down zero and add 1 to the next digit position, etc.

Counting in Binary

Dec	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Note the pattern!

- LSB (bit 0) toggles on *every* count.
- Bit 1 toggles on every *other* count.
- Bit 2 toggles on every *fourth* count.
- Etc….

Memorize This!

Hex	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Hex	Binary
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Binary/Hex Conversions

- Hex digits are in one-to-one correspondence with groups of four binary digits:
- 0011 1010 0101 0110 . 1110 0010 1111 1000

  3 A 5 6 . E 2 F 8
- Conversion is a simple table lookup!
- Zero-fill on left and right ends to complete the groups!
- Works because $16 = 2^4$ (power relationship)

Problems: $N_{R1} \rightarrow N_{R2}$, where $R1=R2^k$

$1101011_2 \rightarrow N_{16}$

$11.01011_2 \rightarrow N_{16}$

$1101011_2 \rightarrow N_8$

$11010.11_2 \rightarrow N_8$

$10220_3 \rightarrow N_9$

$10.220_3 \rightarrow N_9$

$FACE_{16} \rightarrow N_2$

$BEEF_{16} \rightarrow N_4$

$FEED_{16} \rightarrow N_8$

$1846_9 \rightarrow N_3$

Representation Rollover

- Consequence of *fixed precision.*
- Computers use fixed precision!
- Digits are lost on the left-hand end.
- Remaining digits are still correct.
- Rollover while counting . . .

Up: “999999” ➔ “000000” (R^n-1 ➔ 0)

Down: “000000” ➔ “999999” (0 ➔ R^n-1)

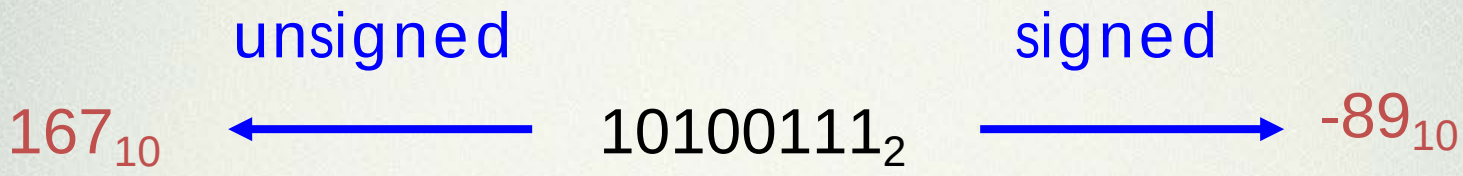
Rollover in Unsigned Binary

- Consider an 8-bit byte used to represent an unsigned integer:
  - Range: 00000000 ➔ 11111111 (0 ➔ 255_{10})
  - Incrementing a value of 255 should yield 256, but this exceeds the range.
  - Decrementing a value of 0 should yield –1, but this exceeds the range.
  - Exceeding the range is known as *overflow*.

Surprise! Rollover is not synonymous with overflow!

- Rollover describes a pattern sequence behavior.
- Overflow describes an arithmetic behavior.
- Whether or not rollover causes overflow depends on how the patterns are interpreted as numeric values!
  - E.g., In signed two's complement representation, 11111111 ➔ 00000000 corresponds to counting from minus one to zero.

Two Interpretations



- Signed vs. unsigned is a matter of interpretation; thus a single bit pattern can represent two different values.
- Allowing both interpretations is useful:

Some data (e.g., count, age) can never be negative, and having a greater range is useful.

Why Not Sign+Magnitude?

+3	0011
+2	0010
+1	0001
+0	0000
-0	1000
-1	1001
-2	1010
-3	1011

- Complicates addition :
  - To add, first check the signs. If they agree, then add the magnitudes and use the same sign; else subtract the *smaller* from the *larger* and use the sign of the larger.
  - How do you determine which is smaller/larger?
- Complicates comparators:
  - Two zeroes!

Why 2's Complement?

+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100

1. Just as easy to determine sign as in sign+magnitude.
2. Almost as easy to change the sign of a number.
3. Addition can proceed w/out worrying about which operand is larger.
4. A single zero!
5. One hardware adder works for both signed and unsigned operands.

Changing the Sign

Sign+Magnitude:

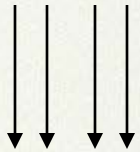
+5 = 0101

Change 1 bit

-5 = 1101

2's Complement:

+5 = 0101



Invert

1010

+1 *Increment*

-5 = 1011

Easier Hand Method

Step 2: Copy the inverse of the remaining bits.

+4 = 0100

-4 = 1100

Step 1: Copy the bits from right to left, through and including the first 1.

Representation Width

Be Careful! You must be sure to pad the original value out to the full representation width before applying the algorithm!

Apply algorithm

Expand to 8-bits

Wrong: +25 = 11001 ➔ 00111 ➔ 00000111 = +7

Right: +25 = 11001 ➔ 00011001 ➔ 11100111 = -25

If positive: Add leading 0's
If negative: Add leading 1's

Apply algorithm

Converting 2's complement numbers to decimal – Approach #1

If MSB is 0, the number is positive.

→ convert as if it were unsigned.

If MSB is 1, the number is negative.

1. Find representation of –N, where N is the original number.
2. Convert N to decimal.
3. Put a minus sign in front.

Converting 2's complement numbers to decimal – Approach #1

Example: $10110010_2 = ?_{10}$

1. $10110010_2 \rightarrow -01001110_2$
2. $01001110_2 = 64 + 8 + 4 + 2 = 78_{10}$
3. Answer: -78_{10}

Converting 2's complement numbers to decimal – Approach #2

Use polynomial evaluation, but make the contribution of the MSB be negative:

Example: $10110010_2 = ?_{10}$

$= -128 + 32 + 16 + 2 = -78_{10}$

2's Complement Anomaly!

-128 = 1000 0000 (8 bits)

+128?

Step 1: Invert all bits ➔ 0111 1111

Step 2: Increment ➔ 1000 0000

Same result with either method! Why?

Range of Unsigned Integers

Each of 'n' bits can have one of two values.

Total # of patterns of n bits = $\underbrace{2 \times 2 \times 2 \times \ldots 2}_{\text{'n' 2's}}$

$= 2^n$

If n-bits are used to represent an unsigned integer value:

Range: 0 to 2^n-1 (2^n different values)

Unsigned Range

- Byte (8 bits) 0 to 255
- Halfword (16 bits) 0 to 65535
- Word (32 bits) 0 to 4,294,967,295
- Double Word 0 to $2^{64}-1$

Range of Signed Integers

- Half of the 2^n patterns will be used for positive values, and half for negative.

- Half is 2^{n-1}.

- Positive Range: 0 to $2^{n-1}-1$ (2^{n-1} patterns)
- Negative Range: -2^{n-1} to -1 (2^{n-1} patterns)

- 8-Bits (n = 8): -2^7 (-128) to $+2^7-1$ (+127)

2's Complement Integer Range

- Byte (8 bits) – 128 to 127
- Halfword (16 bits) – 32,768 to 32,767
- Word (32 bits) – 2,147,483,648 to 2,147,483,647
- Double Word – 2^{63} to $2^{63} - 1$

Addition & Carries

Cin	X	Y	n	Cout	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Column “n” is simply the sum of Cin, X and Y.

Columns Cout & S are simply the binary representation of n.

Subtraction & Borrows

Bin	X	Y	n	Bout	D
0	0	0	0	0	0
0	0	-1	-1	1	1
0	+1	0	+1	0	1
0	+1	-1	0	0	0
-1	0	0	-1	1	1
-1	0	-1	-2	1	0
-1	+1	0	0	0	0
-1	+1	-1	-1	1	1

Column “n” is simply the sum of Bin, X and Y.

Columns Bout & D are simply the 2’s compl. representation of n.

Unsigned Overflow

```
  1 1 0 0   (12)
+ 0 1 1 1   ( 7)
-----------
  1 0 0 1 1
```

Lost ↑

(Result limited by word size)

(3) **wrong**

Value of lost bit is 2^n (16).

16 + 3 = 19

(The right answer!)

Signed Overflow

- Overflow is impossible ☺ when adding (subtracting) numbers that have different (same) signs.

- Overflow occurs when the magnitude of the result extends into the sign bit position:

01111111 ➔ (0)10000000

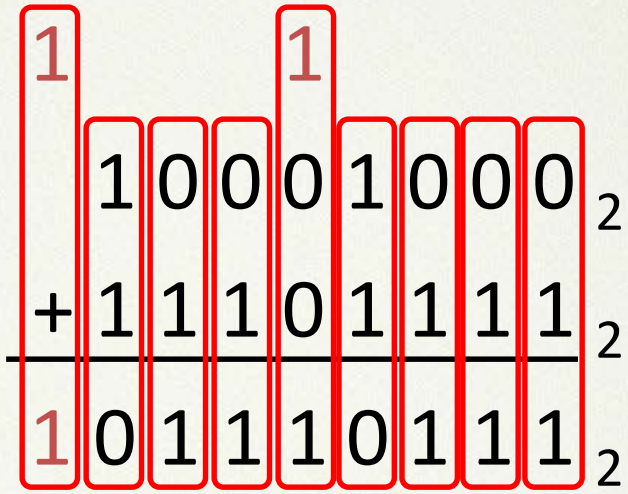
This is not rollover!

Signed Overflow

-120_{10}

-17_{10}

sum: -137_{10}



$0\ 1\ 1\ 1\ 0\ 1\ 1\ 1_{\ 2}$ (keep 8 bits)

$(+119_{10})$ **wrong**

Note: $119 - 2^8 = 119 - 256 = -137$

Detecting Overflow

Unsigned:

Carry-out of MSB when incrementing or adding.

Borrow-out of MSB when decrementing or subtracting.

Signed (2's complement):

Impossible when adding numbers of different signs.

Impossible when subtracting numbers of same sign.

Human Method: Sign of result different from operands.

Computer Method: Carries/Borrows in/out of MSB differ.

Problems: Overflow

Unsigned (4 bits)

```
01010    11100
 0101(5)    1011(11)
+0101(5)   +0110(6)
 1010(10)   0001(1)

11100    01000
 0100(4)    1011(11)
-0110(6)   -0101(5)
 1110(14)   0110(6)
```

2's comp. (4 bits)

```
01010    11100
 0101(5)  1011(-5)
+0101(5)   +0110(6)
 1010(-6)   0001(1)

11100    01000
 0100(4)  1011(-5)
-0110(6)   -0101(5)
 1110(-2)   0110(6)
```

Comparing Integers
Which is Greater: 1001 or 0011?

Unsigned:

Borrows	**0**	1	1	0	0
X		1	0	0	1
Y	-	0	0	1	1
X-Y		0	1	1	0

Borrow Out = 0

1001 ≥ 0011

Borrow Out	Relationship
0	X ≥ Y
1	X < Y

Signed (2's Complement):

Borrows	**0**	**1**	1	0	0
X		1	0	0	1
Y	-	0	0	1	1
X-Y		**0**	1	1	0

Overflow!

1001 < 0011

Positive

Sign of X-Y	Overflow	True Sign	Relationship
Positive	No	Positive	X ≥ Y
Negative	Yes	Positive	
Positive	Yes	Negative	X < Y
Negative	No	Negative	

Floating Point (Real Nos)

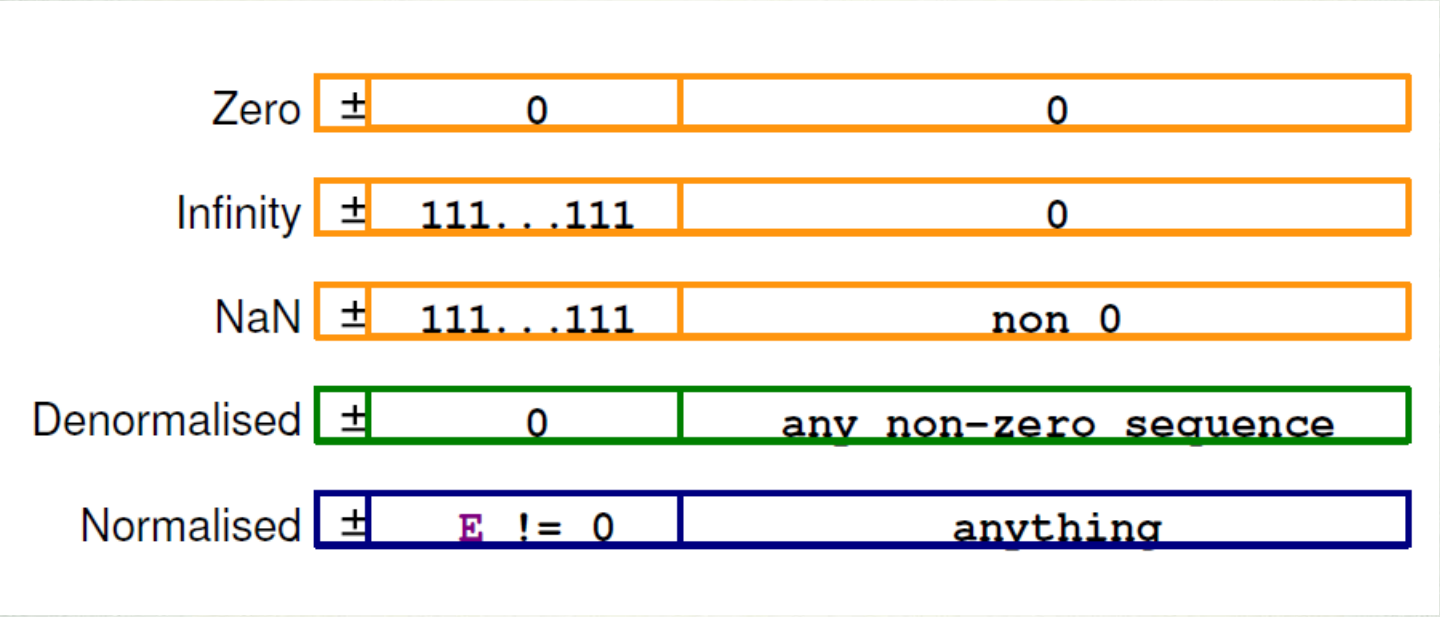
- Floating point numbers are also know as real numbers (i.e. 3.141)
- Floating point extends the range of numbers that can be stored by a computer, and the accuracy is independent of the number magnitude.
- Generally floating point numbers store a sign (***S***), exponent (***E***) *and mantissa* (***F***) and are of predetermined number base (***B***). ***B*** is usually set to base 2.
- Floating point value = $(-1)^S \times F \times B^E$
- 32-bit Single precision format

content	Sign	Exponent	Mantissa
Bit	31	23 to 30	0 to 22

IEEE standard 754

- Used in almost all modern FPUs.
- IEEE754 has 1 sign (**S), 8 exponent (E) and 23 mantissa bits (F)** for single precision and for 1 (S), 11 (E) and 52 bits (F) for double precision.
- The exponent is stored in excess 127 for single precision (and excess 1023 for double precision). Will focus on single precision for now.
- The mantissa is slightly unusual; for "normal" numbers, the 23 bits mantissa is a fraction and it is assumed that 1 must be added to the mantissa, but the 1 is not stored. In other words, the mantissa is a fractional value ranging from 1 (mantissa bits 0) to slightly below 2 (all mantissa bits 1).
- Not "normal" numbers will be discussed later.
- As the exponent is a power of 2, the mantissa never needs to be greater than 2 (you would just add 1 to the exponent instead).

5 different number types



Exponent

- IEEE 754's exponent is the actual exponent + 127

Actual Exponent	IEEE 754 Exponent
NaN or infinity!	255
…	…
3	130
2	129
1	128
0	127
-1	126
…	…
Subnormal number !	0

“Normal”Single-precision Floating-point Representation

	S	Exp+127	Mantissa
2.000	0	10000000	(1).00000000000000000000000
1.000	0	01111111	(1).00000000000000000000000
0.750	0	01111110	(1).10000000000000000000000
0.500	0	01111110	(1).00000000000000000000000
0.000	0	00000000	(0).00000000000000000000000
-0.500	1	01111110	(1).00000000000000000000000
-0.750	1	01111110	(1).10000000000000000000000
-1.000	1	01111111	(1).00000000000000000000000
-2.000	1	10000000	(1).00000000000000000000000

“Normal” Floating point examples

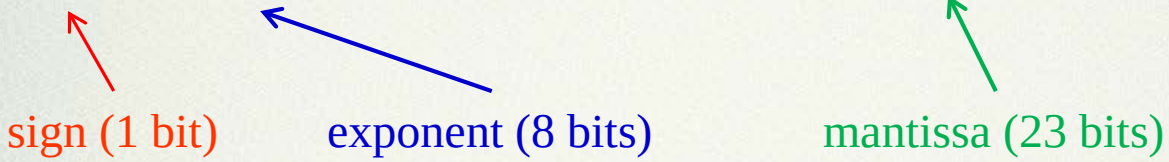
No Binary Representation

1 0011 1111 1000 0000 0000 0000 0000 0000

- This is positive, has exponent of 127−127= 0 (remember it's in excess 127 format), and mantissa of 1 so $1\times2^0 = 1.0$
- The most significant mantissa bit has the value of 0.5 or ½ , the next bit has the value of 0.25 or ¼, and so on.

"Normal" Floating point examples

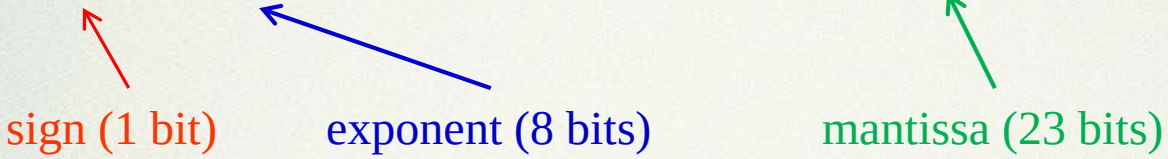
No	Binary Representation
1.5	0011 1111 1100 0000 0000 0000 0000 0000



- This is positive, has exponent of 127−127= 0 (remember it's in excess 127 format), and mantissa of 1.5 so $1.5 \times 2^0 = 1.5$

"Normal" Floating point examples

No	Binary Representation
−40	1100 0010 0010 0000 0000 0000 0000 0000



- This is negative, has exponent of 132−127= 5 (remember it's in excess 127 format), and mantissa of 1.25
- so $(-1) \times 1.25 \times 2^5 = -40$

How to convert a float number into IEEE 754 format?

- Convert 0.17431640625 into IEEE 754
- Step 1 Determine the sign bit

  0.17431640625 is positive, hence sign bit, S, is 0
- Step 2 Determine the mantissa

– The mantissa must be between 1 and 2, hence we multiply by 2 until we get it into the desired range.

Mantissa

0.17431640625 × 2 = 0.3486328125

0.3486328125 × 2 = 0.697265625

0.697265625 × 2 = 1.39453125

Since the 1 is implicit in IEEE 754, 0.39453125 is the value to be stored

Convert mantissa to binary fraction by successive multiplications

0.39453125 X 2 = 0.7890625 ; 0

0.7890625 X 2 = 1.578125 ; 1

0.578125 X 2 = 1.15625 ; 1

0.15625 X 2 = 0.3125 ; 0

0.3125 X 2 = 0.625 ; 0

0.625 X 2 = 1.25 ; 1

0.25 X 2 = 0.5 ; 0

0.5 X 2 = 1 ; 1

Mantissa = 0 1 1 0 0 1 0 1 0.. 0

Exponent

Step 3 Determine the exponent

0.17431640625×2×2×2 = 1.39453125

Or 0.17431640625= 1.39453125×2^{-3}

Exponent = –3

Since exponent in IEEE 754 is excess 127 form or 127 – 3 = 124

124 = 0111 1100B

Final Step

Step 4 Put everything together

S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
0	0	1	1	1	1	1	0	0	0	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3				E				3				2				8				0				0				0			

Counter check your answer using the following website
http://www.h-schmidt.net/FloatConverter/

Or download the IEEE 754 app from Google play store for Android machines

Alternative Method (Calculator required)

- Convert – 3.14159 into IEEE 754 format
- Step 1 Determine the sign bit
  - 3.14159 is negative, hence sign bit, S, is 1
- Step 2 Determine the exponent

– Use floor($\log_2$ 3.14159) = floor (1.65) = 1

– Exponent = 1

– Exponent in IEEE 754 = 127 + 1 = 128

– 128 = 10000000B

Mantissa

- Step 3: Compute Mantissa

– $3.14159 \times 2^{-1} = 1.570795$ (between 1 and 2)

– Convert 0.570795 (subtract 1) into 23 bit binary fraction

– $\text{round}(0.570795 \times 2^{23}) = 4788176$

– Convert 4788176 into hexadecimal first

– 4788176 = 0x490FD0 =

– 100 1001 0000 1111 1101 0000B (23 bits)

Final Step

Step 4 Put everything together

S	E	E	E	E	E	E	E	E	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
1	1	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	1	1	1	1	1	1	0	1	0	0	0	0
C				0				4				9				0				F				D				0			

Counter check your answer using the following website http://www.h-schmidt.net/FloatConverter/

Or download the IEEE 754 app from Google play store for Android machines

Smallest magnitude“Normal” number

No Binary Representation

? 0000 0000 1000 0000 0000 0000 0000 0000

- Smallest exponent is 1 for “normal” floating point number, if exponent is 0, it is a “denormalised” or “subnormal”number.
- This is negative, has exponent of 1−127= −126 (remember it's in excess 127 format), and mantissa of 1.00
- so $(1) \times 1.00 \times 2^{-126} = 2^{-126} \approx 1.1755 \times 10^{-38}$

“Denormalised” or “Subnormal”

- Denormalised floating point numbers are also known as Subnormal floating point numbers.
- They are smaller than “normal” numbers.
- The exponent field is all zeros.
- The mantissa field do not have a 1 added, it is just a straightforward binary fraction.
- The binary fraction has to be multiplied by the smallest magnitude “normal” number.

"Denormalised" Floating point

No Binary Representation

? 0000 0000 0101 0000 0000 0000 0000 0000

- Since exponent is 0, it is a "denormalised" number.
- Mantissa is 0.5+ 0.125 = 0.625 (Note: no need to add 1)
- so $(1) \times 0.625 \times 2^{-126} \approx 7.347 \times 10^{-39}$

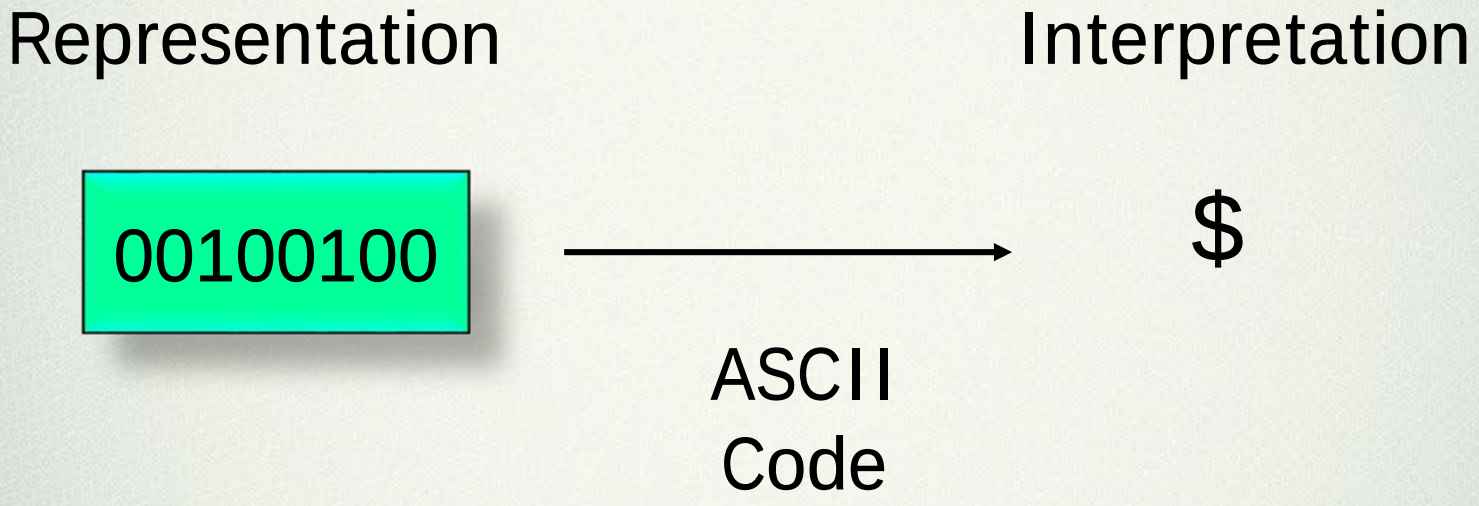
Range of IEEE 754 “normal” numbers

- Single precision range is :

  $\pm(1.2\times10^{-38}$ to $3.4\times10^{38})$
- Double precision range is :

  $\pm(2.2\times10^{-308}$ to $1.8\times10^{308})$
- What are the range of the denormalised numbers?

Representation of Characters



Character Constants in C

- To distinguish a character that is used as data from an identifier that consists of only one character long:

  x is an identifier.

  'x' is a character constant.

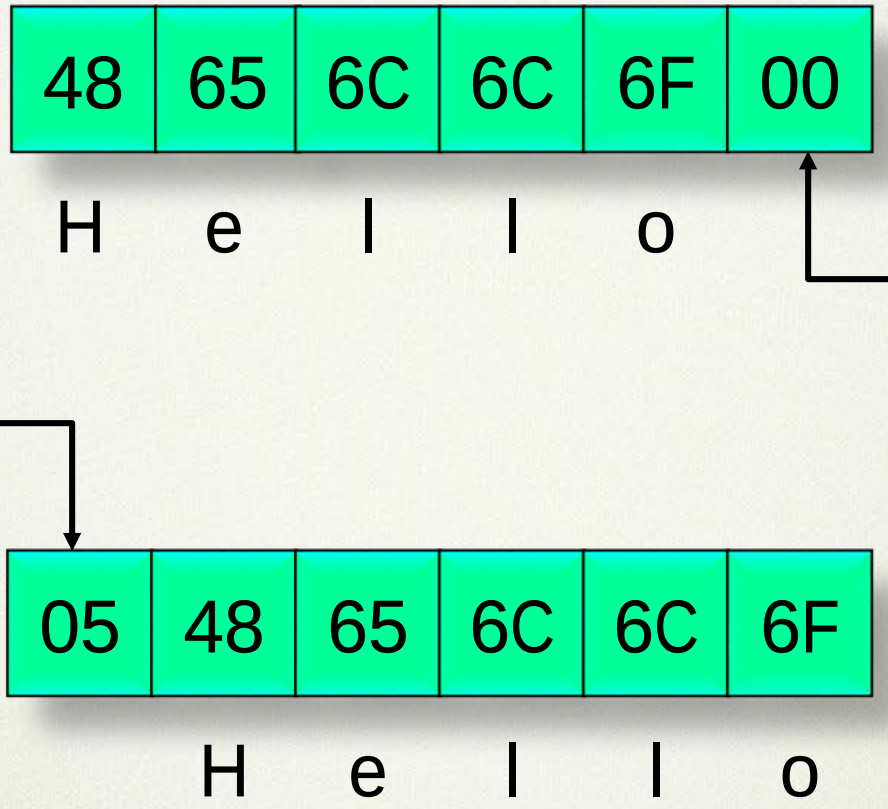
- The value of 'x' is the ASCII code of the character x.

Character Escapes

A way to represent characters that do not have a corresponding graphic symbol.

Escape Character		Character Constant
\b	Backspace	'\b'
\t	Horizontal Tab	'\t'
\n	Linefeed	'\n'
\r	Carriage return	'\r'

Representation of Strings



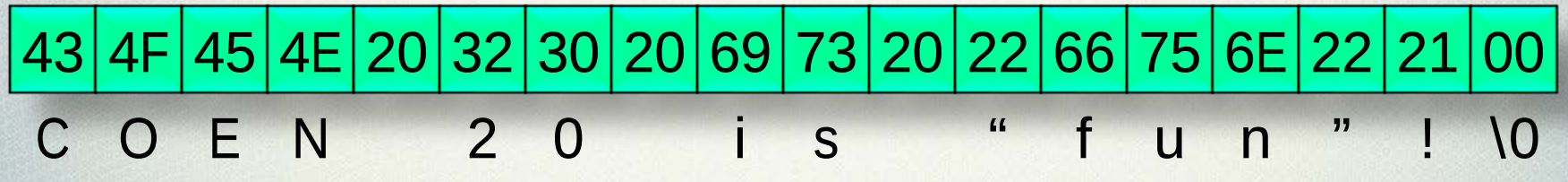
String Constants in C

Character string

COEN 20 is “fun”!

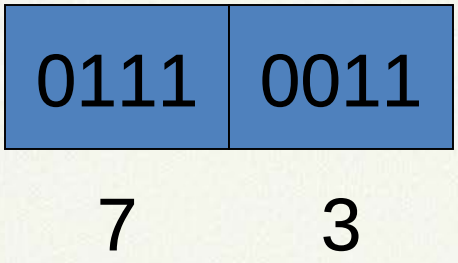
C string constant

“COEN 20 is \“fun\”!”

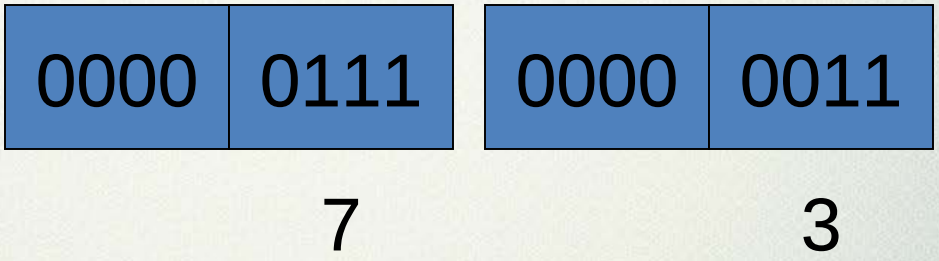


Binary Coded Decimal (BCD)

Packed (2 digits per byte):



Unpacked (1 digit per byte):



Number representation in Assembly Program

- Binary , Decimal and Hexadecimal numbers are commonly used in assembly language program.
- How do one differentiate between 11 binary (3), 11 Decimal or 11 Hexdecimal (17)?
- Prefix for binary numbers, e.g. 2_11
- Prefix x for hexadecimal e.g. 0x11
- Default is decimal e.g. 11

Summary

- Data are stored in computer as bits.
- Number base conversion
- 2's complement representation for negative numbers.
- IEEE 754 floating point representation
- Character and string representations