Lecture 4: Number representation and Bitwise operators

Vikram Iyer

Adapted from material by Blake Hannaford and Justin Hsia

Administrative

- Lab kit pickup
  Mon-Fri: 9am-12pm and 1pm-5pm
- Lab partner sign up due end of Monday 4/8
- C Programming Assignment 1

CSE/ECE 474: Embedded Systems | Home | Syllabus | Calendar | Assignments | Ed Discussion | 24sp

C Programming Assignments

C Programming 1

Due Apr 12 by 11:59pm on Canvas

Instructions, Assignment files (zip)

08	09	10	11	12
9:30-11:30 OH (Deeksha) ECE 345	13:00-15:00 OH (Zach) ECE 345	9:30-11:30 OH (Deeksha) ECE 345	10:00-12:00 OH (Zach) ECE 345	12:30-14:20 Lecture MOR 230 *Lecture 6: Working with Registers and IMU Demo*
11:30-13:30 OH (Alex) ECE 345		12:30-14:20 Lecture MOR 230 *Lecture 5: Hardware and Machine Organization*	12:30-14:30 OH (Alex) ECE 345	14:00-15:00 OH (Vikram) ECE 345
		14:00-15:00 OH (Vikram) ECE 345		23:59 C Programming 1 due

Last time

- Pointers and memory
  - How is your data actually stored in memory?
  - Introduction to memory addresses
  - Pointers and pointer operations

Plan for today

- Allocating memory
- Number representation
  - Overview of Binary and Hex
  - Byte ordering
  - Endianness
  - Encoding integers
- Logical operators
  - Boolean logic
  - Bitwise operators

Allocating Memory in C: malloc()

- Last time we talked about pointers and memory,
  - We can declare an array of fixed size at compile time, but how do you *dynamically* allocate a block of memory while a program is running?

- On a normal CPU, dynamic memory allocation with: malloc()

  Declaration in library file: `void *malloc(size_t size);`

```
ptr = (float*) malloc(100 * sizeof(float));
```

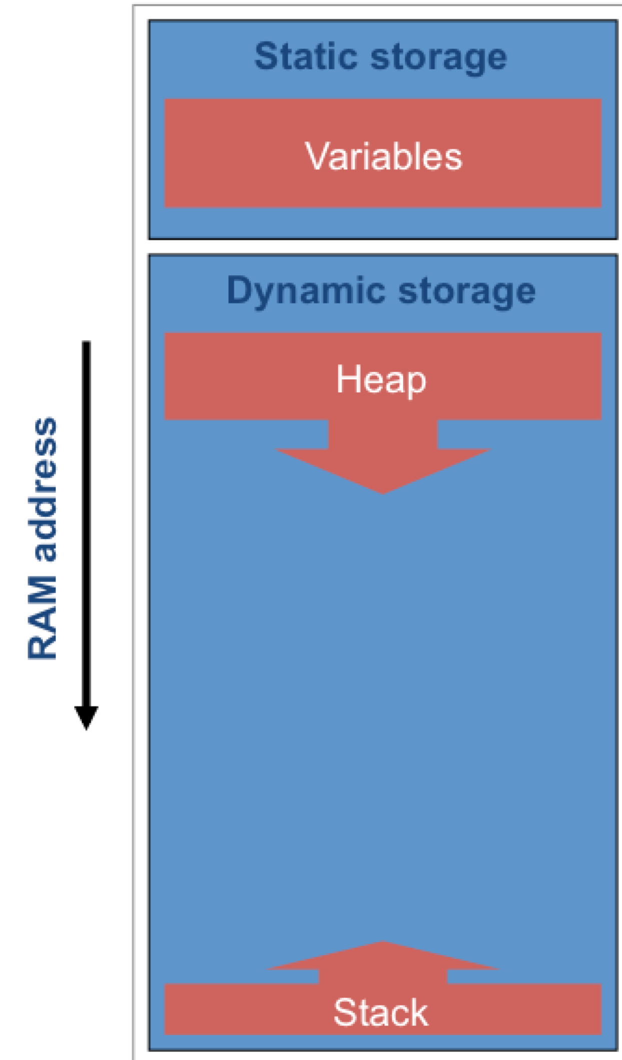
  - When you're done using that memory we have to explicitly call `free()`

```
free(ptr);
```

- **In embedded programming we usually don't do this**

Memory allocation on microcontrollers

- The function is commonly not re-entrant (thread friendly)
  - Challenging for use in real-time operating system
- Its performance is not deterministic
  - Time to allocate a memory block may vary
  - Challenge in a real-time application.
- Memory allocation may fail
  - If you run out of memory or memory is fragmented
- **malloc() is slow**



Number representation

- How does our Arduino represent numbers?
  - *Humans* think about numbers in **base 10**, but *computers* represent numbers in **base 2**
  - Binary encoding is what allows computers to do all of the amazing things that they do!
  - Hexadecimal (Hex) encoding is **base 16**

- Helpful to memorize this table for converting between them

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Binary Encoding

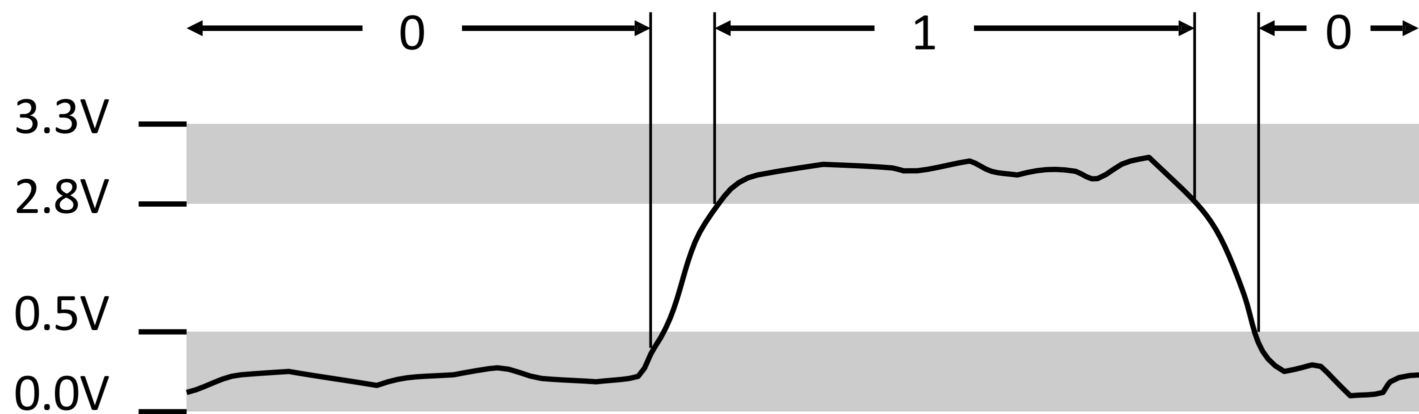
- With n binary digits, how many "things" can you represent?
  - Need n binary digits to represent N things, where $2^n \geq N$
  - Example: 5 binary digits for alphabet because $2^5 = 32 > 26$
- A binary digit is known as a bit
- A group of 4 bits (1 hex digit) is called a nibble
- A group of 8 bits (2 hex digits) is called a byte
  - 1 bit $\rightarrow$ 2 things, 1 nibble $\rightarrow$ 16 things, 1 byte $\rightarrow$ 256 things

	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
MSB	0	0	0	1	0	0	0	1	LSB

MSB = Most significant bit
LSB = Least significant bit

Why do we use Base 2?

- Electronic implementation
  - Easy to store with bi-stable elements
  - Reliably transmitted on noisy and inaccurate wires

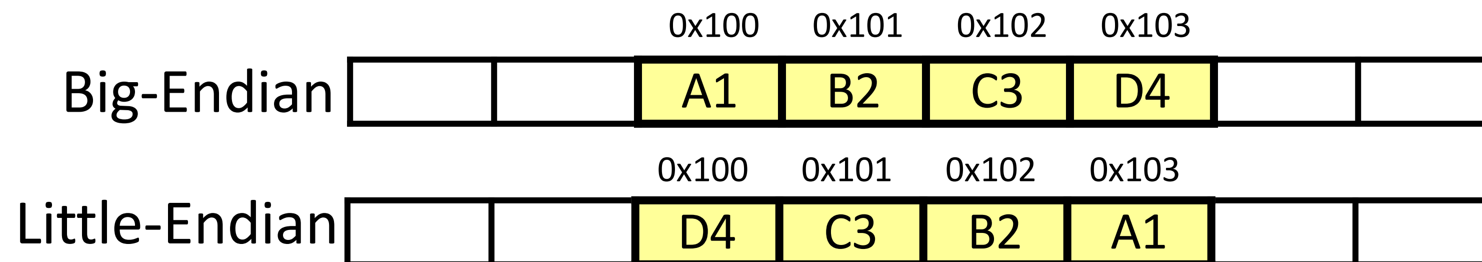


- Other bases possible, but not yet viable:
  - DNA data storage (base 4: A, C, G, T) is hot @UW
  - Quantum computing

Byte Ordering

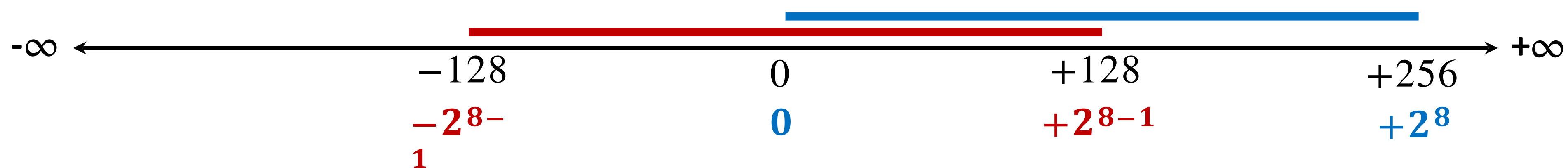
- How should bytes within a word be ordered *in memory?*
  - Want to keep consecutive bytes in consecutive addresses
  - **Example:** store the 4-byte (32-bit) `float: 0x A1 B2 C3 D4`
- By convention, ordering of bytes in hardware called *endianness*
  - The two options are big-endian and little-endian

Example: 4-byte data 0xA1B2C3D4 at address 0x100



Encoding Integers

- The hardware (and C) supports two flavors of integers
  - *unsigned* – only the non-negatives
  - *signed* – both negatives and non-negatives
- Cannot represent all integers with w bits
  - Only 2^w distinct bit patterns
  - Unsigned values: $0 \ldots 2^w-1$
  - Signed values: $-2^{w-1} \ldots 2^{w-1}-1$
- **Example:** 8-bit integers (*e.g.*, `char`)



Unsigned Integers

- Unsigned values follow the standard base 2 system
  - $b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0 = b_7\, 2^7 + b_6\, 2^6 + \cdots + b_1\, 2^1 + b_0\, 2^0$

```
unsigned int number = 17;
uint8_t number = 17;
```

		32	16	8	4	2	1	=17
		2^5	2^4	2^3	2^2	2^1	2^0	
0	0	0	1	0	0	0	1	

- Useful formula: $2^{N-1} + 2^{N-2} + \ldots + 2 + 1 = 2^N - 1$
  - *i.e.*, N ones in a row = $2^N - 1$
  - *e.g.*, 0b111111 = 63

Sign and Magnitude

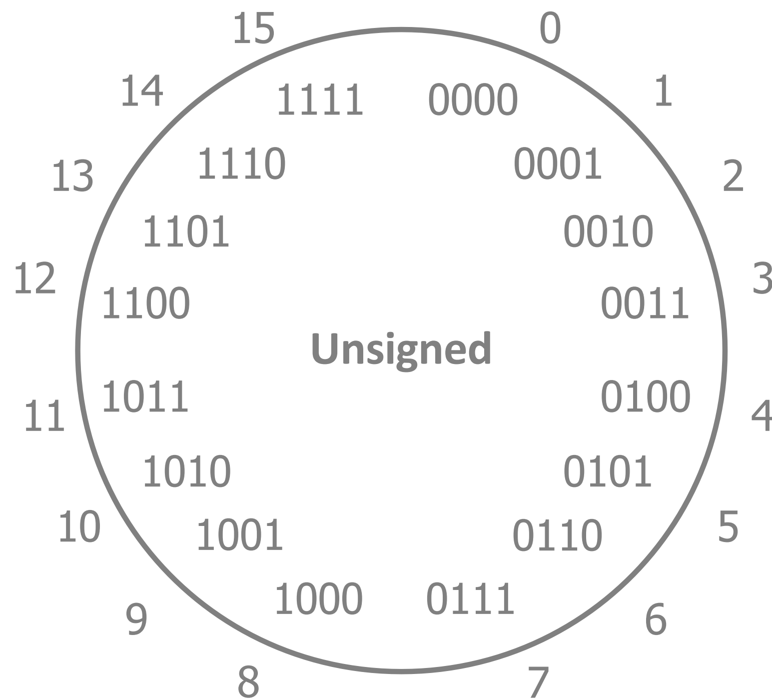
Not used in practice for integers!

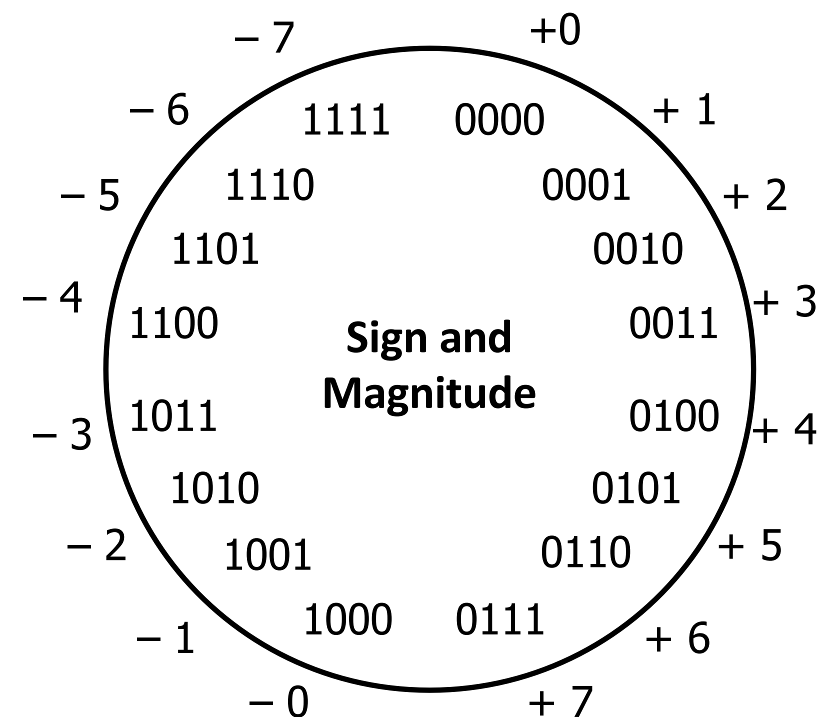
- Designate the high-order bit (MSB) as the "sign bit"
  - `sign=0`: positive numbers; `sign=1`: negative numbers
- Benefits:
  - Using MSB as sign bit matches positive numbers with unsigned
  - All zeros encoding is still = 0
- Examples (8 bits):
  - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
  - 0x7F = 01111111_2 is non-negative ($+127_{10}$)
  - 0x85 = 10000101_2 is negative (-5_{10})
  - 0x80 = 10000000_2 is negative... zero???

Sign and Magnitude

Not used in practice for integers!

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks?

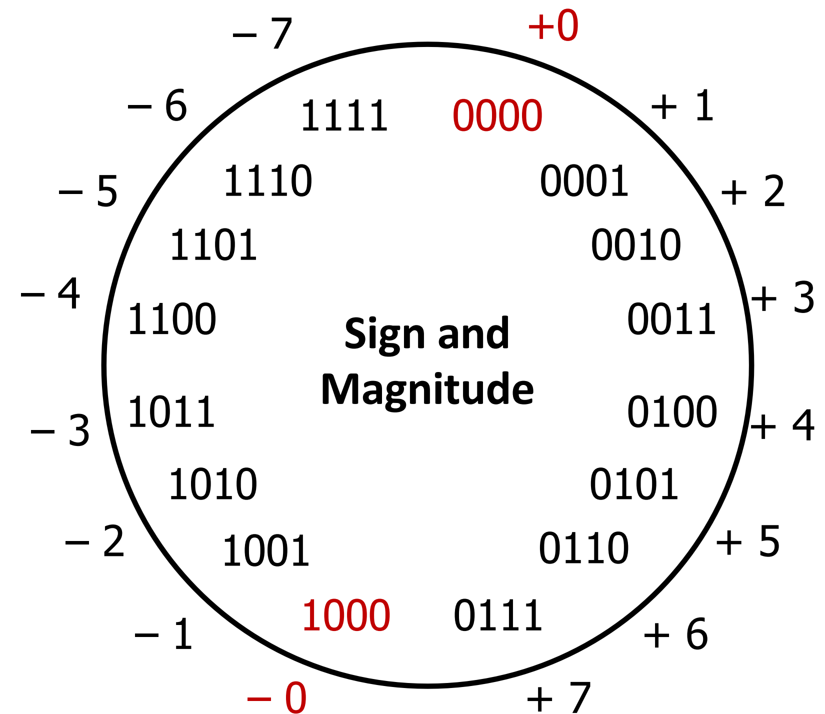




Sign and Magnitude

Not used in practice for integers!

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)



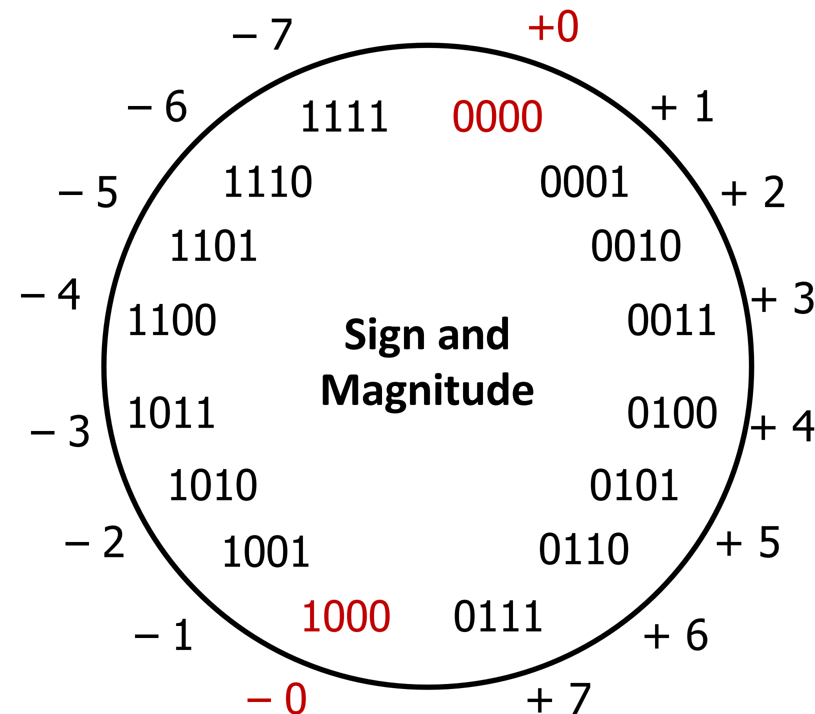
Sign and Magnitude

Not used in practice for integers!

- MSB is the sign bit, rest of the bits are magnitude
- Drawbacks:
  - Two representations of 0 (bad for checking equality)
  - Arithmetic is cumbersome
    - Example: `4-3 != 4+(-3)`

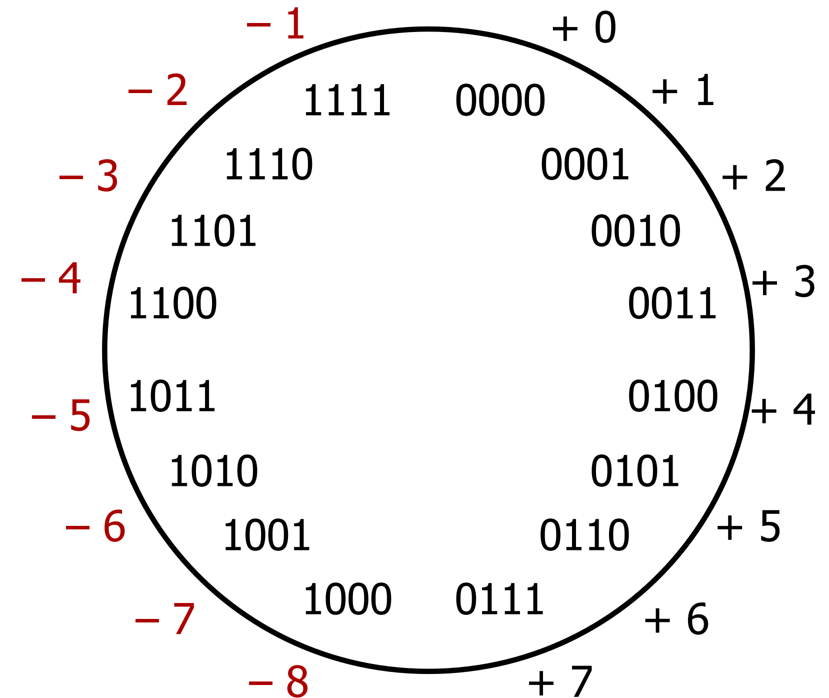
4	0100	4	0100
− 3	− 0011	+ -3	+ 1011
1	0001	-7	1111

    - Negatives “increment” in wrong direction!



Two's Complement

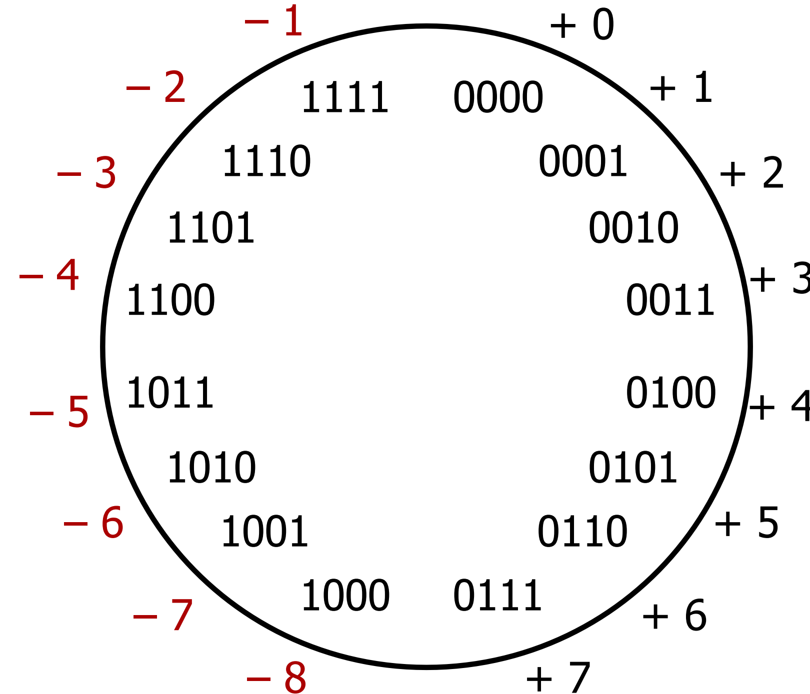
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works



Two's Complement

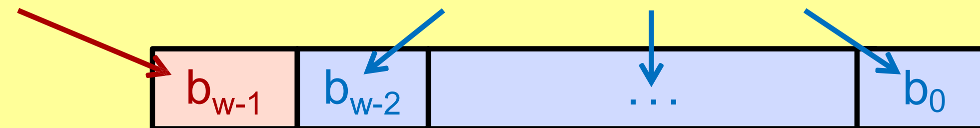
- Let's fix these problems:
  1) "Flip" negative encodings so incrementing works
  2) "Shift" negative numbers to eliminate –0

- MSB *still* indicates sign!
  - This is why we represent one more negative than positive number (-2^{N-1} to 2^{N-1} –1)

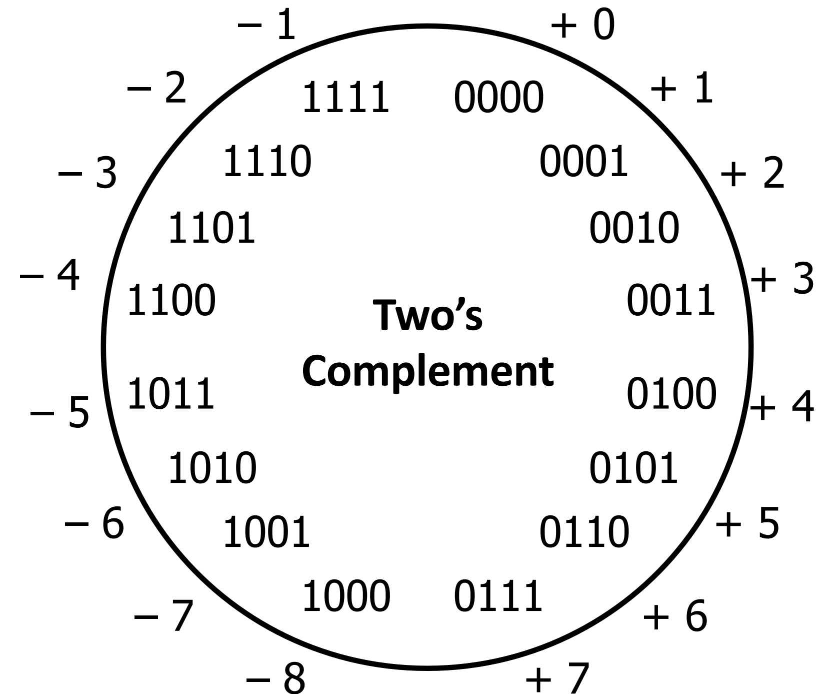


Two's Complement Negatives

- Accomplished with one neat mathematical trick!



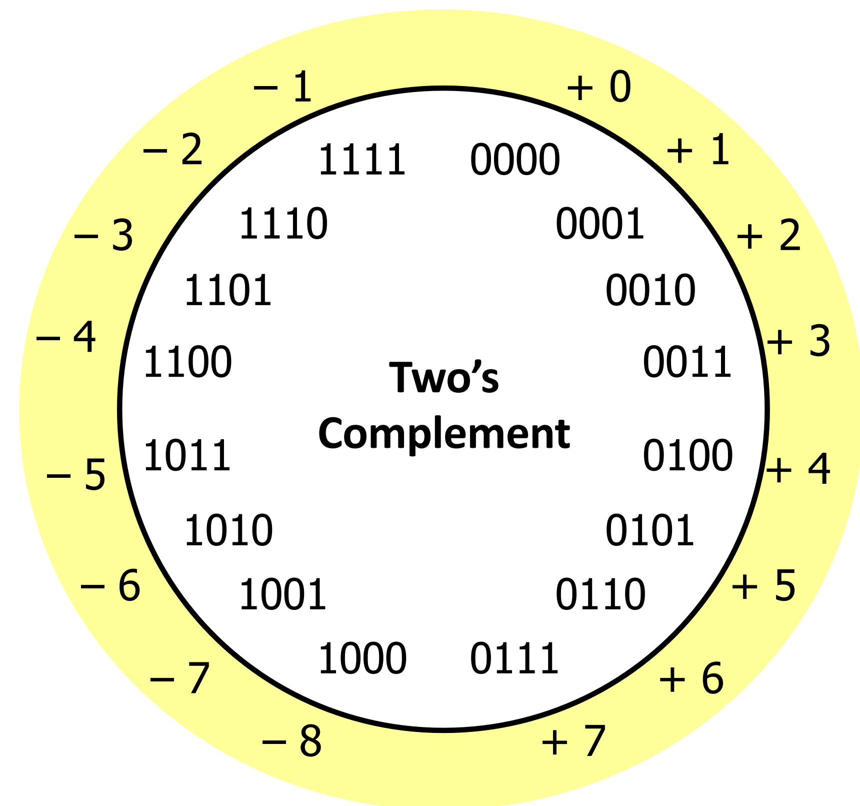
- 4-bit Examples:
  - 1010_2 unsigned:
    $1*2^3+0*2^2+1*2^1+0*2^0$ = **10**
  - 1010_2 two's complement:
    $-1*2^3+0*2^2+1*2^1+0*2^0$ = **–6**
- -1 represented as:
  $1111_2 = -2^3+(2^3 – 1)$
  - MSB makes it super negative, add up all the other bits to get back up to -1



Two's Complement is Great

- Roughly same number of (+) and (–) numbers
- Positive number encodings match unsigned
- Single zero
- All zeros encoding = 0

- Simple negation procedure:
  - Get negative representation of any integer by taking bitwise complement and then adding one!
    `( ~x + 1 == -x )`



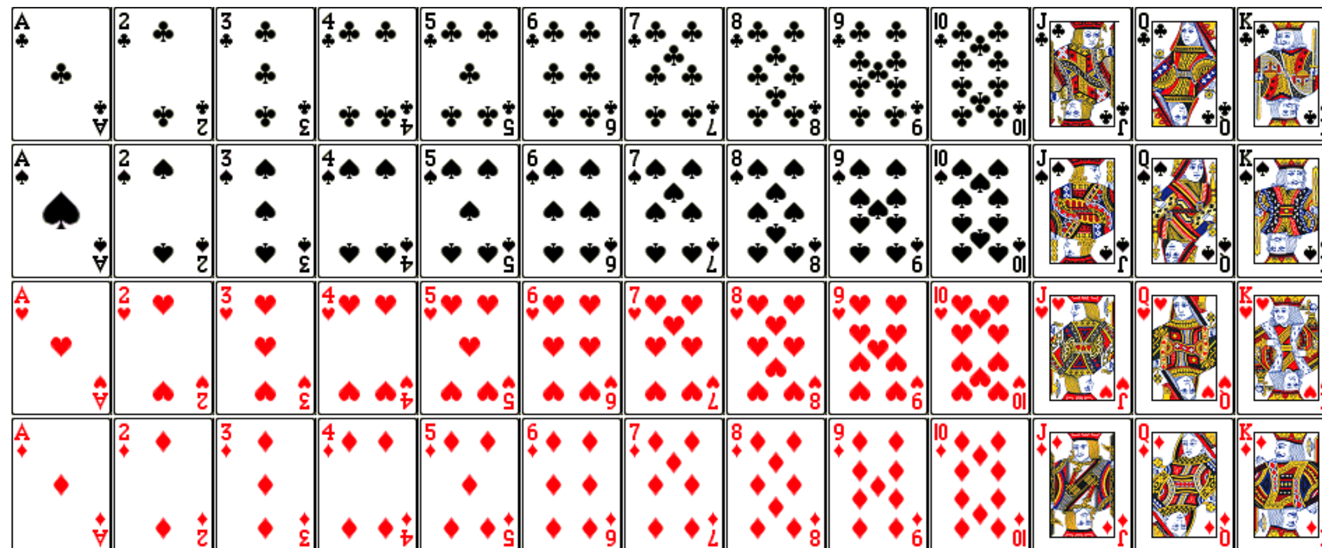
Numerical Encoding Design Example

Encode a standard deck of playing cards

- 52 cards in 4 suits

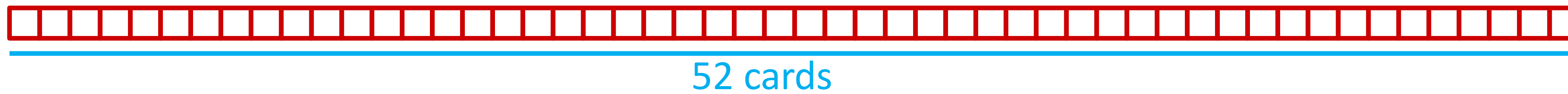
Operations to implement:

- Which is the higher value card?
- Are they the same suit?

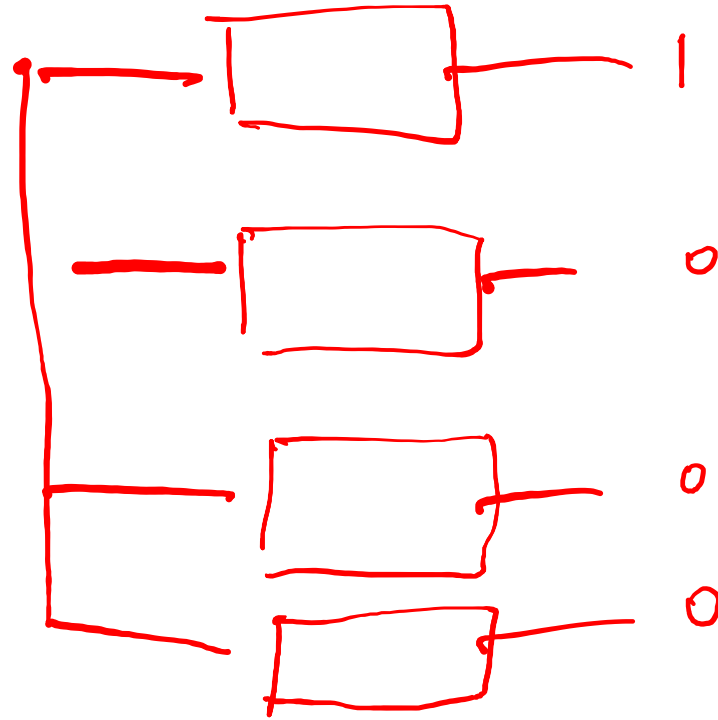


Representations and Fields

1) 1 bit per card (52): bit corresponding to card set to 1

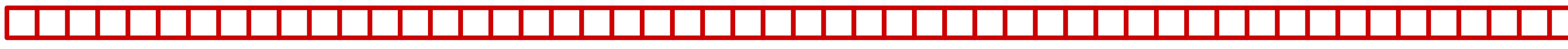


- "One-hot" encoding (similar to set notation)



Representations and Fields

1) 1 bit per card (52): bit corresponding to card set to 1

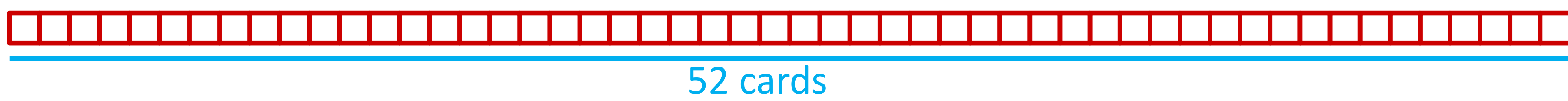


52 cards

- "One-hot" encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

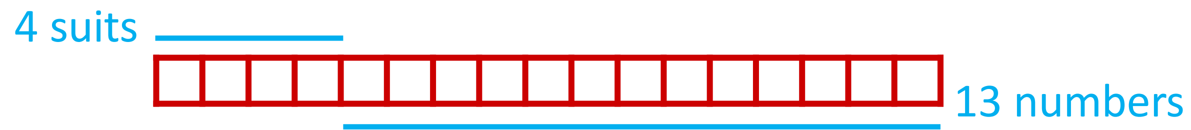
Representations and Fields

1) 1 bit per card (52): bit corresponding to card set to 1



- "One-hot" encoding (similar to set notation)
- Drawbacks:
  - Hard to compare values and suits
  - Large number of bits required

2) 1 bit per suit (4), 1 bit per number (13): 2 bits set

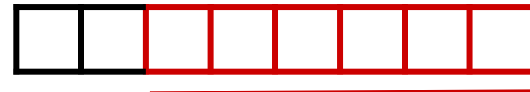


- Pair of one-hot encoded values
- Easier to compare suits and values, but still lots of bits used

Representations and Fields

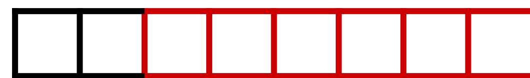
3) Binary encoding of all 52 cards – only 6 bits needed

- $2^6 = 64 \geq 52$



- Fits in one byte (smaller than one-hot encodings)
- How can we make value and suit comparisons easier?

4) Separate binary encodings of suit (2 bits) and value (4 bits)



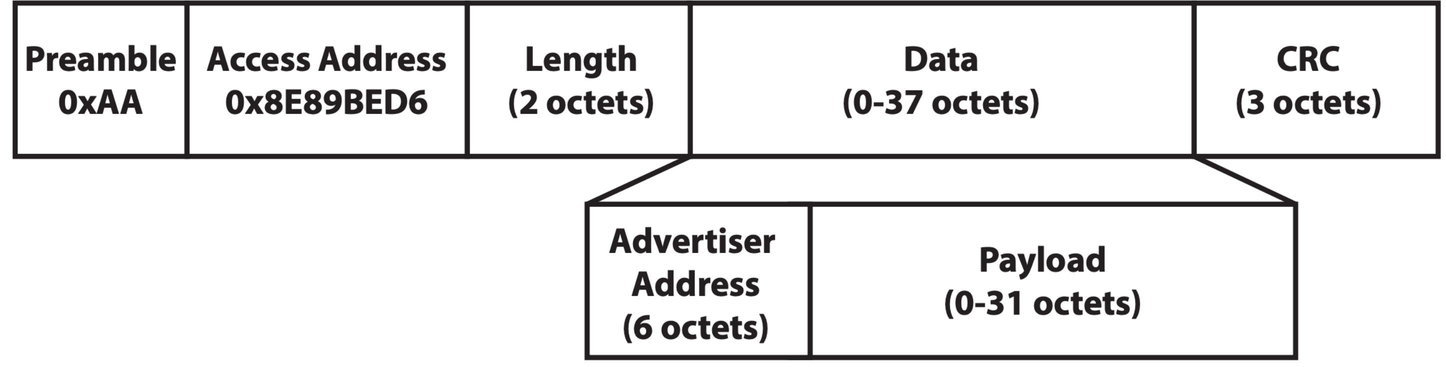
- Also fits in one byte, and easy to do comparisons

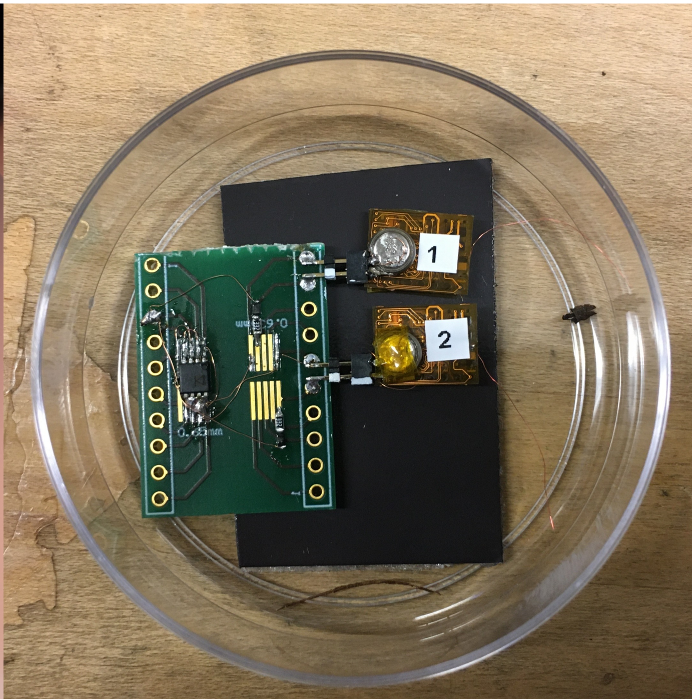
♣	00
♦	01
♥	10
♠	11

K	Q	J	. . .	3	2	A
1101	1100	1011	. . .	0011	0010	0001

Why go through all this effort to save space?

Example: Bluetooth Beacons





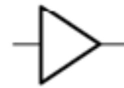
Manipulating bits: Logical operators

True	a non zero value
False	zero
&&	**AND**
`A && B`	True if *both* `A` and `B` are non-zero.
`||`	**OR**
`A || B`	True if *either* `A` and `B` are non-zero.
`!`	**NOT**

Logical operators

True	a non zero value.
False	zero.
&&	**AND**
`A && B`	True if *both* `A` and `B` are non-zero.
||	**OR**
`A || B`	True if *either* `A` and `B` are non-zero.
!	**NOT**

YES

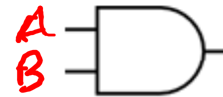


INPUT	OUTPUT
A	
0	0
1	1

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND



	INPUT		OUTPUT
	A	B	
False	0	0	0
	1	0	0
	0	1	0
True	1	1	1

OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

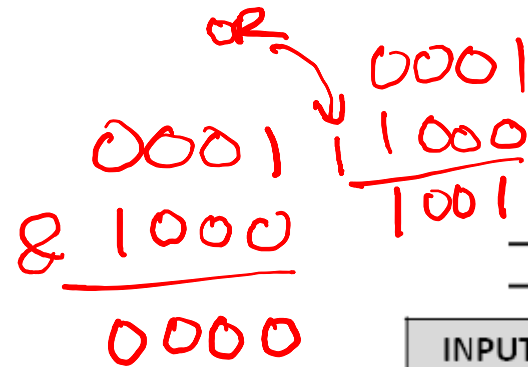
XOR

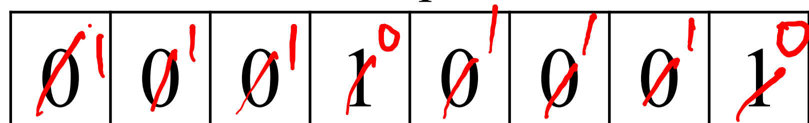
INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

Bitwise operators

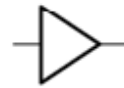
A very important C feature for embedded microcomputer systems. These operators are only defined for integer-like variables i.e. `char`, `short`, `int`, and `long` signed or unsigned.

&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	right shift
~	ones complement





YES

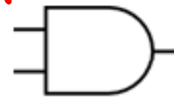


INPUT	OUTPUT
A	
0	0
1	1

NOT

INPUT	OUTPUT
A	
0	1
1	0

AND



INPUT		OUTPUT
A	B	
0	0	0
1	0	0
0	1	0
1	1	1

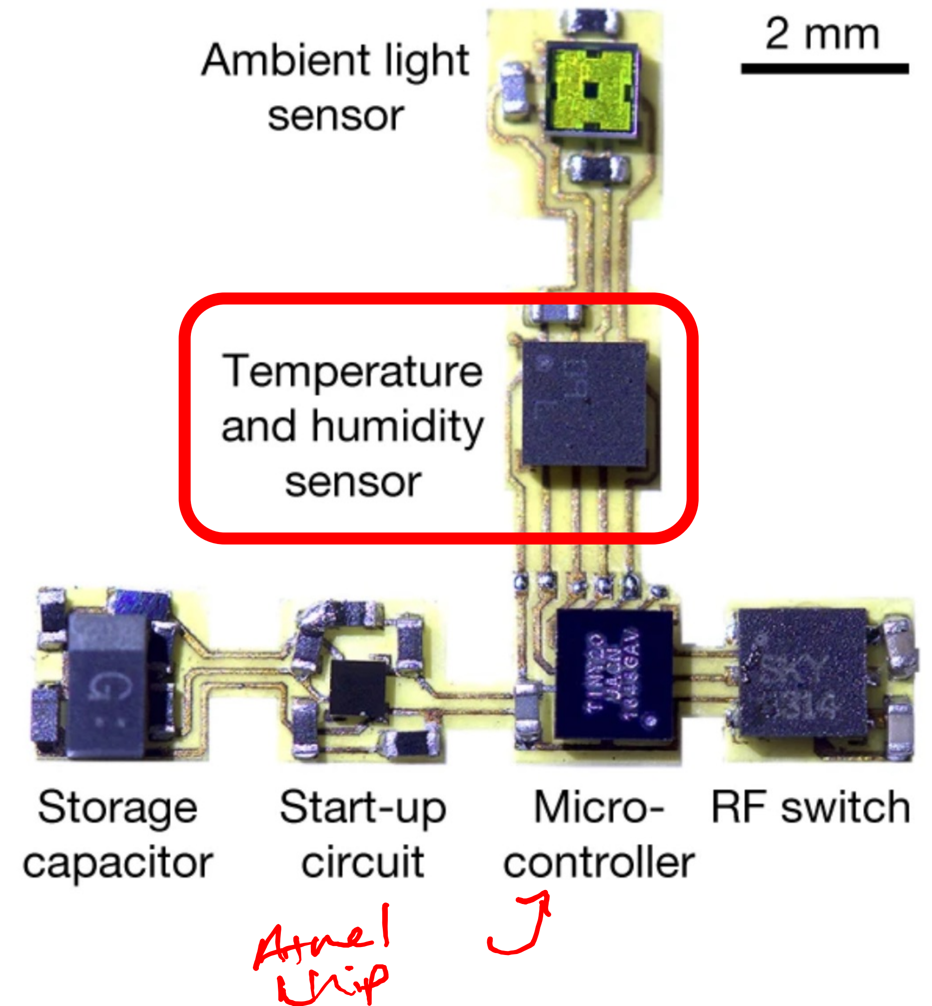
OR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	1

XOR

INPUT		OUTPUT
A	B	
0	0	0
1	0	1
0	1	1
1	1	0

Example: Wind dispersed wireless sensors



Supplementary Video 5

Outdoor wind dispersal

Datasheet

HDC2010
SNAS693D – JULY 2017 – REVISED FEBRUARY 2021

HDC2010 Low-Power Humidity and Temperature Digital Sensors

1 Features

- Relative humidity range: 0% to 100%
- Humidity accuracy: ±2%
- Sleep current: 50 nA
- Average supply current (1 measurement per second)
  - RH only (11 bit): 300 nA
  - RH (11 bit) + temperature (11 bit): 550 nA
- Temperature range:
  - Operating: –40°C to 85°C
  - Functional: –40°C to 125°C
- Temperature accuracy: ±0.2°C typical
- Supply voltage: 1.62 V to 3.6 V
- Programmable sampling rate (5 Hz, 2 Hz, 1 Hz, 0.2 Hz, 0.1 Hz, 1/60 Hz, 1/120 Hz) or trigger on demand
- I^2C interface

3 Description

The HDC2010 is an integrated humidity and temperature sensor that provides high accuracy measurements with very low power consumption, in an ultra-compact WLCSP (Wafer Level Chip Scale Package). The sensing element of the HDC2010 is placed on the bottom part of the device, which makes the HDC2010 more robust against dirt, dust, and other environmental contaminants. The capacitive-based sensor includes new integrated digital features and a heating element to dissipate condensation and moisture. The HDC2010 digital features include programmable interrupt thresholds to provide alerts/ system wake-ups without requiring a microcontroller to be continuously monitoring the system. This, combined with programmable sampling intervals, low inherent power consumption, and support for 1.8-V supply voltage, make the HDC2010 well suited for battery-operated systems.

Datasheet

7.6.1 Address 0x00 Temperature LSB

Table 7-7. Address 0x00 Temperature LSB Register

7	6	5	4	3	2	1	0
TEMP[7:0]							

Table 7-8. Address 0x00 Temperature LSB Field Descriptions

BIT	FIELD	TYPE	RESET	DESCRIPTION
[7:0]	TEMPERATURE [7:0]	R	00000000	Temperature LSB

7.6.2 Address 0x01 Temperature MSB

The temperature register is a 16-bit result register in binary format (the 2 LSBs D1 and D0 are always 0). The result of the acquisition is always a 14-bit value, while the resolution is related to one selected in Measurement Configuration register. The temperature must be read LSB first.

Table 7-9. Address 0x01 Temperature MSB Register

7	6	5	4	3	2	1	0
TEMP[15:8]							

Table 7-10. Address 0x01 Temperature MSB Field Descriptions

BIT	FIELD	TYPE	RESET	DESCRIPTION
[15:8]	TEMPERATURE [15:8]	R	00000000	Temperature MSB

The temperature can be calculated from the output data with Equation 1:

$$\text{Temperature } (^{\circ}\text{C}) = \left(\frac{\text{TEMPERATURE } [15:0]}{2^{16}}\right) \times 165 - 40 \quad (1)$$

```
bool readSample()
{
  uint8_t data[6];
  Wire.beginTransmission(0x40);
  Wire.write(byte(0x0F));
  Wire.write(byte(0x01));
  Wire.endTransmission();

  Wire.beginTransmission(0x40);
  Wire.write(byte(0x00));
  Wire.endTransmission();
  Wire.requestFrom(0x40, 4);
  data[0] = Wire.read();
  data[1] = Wire.read();
  data[2] = Wire.read();
  data[3] = Wire.read();
```

```
  // … example continued
  // convert to Temperature/Humidity
  uint16_t val;
  val = (data[1] << 8) + data[0];
  Serial.print(
   ((float)val*165)/65536-40);
  Serial.print(", ");

  val = (data[3] << 8) + data[2];
  Serial.println(
    ((float)100*val)/65536);

  return true;
}
```