CSCE 312: Computer Organization

David Kebo Houngninou

Representing and Manipulating Data

Analog signal

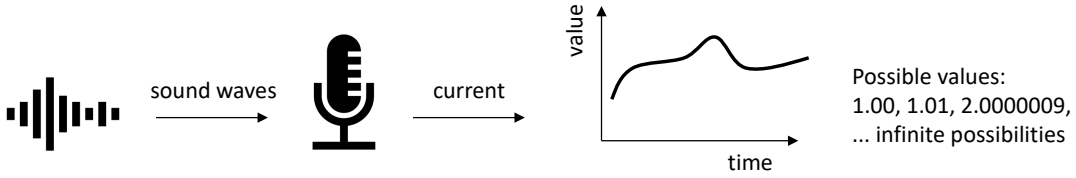
An analog signal is a continuous signal that changes over time with an infinite number of possible values.

Analog signals applications:

Radio

Audio recording

Video transmission (VGA, S-Video).



Digital signal

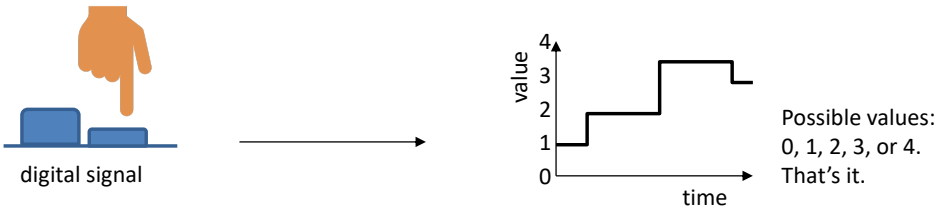
A digital signal is a signal that represents data as a sequence of discrete values. A digital signal has a finite set of possible values.

Digital signals application:

Video transmission (HDMI)

Audio transmission (MIDI).

Integrated circuits communication (Serial, I2C)



Analog vs. Digital

Comparison of analog signals and digital signals.

	Analog signal	Digital signal
Representation	Sine wave.	Square wave
Description	Amplitude, period or frequency, and phase.	Bit rate and bit intervals.
Range	No fixed range.	Finite numbers (0 and 1)
Distortion	More prone to distortion.	Less prone to distortion.
Transmit	Transmit data in the form of a wave.	Transmit data in the binary (0, 1)

Why use Digital over Analog?

Analog signals (e.g., audio) may lose quality if voltage levels not transmitted perfectly.

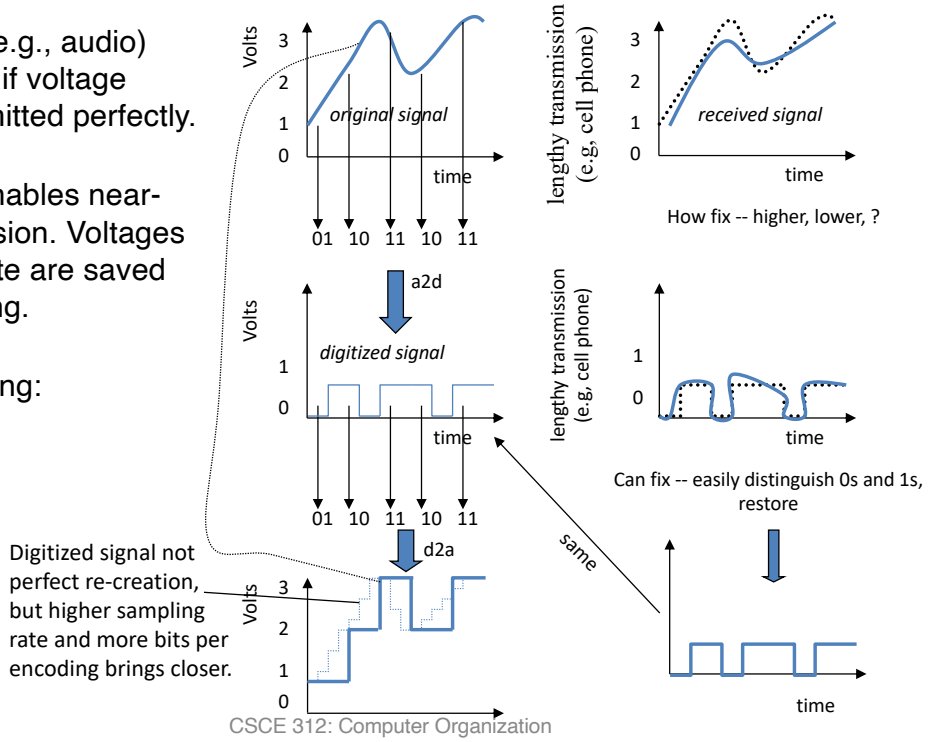
Digital signals enables near-perfect transmission. Voltages at a particular rate are saved using bit encoding.

e.g. of bit encoding:

1 V: “01”

2 V: “10”

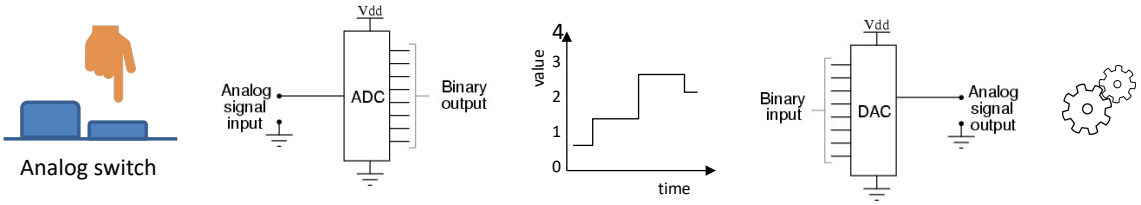
3 V: “11”



Data encoding

The physical world is analog.

For a computer to interact with the physical world, it needs to convert analog signals to digital signal then back to analog.



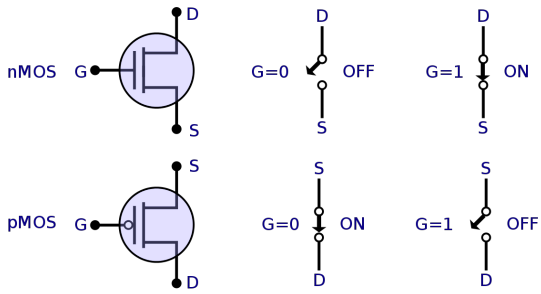
Digital signal in binary

A binary number is a number expressed in the base-2 numeral system with two symbols: 0 and 1. One binary digit is called a bit.

Binary is popular in computers because transistors operate using two voltages.

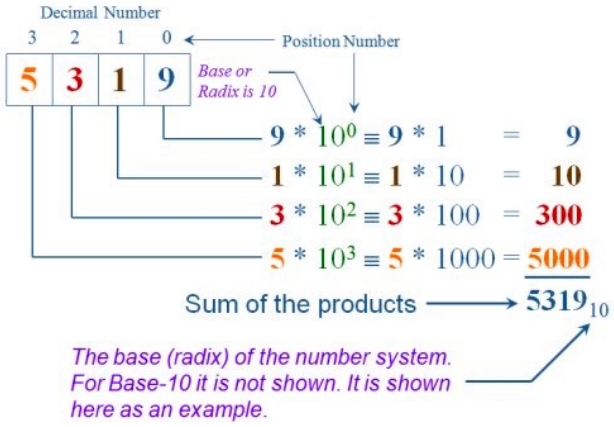
Transistors are switches with three terminals: gate, drain and source.

When the gate terminal is powered, current flows from the source to the drain.



The base of a number system

A radix, or base, is the number of unique digits, including zero, used to represent numbers in a positional numeral system.



The base of a number system

Generalized form of positional systems in Base B:

Commonly used numeral systems include:

Binary (Base 2): 0, 1

Octal (Base 8): 0, 1, 2, 3, 4, 5, 6, 7

Decimal (Base 10): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Hexadecimal (Base 16): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Digital signal in binary

Computers use Boolean logic for all computations.

We use this format to electrically represent 0 and 1 for different levels of voltage, e.g. 0V and +5V.

Conversion of a number from decimal to binary:

To convert a decimal number to binary:

1. First, subtract the largest possible power of two
2. Keep subtracting the next largest possible power of 2 from the remainder, marking 1s in each place where this is possible and 0s where it is not.

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 to binary conversion

Converting the decimal (base 10) number 86 to binary (base 2)

64 is the largest power of 2 that goes into 86. The result is:

1 ? ? ? ? ? ?
64 32 16 8 4 2 1

86 - 64 is 22. 32 is larger than 22, so a 0 is placed in the bit for the value 32. 16 is less than 22, so a 1 is placed in the bit for the value 16.

1 0 1 ? ? ? ?
64 32 16 8 4 2 1

22 - 16 is 6. 8 is larger than 6.
The bits 4+2 equal 6 so each of those bits become 1

1 0 1 0 1 1 0
64 32 16 8 4 2 1

This 86 in decimal is 1010110 in binary. $86_{10} = 1010110_2$

Binary to decimal conversion

Conversion of a number from binary to decimal:

$N_2 =$ 1 0 0 0 0 1 1 0 0 1
9 8 7 6 5 4 3 2 1 0

The value of N_2 in the Decimal Base N_{10} is:

$$N_{10} = 1 \times 2^9 + 0 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 537$$

Hexadecimal

Binary number can be long and hard to read, so hexadecimal numbers were introduced.

Hexadecimal (HEX) combines 4 bits into a single digit, written in base 16.

Hexadecimal is more compact and more readable. It uses the symbols A, B, C, D, E, F for the numbers 10, 11, 12, 13, 14, and 15, respectively.

Decimal	Binary	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	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
16	1 0000	10
17	1 0001	11

Activity 2.1

Perform the following number conversions:

A. 0x39A7F8 to binary

B. Binary 1100100101111011 to hexadecimal

C. 0xD5E4C to binary

D. Binary 1001101110011110110101 to hexadecimal

Activity 2.1 solution

Perform the following number conversions:

A. 0x39A7F8 to binary

0011 1001 1010 0111 1111 1000

B. Binary 1100100101111011 to hexadecimal

1100 1001 0111 1011 = C 9 7 B

C. 0xD5E4C to binary

1101 0101 1110 0100 1100

D. Binary 1001101110011110110101 to hexadecimal

0010 0110 1110 0111 1011 0101 = 2 6 E 7 B 5

Activity 2.2

Fill in the missing entries in the following table:

#	Decimal	Binary	Hexadecimal
1	0	0000 0000	0x00
2	**167**		
3	62		
4	188		
5		0011 0111	
6		1000 1000	
7		1111 0011	
8			0x52
9			0xAC
10			0xE7

Activity 2.2 solution

Fill in the missing entries in the following table:

#	Decimal	Binary	Hexadecimal
1	0	0000 0000	0x00
2	**167 = 10*16+7**	1010 0111	0xA7
3	62 = 3*16 + 14	0011 1110	0x3E
4	188 = 11*16 + 12	1011 1100	0xBC
5	3*16+7 = 55	0011 0111	0x37
6	8*16+8 = 136	1000 1000	0x88
7	15*16+3 = 243	1111 0011	0xF3
8	5*16+2 = 82	0101 0010	0x52
9	10*16+12 = 172	1010 1100	0xAC
10	14*16+7 = 231	1110 0111	0xE7

ASCII

ASCII:

American Standard Code for Information Interchange, is a character encoding standard for electronic communication.

Every ASCII character is 1 Byte (8 bits).

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Unsigned integers

An unsigned integer containing n bits can have a value between 0 and $2^n - 1$

Memory addresses are always represented by unsigned integers

e.g. the binary number 11001

11001 = 16 + 8 + 0 + 0 + 1 = 25

Signed integers

Computers can represent negative values, using the high-order bit to indicate the sign of a value.

The high-order bit or most significant bit is the leftmost bit in a binary number.

The remaining bits contain the value of the number.

Signed binary numbers can be expressed as:

- Signed magnitude
- One's complement
- Two's complement

Signed magnitude

In an 8-bit number, the signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit.

4 = 0 0000100

-4 = 1 0000100

e.g. Sum of 74 and 46

1. Convert 74 and 46 to binary
2. Arrange as a sum but separate the sign bits from the magnitude bits

```
    0  1001010
+
    0  0101110
=   ———————————
    0  1111000   =  120
```

Signed magnitude

What if the sum of the two values does not fit into seven bits?

e.g. calculate the sum of 102 and 46

The carry from the seventh bit overflows.

```
    0 1100110
+
    0 0101110
=   ---------
   010010100  =  148
```


One's complement

To express a number in one's complement: invert all the bits in the binary representation of the number.

e.g.: An 8-bit binary number using one's complement:

4 = 0 0000100

-4 = 1 1111011

In one's complement, as with signed magnitude, negative values are indicated by a 1 in the high order bit.

Complement systems are useful because they eliminate the need for subtraction.

One's complement

With one's complement addition, the carry bit is carried around and added to the sum.

e.g. Sum of 48 and -19

48 = 00110000

19 = 00010011

-19 = 11101100

```
    00110000
+
    11101100
    --------
=   00011100
+           1
    --------
    00011101
```


Two's complement

To express a value in two's complement:

If the number is positive: convert it to binary.

If the number is negative: invert all the bits in the binary number (one's complement) and add one to it.

e.g.:

4 = 0 0000100

-4 = 1 1111100

Two's complement

e.g.: Calculate the sum of 48 and -19

19 in binary is 00010011

-19 in one's complement is 11101100

-19 in two's complement is 11101101

Note: for the sum, discard the carry emitting from the high order bit!

```
  00110000
+ 11101101
----------
= 1 00011101 = 29
```


Overflow

In a computing system, resources are finite.

There is always the risk that the result of a calculation becomes too large to store.

An overflow cannot always be prevented but can be detected!

Using two's complement binary arithmetic, the sum of 104 and 46 is

```
  01101000
+ 00101110
----------
= 10010110   = -106
```

The nonzero carry from the seventh bit overflows into the sign bit, resulting in an erroneous value! 104 + 46 = -106

Overflow

Good programmers stay alert for it!

- Rule for detecting signed two's complement overflow:

  Carry in and carry out of the sign bit are different

- Rule for detecting unsigned number overflow:

  There is carry out of the leftmost bit

  1111 + 1 = 0000

Representation ranges

3 bits
Signed: -3, 3
1's: -3, 3
2's: -4, 3

5 bits
Signed: -15, 15
1's: -15, 15
2's: -16, 15

6 bits
Signed: -31, 31
1's: -31, 31
2's: -32, 31

8 bits
Signed: -127, 127
1's: -127, 127
2's: -128, 127

Formula for calculating the range for n bits

Signed: $-(2^{n-1} - 1), (2^{n-1} - 1)$
1's: $-(2^{n-1} - 1), (2^{n-1} - 1)$
2's: $-2^{n-1}, (2^{n-1} - 1)$

Data sizes

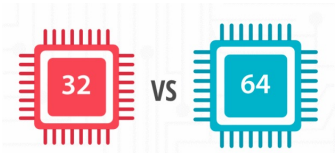
Every computer has a word size, indicating the nominal size of integer and pointer data.

A virtual address is encoded by the word.

The word size determines the maximum size of the virtual address space.

An n-bit machine, has a range of $2^n - 1$ virtual addresses.

e.g. A 32-bit word limits the virtual address space to 4 Gigabytes (4GB)



Data Organization in Memory

Memory contains locations that store fixed size data.

Each location is provided with a unique address.

Depending on the data path/size of the processor.

The memory content is accessible in sizes of:

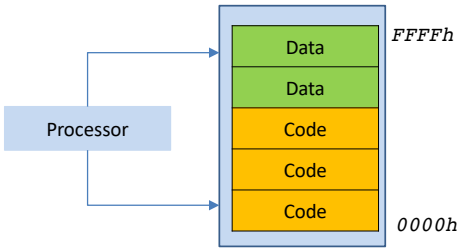
- Byte: 8-bit
- Half word: 16-bit
- Word: 32-bit
- Double word: 64-bit

Address Space

The address space is the range of addresses that can be accessed by the processor.

Some processor families (e.g. ARM) utilize only one address space for both memory and I/O devices

i.e. everything is mapped in the same address space



Data Alignment

32-bit data consists of four bytes of data, and is stored in four successive memory locations.

Data and code must be aligned to the respective address size boundary.

e.g. for a 32-bit system, align to the word boundary, with the lowest two address bits equal to zero

But what is the order of the four bytes of data?

It depends on the Endianness of the processor!

Data Endianness

Little Endian format:

The least significant byte (LSB) is stored in the lowest address of the memory.

The most significant byte (MSB) stored in the highest address location of the memory.

Big Endian format:

The least significant byte (LSB) is stored in the highest address of the memory.

The most significant byte (MSB) is stored in the lowest address location of the memory.

Data Endianess

Little Endian: x86, ARM processors running Android, iOS, and Windows

Least significant byte has lowest address

Big Endian: Sun, PPC Mac, Internet

Least significant byte has highest address

Storing data in memory

```
STR r3, [r8]   ;Store r3 to the address r8
```

r3 content

0xE011CFD0

r8 content

0x00008000

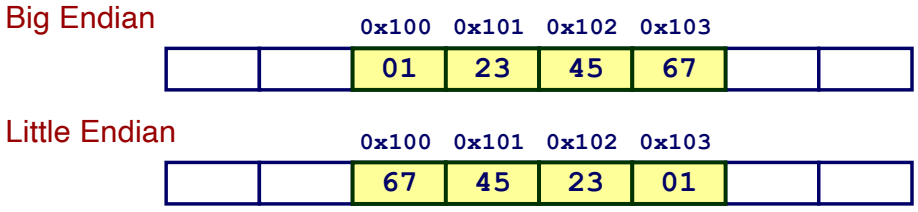
Memory after Store

Address	Data
0x8000	0xD0
0x8001	0xCF
0x8002	0x11
0x8003	0xE0
0x8004	0x00
0x8005	0x00
0x8006	0x00

Storing data in memory

Example

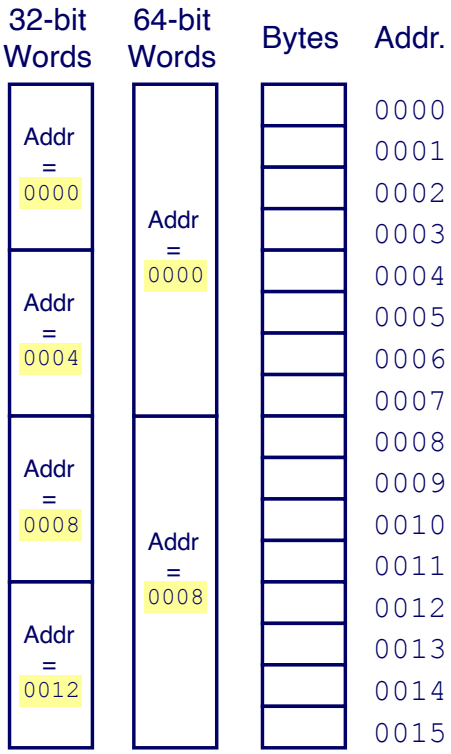
- Variable x has 4-byte value of 0x01234567
- Address given by &x is 0x100



Word-Oriented Memory Organization

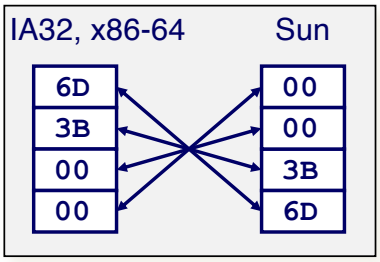
Addresses Specify Byte Locations

- Address of first byte in word
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)



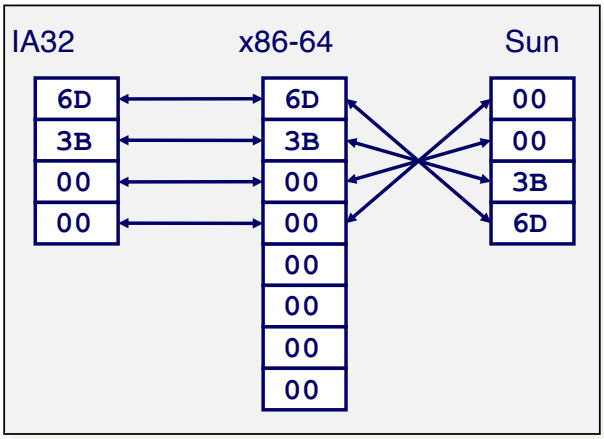
Representing Integers

```
int A = 15213;
```

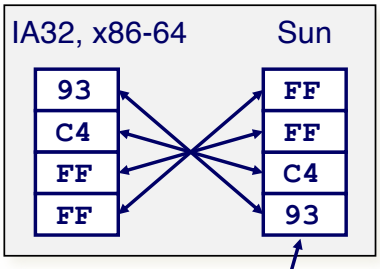


Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

```
long int C = 15213;
```




```
int B = -15213;
```



Two's complement representation

Byte representations

Linux 32: Intel IA32 core running Linux.

Windows: Intel IA32 core running Windows.

Sun: Sun Microsystems SPARC core running Solaris.

Linux 64: Intel x86-64 core running Linux.

Machine	Value	Type	Bytes (hex)
Linux 32	12,345	int	39 30 00 00
Windows	12,345	int	39 30 00 00
Sun	12,345	int	00 00 30 39
Linux 64	12,345	int	39 30 00 00
Linux 32	12,345.0	float	00 e4 40 46
Windows	12,345.0	float	00 e4 40 46
Sun	12,345.0	float	46 40 e4 00
Linux 64	12,345.0	float	00 e4 40 46
Linux 32	&ival	int *	e4 f9 ff bf
Windows	&ival	int *	b4 cc 22 00
Sun	&ival	int *	ef ff fa 0c
Linux 64	&ival	int *	b8 11 e5 ff ff 7f 00 00

Pointer values are machine dependent.

The byte representations of different data values. Results for int and float are identical, except for byte ordering.

Data sizes

Most 64-bit computers can run compiled program compiled for 32-bit computers (backward compatibility)

```
linux > gcc –m32 program.c
```

This program can run on either 32-bit or 64-bit machine.

However a program compiled with the directive:

```
linux > gcc –m64 program.c
```

will only run on a 64 bit computer.

Computers and compilers support multiple data formats and encode data in different formats e.g. int, float.

Data sizes and pointers

Pointers in C provide the mechanism for referencing elements of data structures, including arrays.

Just like a variable, a pointer has a value and a type.

1. The value indicates the location of the object.
2. The type indicates the kind of object (integer, floating-point etc.) that is stored at that location.

e.g.

```
T *p; // p is a pointer variable, pointing to an object of type T.
```

```
char *k; // k is a pointer variable, pointing to an object of type char.
```


Data sizes

The C language supports multiple data formats for both integer and floating-point data.

The C data type char is a single byte.

The type char stores a single character in a text string, and can also store integer values.

The C data type int can also be prefixed by the qualifiers short, long, and recently long long, providing integer representations of various sizes.

C Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Examining Data Representations

Code to Print Byte Representation of Data

Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

`show_bytes` Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Result (Linux x86-64):

```
int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

Naming data types with typedef

The typedef declaration in C gives a name to a data type.

The syntax for typedef is like that of declaring a variable, except that it uses a type name rather than a variable name.

e.g.

`typedef int *int_pointer;` // define type int_pointer to be a pointer to an int

`int_pointer ip;` // declare a variable ip of this type.

Alternatively, we could declare this variable directly as:

`int *ip;`

Formatting data types with printf

Functions printf, fprintf and sprint provide a way to print data with a format.

The first argument is a format string, and the remaining arguments are values to print.

Each character sequence starting with '%' indicates how to format the next argument.

%c	character
%d	decimal
%e	exponential floating-point
%f	floating-point number
%i	integer
%o	octal number
%s	string of characters
%u	unsigned decimal number
%x	hexadecimal
%%	print a percent sign
\%	print a percent sign

Pointers and arrays

In show_bytes() we observe a relation between pointers and arrays.

The function has an argument start of type byte_pointer (defined to be a pointer to unsigned char), but we see the array reference start[i]

In C, we can dereference a pointer with the array notation, and we can reference array elements with pointer notation.

In the code example, the reference start[i] indicates that we want to read the byte that is i positions beyond the location pointed to by start.

Read more about pointers: https://boredzo.org/pointers/