



Digital representation workshop

Bits and bytes and all that



Prerequisites

We assume that you have read the following prior to attending this workshop:

- Kernigan - Chapter 2 - “Bits, Bytes and Representation of Information” (pp. 21-34)
- Juneday wiki - http://wiki.juneday.se/mediawiki/index.php/Computing:Binary_representation

And seen:

- [Binary Representation \(Full playlist\)](#) [Binary Representation - Text \(Full playlist\)](#)

00000101 00110100 11000001 00010110 00100100 00101001 00010001 01001100 00001100 000
01010111 10000000 10000111 10010001 11100111 11000100 01111110 10101000 00001100 000
10000000 00110011 11101110 00101011 01111110 11010111 10000111 00100011 10010100 000
00110001 10011001 01110011 01110110 01101110 10010100 00101001 00000000 01101110 000
11001100 01100000 00100111 01110100 00011011 00111000 11001001 00000000 10110110 100
10000000 10111101 11000111 00111011 01011111 10001110 00110101 01111011 00100110 001
11001010 10000000 01100010 00010111 10110101 11110010 00011001 10011000 11010000 100
11111101 00111111 01000010 10111101 00111101 10011001 0100110 10011101 11001110 110
10101001 00001001 11110100 00110101 10011001 00111101 001111 00100010 01100100 101
00011011 10000110 11011010 10011001 01111001 10011001 010011 10010001 01111000 110
01010100 01111111 10101011 00111001 01111001 01111001 110000 11101100 10101111 111
01000010 11110101 01111110 01110101 01111001 01111001 011001 11100110 01111100 101
00111001 00111001 00000111 01111001 01111001 01111001 001100 11110000 11001110 011
10101000 00111100 11000111 11101101 01111001 01111001 110101 10110001 11100010 000
00100000 01011110 00100111 10101100 01111001 11001110 01001111 00011011 011
01111000 10101011 01111011 00111000 01011010 01110100 10100001 11101110 01101110 011
00101111 01001011 01000100 11111101 10100011 00111100 01111001 11111011 10101100 100
10101011 11011001 10001010 11010111 11010011 01100000 10101110 01101011 11000000 000
00110000 11111111 11010111 10011100 01110111 11101111 10110000 10101111 11110101 111
1110100 10101111 01111111 11101101 01111000 10010000 10001011 11111011 11000100 000

DATA

Objectives

Understand

- how information is represented in computers
- the binary number system
- how numbers can be used to represent anything - like text for instance

Know

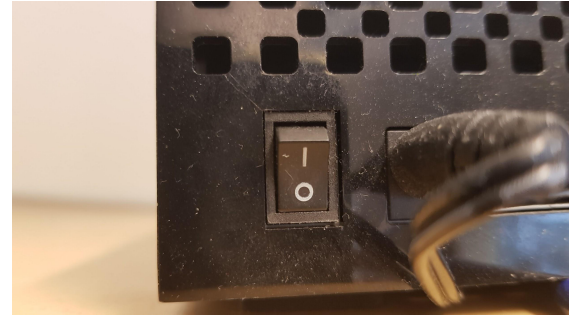
- how to represent text as binary numbers
- how to convert between different bases

Divide yourselves in groups

- 4-5 persons in each group
- You will work together during the workshop

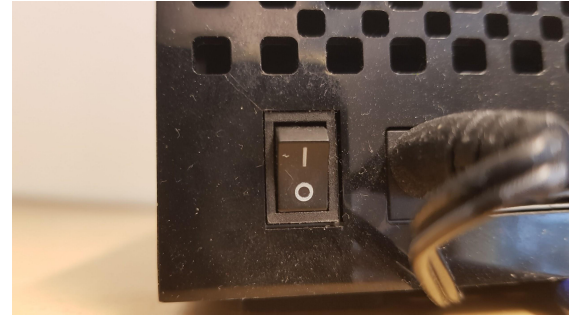
Binary basics

- How much information can we store in one bit (0 | 1)
 - Why?
 - Examples of one bit information
- Number of combinations in n bits
 - How many combinations are there in 10 bits?
- How many bits do we need to store one million numbers?
- A ball park figure for 2^{32} ?
 - Recap of multiplication of exponentiations, e.g. $2^n 2^m = 2^{n+m}$
- Discuss in your groups and agree on an answer



Binary basics - suggested solution

- How much information can we store in one bit (0 | 1)
 - Why?
 - Examples of one bit information
- We can store two states, like On/Off, odd/even etc
- This is because 2^1 is two



Binary basics - suggested solution

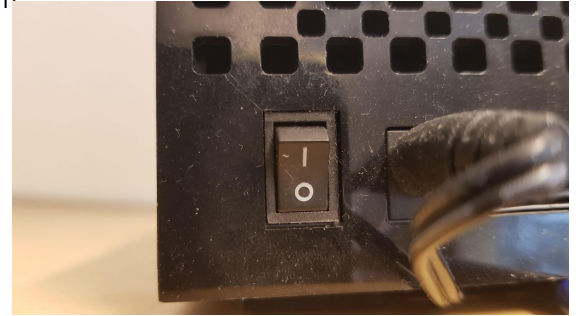
- Number of combinations in n bits
 - How many combinations are there in 10 bits?
- 10 bits can represent 1024 different things, like numbers. Remember this number, it is good for estimates.
- $2^4 = 16$.
 $2^6 = 64$.
 $2^{10} = 2^4 * 2^6 = 2^{4+6}$.
 $2^4 * 2^6 = 16 * 64 = 1024$.
- Or:
- $2*2*2*2*2*2*2*2*2*2 = 1024$.

Binary basics - suggested solution

- How many bits do we need to store one million numbers?
- We need 20 bits, since 2^{20} is the smallest power of two, that is larger than or equal to one million. How could we use previous knowledge to solve this?
- We learnt that 2^{10} was 1024. We know that $1024 * 1024 \approx 1\,000\,000$.
- We know that $2^{20} = 2^{10} * 2^{10}$.
- Answer: 20 bits can represent over 1 000 000 numbers.
- Exact answer: 20 bits can represent 1 048 576 numbers.

Binary basics - suggested solution

- A ball park figure for 2^{32} ?
 - Recap of multiplication of exponentiations, e.g. $2^n 2^m = 2^{n+m}$
- About 4 000 000 000, because:
- $2^{10} * 2^{10} * 2^{10} * 2^2 = 2^{10+10+10+2}$.
- $2^{10} * 2^{10} * 2^{10} \approx 1\ 000\ 000\ 000$.
- $2^2 = 4$.
- Answer: about 4 000 000 000
- Exact answer: 4 294 967 296



Binary numbers - converting to decimal

- Just like decimal numbers, the most significant digit is to the left.
- Using four bits for unsigned numbers, the rightmost bit is worth 2^0 (1_{10}) if set. The next 2^1 and the next 2^2 and the next (leftmost) 2^3 .

1010

			`	0	*	2^0
		`	-	1	*	2^1
	`	--		0	*	2^2
`	---			1	*	2^3

$$\text{Sum: } 0_{10} + 2_{10} + 0_{10} + 8_{10} = 10_{10}$$

Binary numbers - converting to decimal

- Figure out what the following unsigned four bit numbers represent converted to decimal:
 - 1111
 - 1110
 - 1101
 - 1100
 - 1011
 - 1010
 - 1001
 - 1000

Binary numbers - converting to decimal - answers

- $1111_2 = (1 + 2 + 4 + 8 = 15)_{10}$
- $1110_2 = (0 + 2 + 4 + 8 = 14)_{10}$
- $1101_2 = (1 + 0 + 4 + 8 = 13)_{10}$
- $1100_2 = (0 + 0 + 4 + 8 = 12)_{10}$
- $1011_2 = (1 + 2 + 0 + 8 = 11)_{10}$
- $1010_2 = (0 + 2 + 0 + 8 = 10)_{10}$
- $1001_2 = (1 + 0 + 0 + 8 = 9)_{10}$
- $1000_2 = (0 + 0 + 0 + 8 = 8)_{10}$
- How many bits do we have to investigate, in order to tell if any binary number is odd or even?

Binary numbers, convert from decimal

Calculate the four bit unsigned representation (pad with 0s if necessary) of the following unsigned numbers:

- 5
- 3
- 7
- 6
- 0
- 2

Binary numbers, convert from decimal - answers

- $5_{10} = (0101)_2$
- $3_{10} = (0011)_2$
- $7_{10} = (0111)_2$
- $6_{10} = (0110)_2$
- $0_{10} = (0000)_2$
- $2_{10} = (0010)_2$

$7/2 = 3$, remainder 1, so least significant bit is 1 -> 1

$3/2 = 1$, remainder 1, so next bit is 1 -> 11

$1/2 = 0$, remainder 1, so next bit is 1 -> 111

Pad with 0 -> 0111

Bonus - bash one liner for eight bit binary

```
i=111; result=$(while ((i/2 > 0));do echo -n $((i % 2));i=$((i/2));done;echo $((i % 2));printf %08d $(echo $result|rev);echo
```

formatted:

```
i=111;
result=$(
  while ((i/2 > 0));
  do
    echo -n $((i % 2));
    i=$((i/2));
  done;
echo $((i % 2));
printf %08d $(echo $result|rev);
echo
```


Over- and underflow

If we are using 8 bits to represent numbers, what happens when we add such numbers together?

Largest unsigned number using 8 bits is $1111\ 1111_2$ (255_{10}).

What happens when we add $0000\ 0011_2$ (3_{10}) to it?

Remember how to add binary numbers:

$$0+0 = 0$$

$$0+1 = 1$$

$$1+0 = 1$$

$$1+1 = 10 \text{ (one in carry)}$$

Over- and underflow

If we are using 8 bits to represent numbers, what happens when we add such numbers together?

Largest unsigned number using 8 bits is $1111\ 1111_2$ (255_{10}).

What happens when we add $0000\ 0011_2$ (3_{10}) to it?

```
11111 111
 1111 1111
+0000 0011
-----
0000 0010 ← uh-oh
```

Signed numbers (2's complement)

- In signed numbers (using 2's complement) the leftmost bit carries the sign
 - 1 means negative number, 0 means positive
 - We'll "lose" one bit, the range becomes $-2^{n-1} .. 2^{n-1}-1$ (n is the number of bits)
- Overflow may change the sign
- 2's complement: if negative, flip the bits and add one
- Example: four bits, -8_{10} is:

1000 ("normal" 8)

0111 (flipped)

0111

+0001

1000 (one added)

Signed numbers overflow

Using 2's complement, what is 7+2 in binary?

$$\begin{array}{r} 0111 \\ +0010 \\ \hline ???? \end{array}$$

Signed numbers overflow

Using 2's complement, what is 7+2 in binary?

```
  0111  
+0010  
      
  1001
```

Negative, so flip bits and add one:

```
  0110  
0001
```

```
0111 (7, so it represents -7)
```

Signed numbers overflow

Wait, whaaat?

```
  0111
+0010
-----
  1001
```

A signed 4 bit number uses one bit (leftmost) as sign bit, and three bits for magnitude (value). So only 1-7 (positive) 0 and -8..-1 negative. So you can't store a value greater than 7 in four bits signed (two's complement).

Representing text in (8 bit) binary format

Characters in a computer can be assigned a number (e.g. ASCII table).

Therefore, we can represent text as a sequence of such numbers binary.

Excerpt (using 8 bit ascii):

Binary	Oct	Dec	Hex		Glyph
0000 1010	012	10	0A		<Line Feed>
...					
0110 0001	141	97	61		a
0110 0010	142	98	62		b
0110 0011	143	99	63		c
0110 0100	144	100	64		d
0110 0101	145	101	65		e

Representing text in (8 bit) binary format

Group exercise: What text does the following binary information represent?

```
01000010 01101001 01101110 01100001 01110010 01111001
00100000 01101110 01110101 01101101 01100010 01100101
01110010 01110011 00100000 01100001 01110010 01100101
00100000 01100110 01110101 01101110 00101110 00001010
01000001 01110010 01100101 01101110 00100111 01110100
00100000 01110100 01101000 01100101 01111001 00111111
```

Hint: <https://en.wikipedia.org/wiki/ASCII> (leading zeros don't show in the table)

Hint: install the `ascii` and `ascii2binary` commands

Representing text in (8 bit) binary format

Group exercise: What text does the following binary information represent?

```
01000010 01101001 01101110 01100001 01110010 01111001
00100000 01101110 01110101 01101101 01100010 01100101
01110010 01110011 00100000 01100001 01110010 01100101
00100000 01100110 01110101 01101110 00101110 00001010
01000001 01110010 01100101 01101110 00100111 01110100
00100000 01110100 01101000 01100101 01111001 00111111
```

```
$ for b in 01000010 01101001 01101110 01100001 01110010 01111001 00100000 01101110 01110101
01101101 01100010 01100101 01110010 01110011 00100000 01100001 01110010 01100101 00100000
01100110 01110101 01101110 00101110 00001010 01000001 01110010 01100101 01101110 00100111
01110100 00100000 01110100 01101000 01100101 01111001 00111111;do echo "$b"|ascii2binary -b
b;done;echo
```

Binary numbers are fun.

Aren't they?

Octal (base 8) Group task.

- Octal numbers have base 8 (digits 0-7)
- $2^3 = 8$ - so three bits are needed to represent one octal digit
- Convenient - you can divide bits in groups of three bits to convert:

16 is in binary 10000.

Grouped in threes: 010 000 which then becomes 20_8 .

32 becomes 100 000, which becomes 40_8 .

What is 65_{10} in octal?

Octal (base 8) Suggested solution

- Octal numbers have base 8 (digits 0-7)
- $2^3 = 8$ - so three bits are needed to represent one octal digit
- Convenient - you can divide bits in groups of three bits to convert:

16 is in binary 10000.

Grouped in threes: 010 000 which then becomes 20_8 .

32 is 100 000 in binary, which becomes 40_8 .

What is 65_{10} in octal?

65 becomes 001 000 001, which becomes 101_8 .

Hexadecimal (base 16) Group task

- Base 16, digits 0-9A-F (A: 10, B:11, ..., F:15)
- 2^4 is 16, so four bits needed for one hex digit, divide in groups of four (pad with zeros)

16 is in binary 10000 and padded: 0001 0000

10_{16} ($0 \times 16^0 + 1 \times 16^1$)

32 is 100 000 in binary, padded: 0010 000

20_{16} ($0 \times 16^0 + 2 \times 16^1$)

What is 65 in hex?

Hexadecimal (base 16) Suggested solution

- Base 16, digits 0-9A-F (A: 10, B:11, ..., F:15)
- 2^4 is 16, so four bits needed for one hex digit, divide in groups of four (pad with zeros)

16 is in binary 10000 and padded: 0001 0000

10_{16} ($0 \times 16^0 + 1 \times 16^1$)

32 is 100 000 in binary, padded: 0010 000

20_{16} ($0 \times 16^0 + 2 \times 16^1$)

What is 65 in hex?

65 is in binary 0100 0001, which becomes 41_{16} .

$1 \times 16^0 + 4 \times 16^1$.