UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Bits

COMPUTER ARCHITECTURE AND OPERATING SYSTEMS

Bioinformatics

2025/26 Spring Term

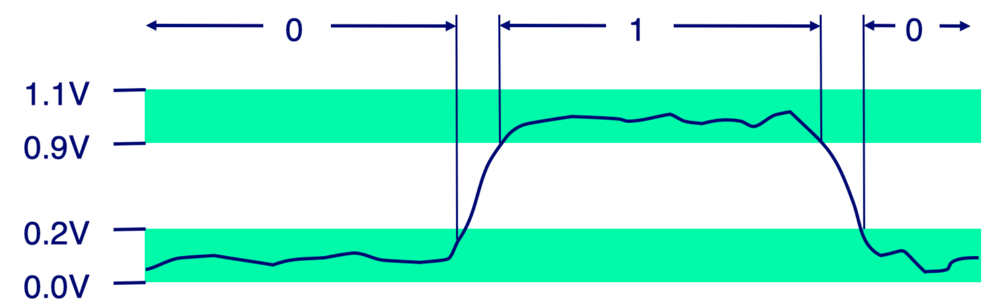
Jordi Fornés

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Departament d'Arquitectura de Computadors

Contents

Contents

Numeral systems



- Humans are acostumed to decimal (base 10) arithmetic
- A computer performs only binary (bas 2) arithmetic
- N-bit registers impose limitations on the size of fields and require special treatment for large values
- Hexadecimal (base 16) allows a compact notation and a straight forward conversion from/to binary.

Base conversion

dec	bin	hex
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

Base conversion

To convert a base 10 (decimal) number to base 16 (hexadecimal):

314156 = 16 * 19634 + **12**	C
19634 = 16 * 1227 + **2**	2
1227 = 16 * 76 + **11**	B
76 = 16 * 4 + **12**	C
4 = 16 * 0 + **4**	4

Read the remainders from bottom to top:

$$31415610_{10} = 4CB2C_{16}$$

Base conversion

To convert a base 10 (decimal) number to base 2 (binary):

$$\begin{aligned} 174 &= 2 \cdot 87 + \mathbf{0} \\ 87 &= 2 \cdot 43 + \mathbf{1} \\ 43 &= 2 \cdot 21 + \mathbf{1} \\ 21 &= 2 \cdot 10 + \mathbf{1} \\ 10 &= 2 \cdot 5 + \mathbf{0} \\ 5 &= 2 \cdot 2 + \mathbf{1} \\ 2 &= 2 \cdot 1 + \mathbf{0} \\ 1 &= 2 \cdot 0 + \mathbf{1} \end{aligned}$$

Read the remainders from bottom to top:

$$174_{10} = 10101110_2$$

Base conversion

- To convert from base b a number b,with n digits, of the form $y_{n-1}y_{n-2}\ldots y_1y_0$ to base 10 (decimal):

$$y = \sum_{i=0}^{n-1} y_i \cdot b^i$$

- Examples:

$$01001010_2 = 0 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 74_{10}$$

$$3E_{16} = 3 \cdot 16^1 + 14 \cdot 16^0 = 62$$

Numeral Systems in Python

- Constants starting with `0b` or `0B` are interpreted being in binary

```
>>> bin(15)
‘0b1111’
>>> d=0b101
>>> d
5
```

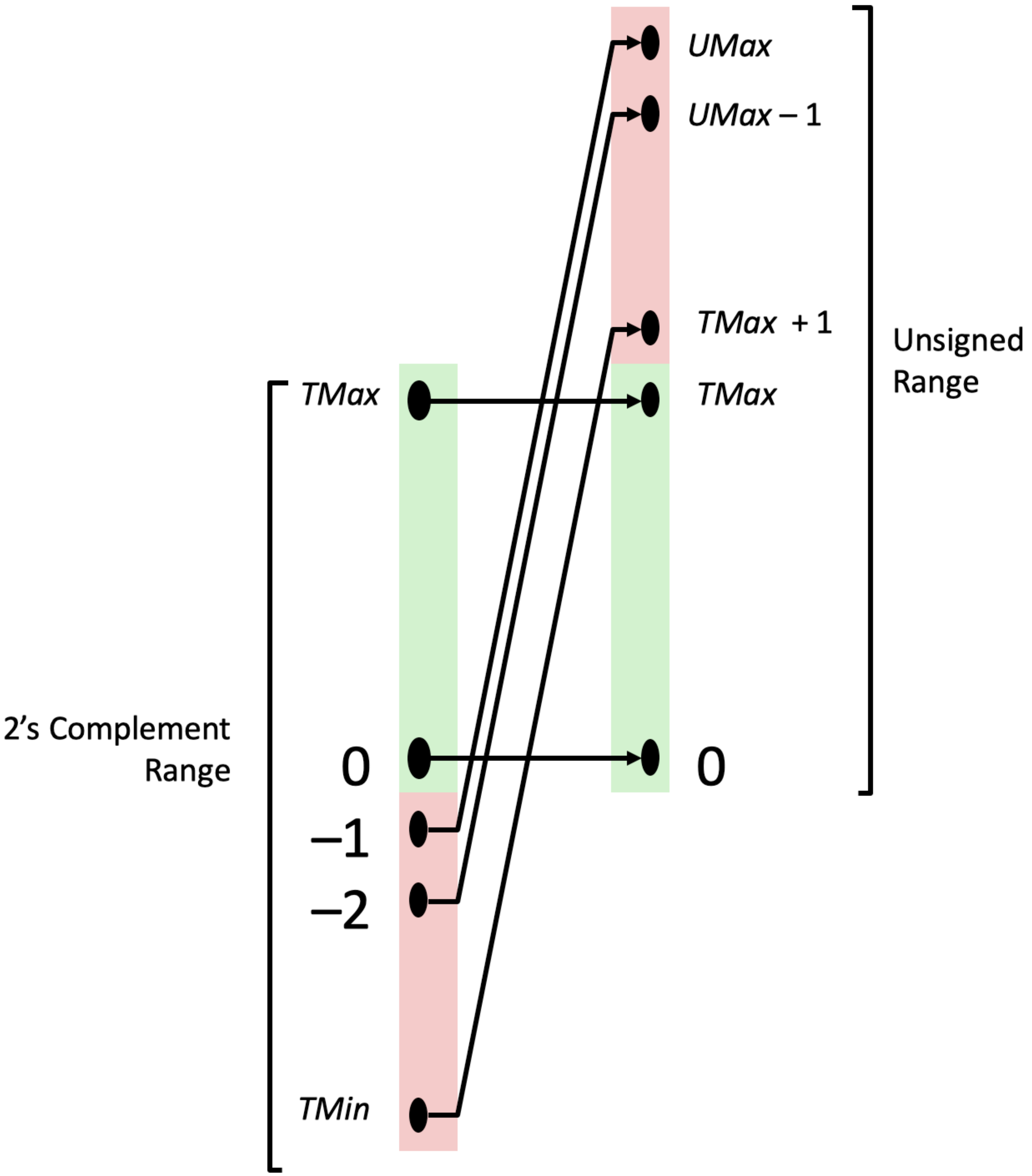
- Constants starting with `0x` or `0X` are interpreted being in hexadecimal

```
>>> hex(44252)
‘0xacdc’
```

Contents

Signed vs unsigned

Signed	Bits	Unsigned
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
-7	1001	9
-6	1010	10
-5	1011	11
-4	1100	12
-3	1101	13
-2	1110	14
-1	1111	15



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Usigned vs signed

- Unsigned: ordinary binary representation

$$y = \sum_{i=0}^{n-1} y_i \cdot base^i$$

  - Range from 0 to $2^n - 1$

- Signed: two's complement

$$y = -y_{n-1} \cdot 2^{n-1} \sum_{i=0}^{n-2} y_i \cdot 2^i$$

  - Range from -2^{n-1} to $2^{n-1} - 1$

Hexa, binary, unsigned, two's complement

Hexa	Binary	Unsigned	Signed
A	1010	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
1	0001	$2^0 = 1$	$2^0 = 1$
B	1011	$2^3 + 2^1 + 2^0 = 11$	$-2^3 + 2^1 + 2^0 = -5$
7	0111	$2^2 + 2^1 + 2^0 = 7$	$2^2 + 2^1 + 2^0 = 7$
C	1100	$2^3 + 2^2 = 12$	$-2^3 + 2^2 = -4$

Two's complement

- From positive to negative
  - Reverse the bits and add 1, ignore the overflow

$$5_{10} = 0101_2; -5_{10} = 1010_2 + 1_2 = 1011_2$$

$$2_{10} = 0010_2; -2_{10} = 1101_2 + 1_2 = 1110_2$$

- One special case:
  - The two's complement of the most negative number representable is itself
  - Example: n = 4 bits, the most negative number representable is -2^3

$$8_{10} = 1000_2; -8_{10} = 0111_2 + 1_2 = 1000_2$$

Signed vs unsigned in Python

- `numbers.Integral` represent numbers in an unlimited range, subject to available (virtual) memory only.
- Negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

```
>>> a=-1
```

$111\ldots111_2$

- for the purpose of shift and mask operations, a binary representation is assumed.

```
>>> a=~1
0
```

Python `numbers.Integral`

- Quoted from docs.python.org:
- Booleans (`bool`)
  - These represent the truth values `False` and `True`.
  - The two objects representing the values `False` and `True` are the only `Boolean` objects.
  - The `Boolean` type is a subtype of the `integer` type, and `Boolean` values behave like the values 0 and 1, respectively, in almost all contexts.
  - The exception being that when converted to a string, the strings "False" or "True" are returned, respectively.

```
>>> type(True)
<type 'bool'>
>>> isinstance(True, int)
True
```

Contents

Real Numbers

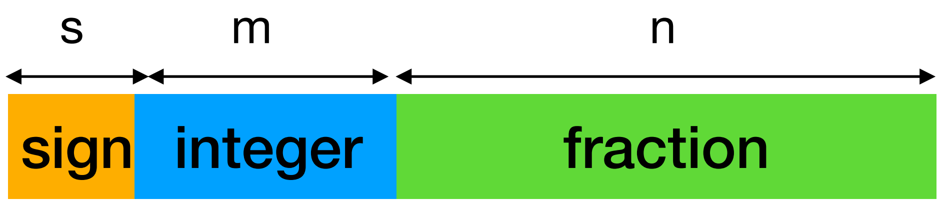
Numbers with a fractional component

- Two main representations:
  - Fixed-point vs Floating-point
  - The radix point is fixed or can float anywhere
    - The symbol to separate integer and fractional parts of a real
- Implementation: tradeoff between cost and precision
  - Lack of hardware resources → e.g. Multimedia decoders
  - Boost performance although degraded precision → e.g. Playstations, Doom

Real Numbers

Fixed-point numbers

- Bits = 1 + m + n
  - 1 bit for sign (if signed)
  - m bits for integer component
  - n bits for fraction component
- Notation: $Q_{m.n}$

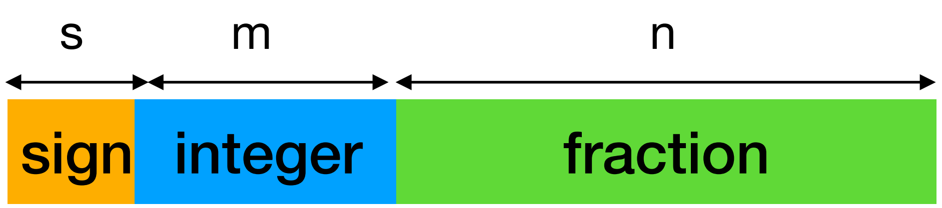


  - Integer number without fraction component $Q_{m.0}$
  - Fractional number without integer component Q_n

Fixed-point Numbers

$$Value = -2^m b'_s + 2^{m-1} b'_{m-1} + \ldots + 2^1 b'_1 + 2^0 b'_0 + 2^{-1} b_{n-1} + 2^{-2} b_{n-2} + \ldots + 2^{-n} b_0$$

- Programming language support
  - C and C++ have no direct suport, but can be implemented
    - Embedded-C supports it (implemented in GCC)
- Python has direct suport via <u>decimal</u> module
- Examples:



  - $Q_{3.0} : -2^3 + 2^2 + 2^1 = -2$
  - $Q_{1.2} : -2^1 + 2^0 + 2^{-1} = -2 + 1 + 0.5 = 0.5$
  - $Q_3 : -2^0 + 2^{-1} + 2^{-2} = -1 + 0.5 + 0.25 = -0.25$

Fixed-point Numbers

Accuracy problems

- Precision loss and overflow
- Results can require more bits than operands
  - Round or truncate
  - Specify different size for result
- Boundary numbers to prevent overflow
  - Exception: overflow flag, if supported by hardware

Floating-point Numbers

- Bits = 1+ e + k
- **1** bit for sign (if signed)
- **e** bits for exponent: $1, \ldots, (2^e - 1) - 1$
- **k** bits for mantissa (fraction)
  - There is an implicit 1-bit (top left) equals to 1, unless exponent is equal to zero
- Most processors follow IEEE floating point standard
  - First version on 1985
  - Standardize formats
  - Special Values

Floating point reference for Intel Architecture

https://www.intel.com/content/www/us/en/content-details/786447/floating-point-reference-sheet-for-intel-architecture.html

Binary Format Floating-Point Number

Sign	Biased Exponent	Significand					
s	E	x_1	x_2	x_3	…	x_{p-1}	x_p
MSB		J-bit	Fraction				*LSB*

$$= \begin{cases} (-1)^s \times x_1.x_2x_3\cdots x_{p-1}x_p \times 2^{E-B}, & \text{if normal} \\ (-1)^s \times x_1.x_2x_3\cdots x_{p-1}x_p \times 2^{e_{min}}, & \text{if denormal} \end{cases}$$

- Sign bit is s = 0 for '+', and s = 1 for '–' (also refer to 's' as 'sign')
- Unbiased exponent is $e = E - B - x_1 + 1$ for nonzero finite numbers
- For standard formats, x_1 equals ($E \neq 0$) and is implicit
- For NaNs, the payload is the bit string from x_3 to x_p

Floating-Point Classes, Encodings, and Parameters

*Standard Formats** — *Extended Format** — *Non-Std**

Class	E	J	Fraction	Values	Half (16b)	Single (32b)	Double (64b)	Quad (128b)	x87 (80b) t***	Bfloat (16b)
Zero	00…00	0	00…00	+Zero	0000	0000 0000	0000 0000 0000 0000	0000 0000 … 0000	0000 **0**000 … 0000	0000
Denormal	00…00	0	00…01 ↔ 11…11	$+D_{min}$	0001	0000 0001	0000 0000 0000 0001	0000 0000 … 0001	0000 **0**000 … 0001	0001
				$+D_{max}$	03ff	007f ffff	000f ffff ffff ffff	0000 ffff … ffff	000**0** **7**fff … ffff	007f
Normal	00…01 ↔ 11…10	1	00…00 ↔ 11…11	$+N_{min}$	0400	0080 0000	0010 0000 0000 0000	0001 0000 … 0000	000**1** **8**000 … 0000	0080
				+One	3c00	3f80 0000	3ff0 0000 0000 0000	3fff 0000 … 0000	3fff **8**000 … 0000	3f80
				$+N_{max}$	7bff	7f7f ffff	7fef ffff ffff ffff	7ffe ffff … ffff	7ff**e** **f**fff … ffff	7f7f
Infinity	11…11	1	00…00	+Infinity	7c00	7f80 0000	7ff0 0000 0000 0000	7fff 0000 … 0000	7ff**f** **8**000 … 0000	7f80
sNaN	11…11	1	00…01 ↔ 01…11	"+"sNaN	7c01	7f80 0001	7ff0 0000 0000 0001	7fff 0000 … 0001	7fff **8**000 … 0001	7f81
					7dff	7fbf ffff	7ff7 ffff ffff ffff	7fff 7fff … ffff	7fff **b**fff … ffff	7fbf
qNaN	11…11	1	10…00	R Ind**	fe00	ffc0 0000	fff8 0000 0000 0000	ffff 8000 … 0000	ffff **c**000 … 0000	ffc0
			10…00 ↔ 11…11	"+"qNaN	7e00	7fc0 0000	7ff8 0000 0000 0000	7fff 8000 … 0000	7fff **c**000 … 0000	7fc0
					7fff	7fff ffff	7fff ffff ffff ffff	7fff ffff … ffff	7fff **f**fff … ffff	7fff

	Half (16b)	Single (32b)	Double (64b)	Quad (128b)	x87 (80b)	Bfloat (16b)
Field	s / E / J / F	s / E / J / F	s / E / J / F	s / E / J / F	s / E / J / F	s / E / J / F
# of Bits	1 / 5 / 0 / 10	1 / 8 / 0 / 23	1 / 11 / 0 / 52	1 / 15 / 0 / 112	1 / 15 / 1 / 63	1 / 8 / 0 / 7
Exp. bias (B)	0x0f (15)	0x7f (127)	0x3ff (1023)	0x3fff (16383)	0x3fff (16383)	0x7f (127)
$e_{min} : e_{max}$	−14 : 15	-126 : 127	-1022 : 1023	−16382 : 16383	−16382 : 16383	-126 : 127

* All examples are in little endian byte order ** R Ind (Real Indefinite), a qNaN, must have sign bit s = 1 and payload = 00…00

*** Two additional classes exist for x87 80-bit format: pseudo-denormal (E = 0, J = 1) and unsupported (E ≠ 0, J = 0)

Floating-point numbers

$$Value = (-1)^{sign} \cdot (1 + \sum_{1}^{k} b_{(k-i)} 2^{-i}) \cdot 2^{E}$$

$$E = Exponent - E_{max}$$

- Example:

$$23.46875_{10} = 23_{10} + 0.46875_{10} =$$

$$10111_2 + 0.01111_2 = 10111.01111_2 = 1.011101111_2 \cdot 2^4$$

32-bit floating point representation:

Sign (1b) = 0, $E = 4_{10}$

Exponent (8b) = $4_{10} + E_{max} = 131_{10} = 10000011_2$

Mantissa (23b) = $0111011110\ldots0_2$

$$01000001101110111100000000000000 = 41BBC000_{hex}$$

Floating-point numbers

Support

- Float in Python:
    - `float` is usually implemented using `double` in C (64b)
    - Check precission and implementation in `sys.float_info`
    - `decimal.Decimal` for floating-point numbers with user-definable precision
- Most 32-bit architectures comprises 64-bit support in FPU (floating-point unit)
    - IA-32 and x86-64 present 80-bit floating-point type (double-extended precision format)
- Quad-precision (128-bit)
    - Software support
    - Few architectures provide hardware support
    - E.g. IBM POWER9 processors (MareNostrum 4)

Floating-point numbers

Accuracy problems

- Numbers that cannot be exactly represented as binary fractions
- E.g. 10^{-1}

  $0.0001100110011001100110011001100110011001100110011001\ldots$
- Conversion to integer loses acuracy due to truncate and roundoff
  - E.g. $56/7 = 8; 0.56/0.07 =$ could be 7
  - E.g. explosion of Ariane5 rocket(1996)
- Conmutative, but not necessary associative and distributive
  - $(a + b) + c$ could be not equal to $a + (b + c)$
  - $(a + b) \cdot c$ could be not equal to $a \cdot c + b \cdot c$

Symbols and characters

Scalar data type: symbols and characters

- Char data type: encode alphanumeric data and symbols
- Several character set encoding
- ASCII code (American Standard Code for Information Interchange)
  - Adopted by microcomputers
  - The standard for the early HTML
  - Single byte using the bottom 7 bits. From 0 to 127
  - The 128 values that represent the printable Latin A-Z (65-90), a-z (97-122), 0-9 (48-57)
  - Many common punctuation characters
  - Several non-displayable device control codes (0-31 and 127

Symbols and characters

Scalar data type: symbols and characters

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

Symbols and characters

Several character set encoding

- 128 symbols are not enough!
- What about the last bit (128-255)?
  - Windows-1252 code (CP-1252): also called ANSI
    - It is a superset of ISO-8859-1(more printable characters)
    - Used by default on legacy components of Microsoft Windows
    - ANSI comprises 8 bits: 1 additional bit compared to ASCII
  - ISO-8859-1 code (Latin Alphabet)
    - 1 full byte (256 characters): extension to ASCII
    - The standard from HTML2.0 to HTML4.01
    - The Latin-1 code-page defines many characters and symbols used by Latin-based languages
    - IBM used code-page 437 that define non-printable characters
    - Shells allow the user to change code-pages, which causes the terminal to display different characters
- Nevertheless, 255 symbols are not enough!
  - the solution: **UNICODE**

Symbols and characters

Handling Unicode

- Unicode or ISO/IEC 10646 is an international standard defining every character/glyph used in almost every writing system on Earth

- Unicode also defines several character encodings:

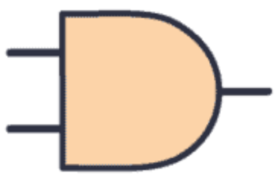
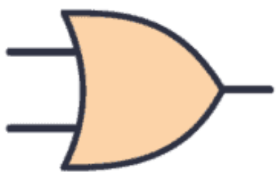
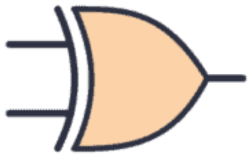
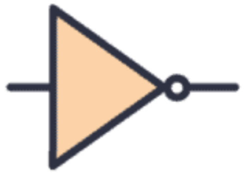
  - UTF-8: 1-byte for the first 127 code points (maintaining compatibility with ASCII), and an optional additional 1-3 bytes (4 bytes total) for other characters

  - UTF-16: 2-bytes for each character. UCS-2 (used internally by Windows) supports encoding the first 65536 code points (know as the Basic Multilingual Plane – BMP). UTF-16 extends UCS-2 by incorporating a 4-byte encoding for 17 additional planes of characters

  - UTF-32: 4-bytes per character

Contents

Logical operations

p q	p AND q	p OR q	p XOR q	NOT p
0 0	0	0	0	1
0 1	0	1	1	1
1 0	0	1	1	0
1 1	1	1	0	0
				

Basic operation (addition)

Operands: n bits

$$
\begin{array}{r}
p_{n-1}p_{n-2}\cdots p_1p_0 \\
+q_{n-1}q_{n-2}\cdots q_1q_0 \\
\hline
r_nr_{n-1}r_{n-2}\cdots r_1r_0
\end{array}
$$

- True sum: n+1 bits
- Standard addition ignores the carry bit output
- Implements modular arithmetic: $p + q \bmod 2^n$
- Example: $p = 5, q = 4, n = 3$

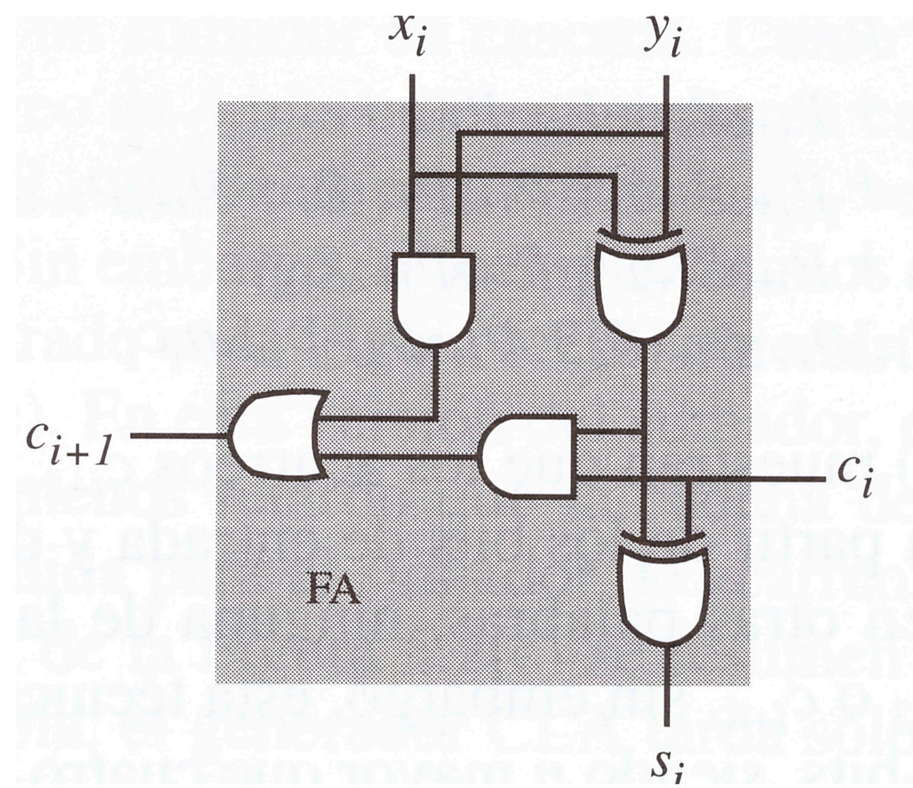
  (5+4) mod 8 = 1

  101 + 100 = ~~1~~ 001

Combinational circuits

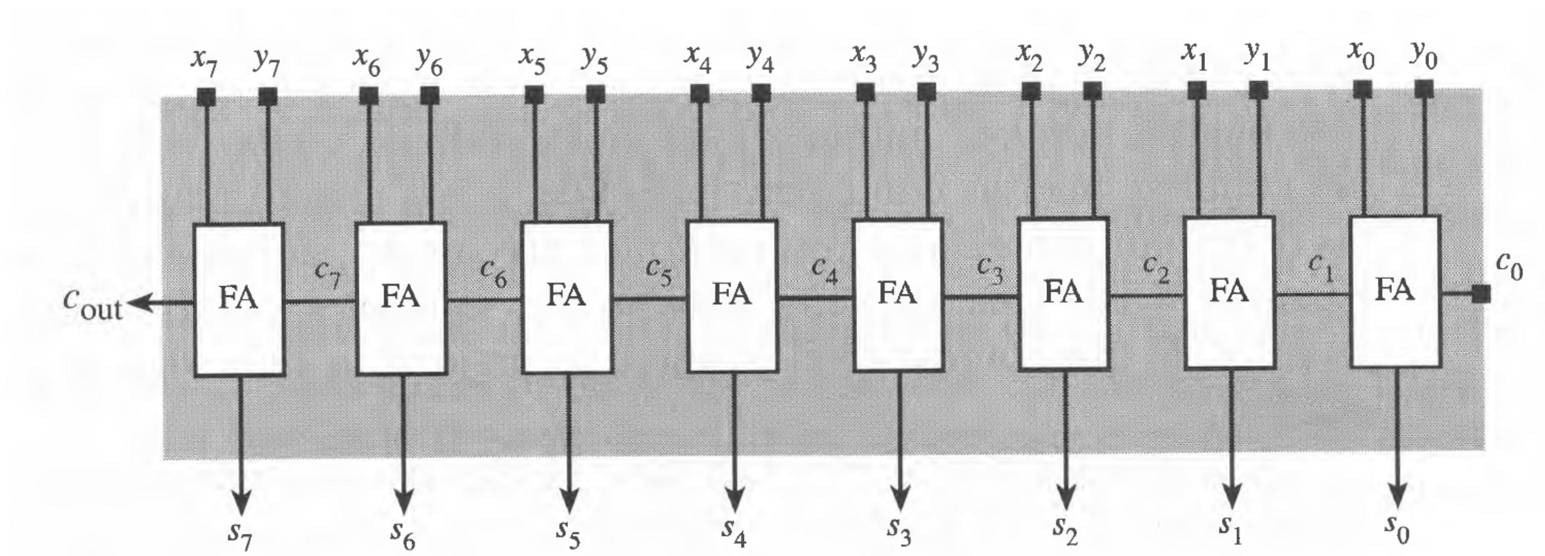
Xi	Yi	Ci	Ci+	Si
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Full adder's truth table



$$s_i = x_i \oplus y_i \oplus c_i$$

$$c_{i+1} = x_i y_i + c_i(x_i \oplus y_i)$$



Images source: Gajski, Daniel. *Principles of digital design*. Prentice-Hall, Inc. 1996.

Bit-level operations in Python

- x << y
- Returns x with the bits shifted to the left by y places, inserting zeroes on the right-hand-side
- $x = x \cdot 2^y$
- Example:

  >>> 4<<1 → 8

  $00000100_2 \rightarrow 000001000_2$

- Returns x with the bits shifted to the right by y places, inserting the sign bit on the left-hand-side.
- $x = \frac{x}{2^y}$
- Example:

  >>> 4>>1→ 2

  $00000100_2 \rightarrow 000000010_2$

Bit-level operations in Python

- x & y
  - Bitwise AND
  - Example:

    >>> 0x69 & 0x55 = 0x41

$$
\begin{array}{r}
01101001_2 \\
\& \quad 01010101_2 \\
\hline
01000001_2
\end{array}
$$

- x | y
  - Bitwise OR
  - Example:

  >>> 0x69 | 0x55 =0 0x7D

$$
\begin{array}{r}
01101001_2 \\
| \quad 01111101_2 \\
\hline
01111101_2
\end{array}
$$

Bit-level operations in Python

- ~x
    - The complement of x, ie, NOT x
    - Example:

    >>> ~0x41 = 0xBE

$$
\begin{array}{r}
\sim 01000001_2 \\
\hline
10111110_2
\end{array}
$$

- x ^ y
    - Bitwise exclusive OR
    - Example:

>>> $0x69 \wedge 0x55 = 0x3C$

$$
\begin{array}{rr}
 & 01101001_2 \\
\wedge & 01010101_2 \\
\hline
 & 00111100_2
\end{array}
$$

Logical operations in Python

- Three Boolean operators: `and`, `or`, and `not`
- `bool` type with two values: `True` and `False`
- Any non-zero number is interpreted as `True`
- Relational operators: `==`, `!=`, `>`, `<`, `>=`, `<=`
- Boolean expressions:

```
x = 5
if x>2:
 if x<10:
  print('Fits')
if x>2 and x < 10:
  print('Fits')
if 2<x<10:
  print('Fits')
else:
  print('no fits')
```

Contents

To sum up

- Memory is full of **bi**nary digi**ts**
- Hardware and software interpret bits
  - Following some kind of codification
  - Natural binary, two's complement, fixed point, floating point, unicode, …
- There are a lot of scalar data types
  - Integers, floats, chars, booleans
  - Bitmaps (enumeration), pointers, …
- And even more agreggated data types
  - Arrays, structures, classes, unions, strings, heaps, stacks, dictionaries, …

To sum up

```
$ od -X CAOS.txt
0000000  534f4143  4f41430a  41430a53  430a534f
0000020  0a534f41  534f4143  4f41430a  41430a53

$ od -F CAOS.txt
0000000     6,099819095891594e+73    9,262436571139304e+14
0000020     2,037357287262983e+93    2,495654619179134e+06

$ od -a CAOS.txt
0000000    C   A   O   S  nl   C   A   O   S  nl   C   A   O   S  nl   C
0000020    A   O   S  nl   C   A   O   S  nl   C   A   O   S  nl   C   A
```

The bibliography

- Randal E. Bryant, David R. O'Hallaron 2015. Chapter 2.
  - *Computer Systems. A programmers perspective*
- VanderPlas, Jake 2016. Chapter 3.
  - *Data Science Handbook*