Number Systems

Class 27

Positional Notation

- for counting quantities from 0 through 9, we use digits because the typical human has 10 fingers and wears shoes

- for values larger than 9, we use a positional notation
- the number 7305 really means:

$$7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$$

- we call this the decimal number system because
  - it uses ten digits (0 – 9)
  - the coefficients are multiplied by powers of 10
- when necessary to disambiguate the radix (base) of 10, we write

$$7305_{10}$$

Binary Numbers

- we can use any positive integer larger than 1 for the radix
- in electronic circuits, it is cheapest to build devices and circuits that distinguish between two stable voltage levels
- it's easy to build devices and circuits that distinguish more levels, such as three or four, but they are much more expensive

- therefore, all normal computers are binary, made to use only two values, so binary numbers are hugely important in CS

$$\begin{aligned} 1101_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 13_{10} \end{aligned}$$

- 1101_2 and 13_{10} are exactly the same value
- they are simply expressed in two different notation systems

Binary and Decimal

- binary numbers are essential for working with computers
- but:
  - humans deal with binary values very poorly

$$11110101011011111010101010101101$$

  - and, there is no obvious correlation between binary and decimal

$$1101_2 = 13_{10}$$

- hexadecimal to the rescue

Bytes

- eight bits is one byte
- one-half of a byte, four bits, is a nibble
- there's nothing magic about a byte, it's just a convenient grouping
- nibbles make it easier for humans to see a byte's value

11010110 vs 1101 0101

- a nibble is a comfortable number of bits to see

Hexadecimal

- hexadecimal numbers are base-16 numbers
- this requires 16 different digits
- but only 10 digits exist in the Hindu-Arabic system
- so to represent hexadecimal numbers, we use the ten decimal digits 0 – 9 that have the same values in base-10 and base-16
- plus the six letters a – f, which have the decimal values 10 – 15 respectively

- thus we have

$$\begin{aligned} 7\mathrm{b}05 &= 7 \times 16^3 + \mathrm{b} \times 16^2 + 0 \times 16^1 + 5 \times 16^0 \\ &= 7 \times 4096_{10} + 11_{10} \times 256_{10} + 0 \times 16_{10} + 5 \times 1 \\ &= 28672_{10} + 2816_{10} + 0 + 5 \\ &= 31493_{10} \end{aligned}$$

- as with binary, there's no obvious correlation between hexadecimal and decimal

Binary and Hexadecimal

- the reason we care about hexadecimal is because of nibbles
- since one nibble represents one of 16 values, one nibble is exactly one hexadecimal digit
- thus a byte, which is 2 nibbles or 8 bits, is exactly 2 hex digits

$$1101010100101011_2 = \mathrm{c52b}_{16}$$

Binary $\leftrightarrow$ Hex Conversion

		$2^0 = 1$
		$2^1 = 2$
$0000 = 0$	$1000 = 8$	$2^2 = 4$
$0001 = 1$	$1001 = 9$	$2^3 = 8$
$0010 = 2$	$1010 = \text{a}\ (10)$	$2^4 = 16$
$0011 = 3$	$1011 = \text{b}\ (11)$	$2^5 = 32$
$0100 = 4$	$1100 = \text{c}\ (12)$	$2^6 = 64$
$0101 = 5$	$1101 = \text{d}\ (13)$	$2^7 = 128$
$0110 = 6$	$1110 = \text{e}\ (14)$	$2^8 = 256$
$0111 = 7$	$1111 = \text{f}\ (15)$	$2^9 = 512$
		$2^{10} = 1024$

Conversion Decimal → Binary

- earlier slides showed how to convert a base-2 or a base-16 representation to decimal
- what about a conversion in the other direction? how to convert a value in decimal notation into binary notation?
- this is done by a series of subtractions: each power of two either contributes to a value or does not
- must know the powers of 2
- it's exactly like making change with coins

example: convert 1304_{10} to binary

Conversion Decimal → Binary

- earlier slides showed how to convert a base-2 or a base-16 representation to decimal
- what about a conversion in the other direction? how to convert a value in decimal notation into binary notation?
- this is done by a series of subtractions: each power of two either contributes to a value or does not
- must know the powers of 2
- it's exactly like making change with coins

example: convert 1304_{10} to binary

$$1304 = 101\,0001\,1000_2$$

- in math, we use the subscript 2 to indicate base-2
- in C++, we use the 0b prefix: 0b10100011000
- in C, we cannot directly represent a binary value, but we can easily represent a hex value with "0x"

Conversion Decimal $\rightarrow$ Hexadecimal

- how to convert a decimal value into hexadecimal?
- the easiest way is to convert decimal $\rightarrow$ binary $\rightarrow$ hexadecimal

example: convert 1304 to hexadecimal

Conversion Decimal → Hexadecimal

- how to convert a decimal value into hexadecimal?
- the easiest way is to convert decimal → binary → hexadecimal

example: convert 1304 to hexadecimal

$$\begin{aligned} 1304 &= 0b101\,0001\,1000 \\ &= 0x518 \end{aligned}$$

Hex Arithmetic

- addresses in memory (and thus pointers) in a computer are expressed in hex notation
- so are ASCII character values
- for the remainder of the semester, we will need to be able to add and subtract hex values
- it's just like normal addition and subtraction except
  - when we carry, we carry 16, not 10
  - when we borrow, we borrow 16, not 10

Hex Arithmetic

example: add `0x518 + 0xe9`

Hex Arithmetic

example: add `0x518 + 0xe9`

example: subtract `0x4a6` – `0x1bf`

using a nibble, we can represent exactly 16 different values

$0000 = 0$

$0001 = 1$

$0010 = 2$

$0011 = 3$

$0100 = 4$

$0101 = 5$

$0110 = 6$

$0111 = 7$

$1000 = 8$

$1001 = 9$

$1010 = \mathsf{a}\,(10)$

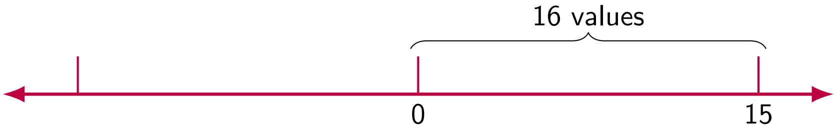
$1011 = \mathsf{b}\,(11)$

$1100 = \mathsf{c}\,(12)$

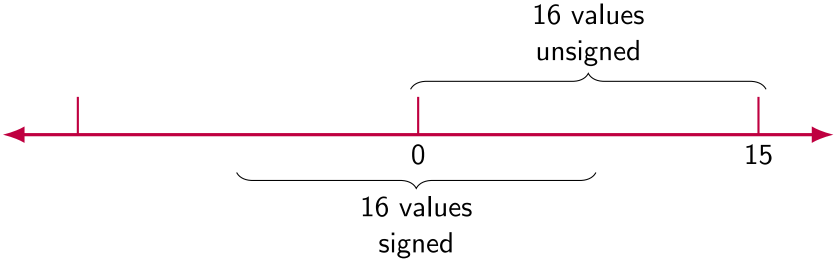
$1101 = \mathsf{d}\,(13)$

$1110 = \mathsf{e}\,(14)$

$1111 = \mathsf{f}\,(15)$



for unsigned values, it is very obvious where the 16 values go if we need signed values, it also seems obvious where they should go



but what is unclear: which bit patterns to represent which values?

Negative Number Encoding

- for characters, there is no "natural" encoding
- 0100 0001 represents 'A' because you and I agree that it does
- for strictly non-negative values, positional binary values make "natural" sense
- but what makes sense for negative numbers?

$$
\begin{aligned}
0000 &= 0 & \qquad 1000 &= ? \\
0001 &= 1 & 1001 &= ? \\
0010 &= 2 & 1010 &= ? \\
0011 &= 3 & 1011 &= ? \\
0100 &= 4 & 1100 &= ? \\
0101 &= 5 & 1101 &= ? \\
0110 &= 6 & 1110 &= ? \\
0111 &= 7 & 1111 &= ?
\end{aligned}
$$

Negative Number Encoding

- various schemes have been proposed, but one is dramatically superior: 2's complement
- every computer uses it
- at first it seems counterintuitive, because $1000 = -8$ seems "smaller" than $1111 = -1$
- note that all nonnegative values start with 0; negatives, with 1

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

2's Complement

- why does 2's complement make sense?
- if you add a positive value and the same value negated, what do you get? 0
- so, using fixed-size binary arithmetic, if we start with a positive binary value, what do we add to it to get zero?

```
  0101 (decimal 5)
+ ????
-------
  0000
```

this value should be -5

2's Complement

- why does 2's complement make sense?
- if you add a positive value and the same value negated, what do you get? 0

- so, using fixed-size binary arithmetic, if we start with a positive binary value, what do we add to it to get zero?

```
  0101 (decimal 5)
+ ????
-------
  0000
```

this value should be -5

- ???? $=$ 1011

Conversion

- you must be able to convert decimal $\longleftrightarrow$ 2's complement
- for non-negative values, it's the same as simple binary
- for negative values, most explanations (and your textbook) kind of go into the weeds
- practice some of these conversions, and learn the pattern

Conversion Decimal $\rightarrow$ 2's Complement

- represent -4_{10} in a nibble

```
   0100 (decimal positive 4)
 + ???? (should be decimal -4)
-------
   0000
```

- what value do you put in to make the answer 0?

Conversion Decimal $\rightarrow$ 2's Complement

- represent -4_{10} in a nibble

```
   0100 (decimal positive 4)
 + ???? (should be decimal -4)
-------
   0000
```

- what value do you put in to make the answer 0?
- 1100

Conversion 2's Complement → Decimal

- what decimal does 2's complement `1010` represent?

```
   1010 (unknown 2's complement value)
 + ???? (the positive complement)
-------
   0000
```

- what value do you put in to make the answer 0?

Conversion 2's Complement → Decimal

- what decimal does 2's complement `1010` represent?

```
  1010 (unknown 2's complement value)
+ ???? (the positive complement)
-------
  0000
```

- what value do you put in to make the answer 0?
- $0110 = 6$
- so, the original was -6

Bigger Storage Locations

- we have been speaking of nibbles
- what if you have an 8-bit storage location?
- or 32-bit, or 64-bit?

- this is the first huge advantage of 2's complement
- in a 4-bit (nibble) location −5 = `1011`
- in an 8-bit (byte) location −5 = `1111 1011`
- in a 16-bit location −5 = `1111 1111 1111 1011`

- in a 4-bit (nibble) location 5 = `0101`
- in an 8-bit (byte) location 5 = `0000 0101`
- in a 16-bit location 5 = `0000 0000 0000 0101`

- this is called sign extension
- in decimal a check for $23.81, is the same as $0023.81

Signed Arithmetic

- the second huge advantage of 2's complement is that arithmetic just works
- in 2's complement binary, add $-6 + 3$

Signed Arithmetic

- the second huge advantage of 2's complement is that arithmetic just works
- in 2's complement binary, add $-6+3$

```
   1010  (-6)
 + 0011  (3)
------------
   1101  (-3)
```

Overflow and Underflow

- think about all possible signed values in a nibble

$$-8,\ -7,\ -6,\ \cdots,\ -1,\ 0,\ 1,\ \cdots,\ 6,\ 7$$

- adding a positive and negative value can never exceed the storage value size
- the result is always correct

Overflow and Underflow

- adding two positive values may overflow

```
    0101 (5)
 +  0100 (4)
------------
    1001 (-7 oops, should be 9)
```

- but in a nibble, 9 cannot be represented

Overflow and Underflow

- adding two negative values may underflow

```
    1001 (-7)
 +  1010 (-6)
--------
    0011 (3 oops, should be -13)
```

Different Sizes

- what if you try to add two signed integers stored in different sized containers?

```
  0000 1101 (d or 13 in a byte)
+      1101 (-3 in a nibble)
------------
```

Different Sizes

- what if you try to add two signed integers stored in different sized containers?

```
  0000 1101 (d or 13 in a byte)
+      1101 (-3 in a nibble)
------------
```

- simply cannot be done correctly
- instead must sign-extend the smaller to the size of the larger

```
  0000 1101 (d or 13 in a byte)
+ 1111 1101 (-3 in a byte)
------------
  0000 1010 (a or 10 in a byte)
```