AWK:

<https://learnxinyminutes.com/docs/awk/>

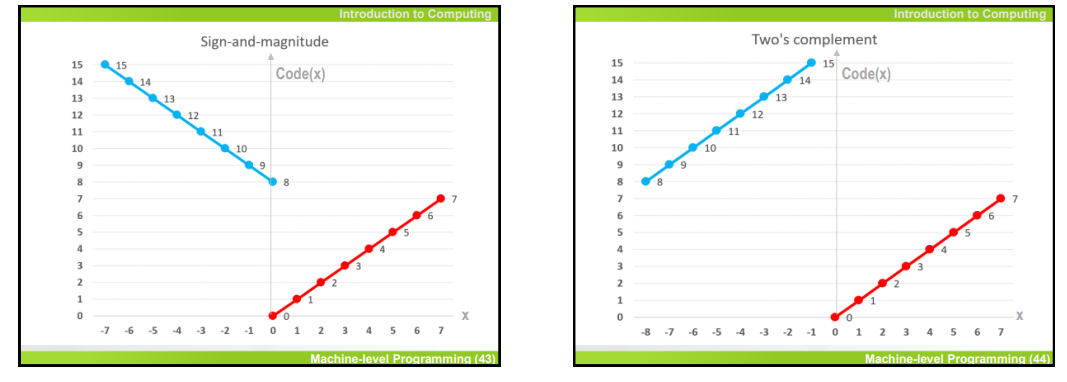
Assembler:

Instrukcje: <https://pl.wikibooks.org/wiki/Asembler_x86/Instrukcje>

Rejestry flag: <https://pl.wikibooks.org/wiki/Asembler_x86/Architektura>

Cheatsheetow nie ma w zdj gogle

Pomniejsze:



Kod uzupełnień do dwóch

Kod uzupełnień do dwóch (w skrócie U2 lub ZU2) – system reprezentacji liczb całkowitych w dwójkowym systemie pozycyjnym. Jest obecnie najpopularniejszym sposobem zapisu liczb całkowitych w systemach cyfrowych. Jego popularność wynika z faktu, że operacje dodawania i odejmowania są w nim wykonywane tak samo jak dla liczb binarnych bez znaku. Z tego też powodu oszczędza się na kodach rozkazów procesora.

Zaletą tego kodu jest również istnienie tylko jednego zera. Przedział kodowanych liczb nie jest przez to symetryczny.

Dla reprezentacji 8-bitowej (jednobajtowej) są to liczby od −128 do 127.

<-2^(n-1), 2^(n-1)-1>

­­

This representation is also called "sign–magnitude" or "sign and magnitude" representation. In this approach, a number's sign is represented with a sign bit: setting that bit (often the most significant bit) to 0 for a positive number or positive zero, and setting it to 1 for a negative number or negative zero. The remaining bits in the number indicate the magnitude (or absolute value). For example, in an eight-bit byte, only seven bits represent the magnitude, which can range from 0000000 (0) to 1111111 (127). Thus numbers ranging from −12710 to +12710 can be represented once the sign bit (the eighth bit) is added. For example, −4310 encoded in an eight-bit byte is 10101011 while 4310 is 00101011. Using signed magnitude representation has multiple consequences which makes them more intricate to implement:[5]

There are two ways to represent zero, 00000000 (0) and 10000000 (−0).

Addition and subtraction require different behavior depending on the sign bit, whereas one's complement can ignore the sign bit and just do an end-around carry, and two's complement can ignore the sign bit and depend on the overflow behavior.

Comparison also require inspecting the sign bit, whereas in two's complement, one can simply subtract the two numbers, and check if the outcome is positive or negative.

The minimum negative number is −127 instead of −128 in the case of two's complement.

Alternatively, a system known as ones' complement[6] can be used to represent negative numbers. The ones' complement form of a negative binary number is the bitwise NOT applied to it, i.e. the "complement" of its positive counterpart. Like sign-and-magnitude representation, ones' complement has two representations of 0: 00000000 (+0) and 11111111 (−0).

# **Metody Numeryczne**

Algorytm przeliczania ułamków dec na bin:

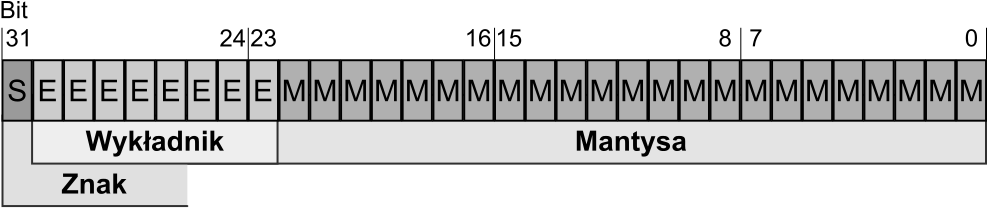
1. Zapisz 0,
2. Pomnóż ułamek przez 2, jeśli wynik jest większy bądź równy 1 należy dopisać 1 w reprezentacji binarnej. W przeciwnym wypadku należy dopisać 0.
3. Jeśli wynik jest równy 1 jest to koniec algorytmu. Jeśli wynik jest mniejszy od 1 należy przejść do punktu drugiego. Jeśli wynik jest większy od 1 należy odjąć od niego 1 i przejść do punktu drugiego.

Precyzja

single prec float 32 bit +-1.18\*10^(-38) do +-3.4\*10^38

double prec double 64 bit +-2.23 \* 10^(-308) do +- 1.8 \* 10^308

IEEE-754 (zapis ułamka w bin)



Liczba=(-1)^S\*m\*2^e

S-bit znaku

M-mantysa (w postaci znormalizowanej czyli 1,0101011010 bez pierwszej jedynki)

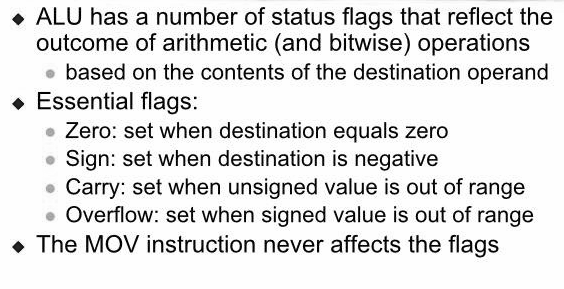
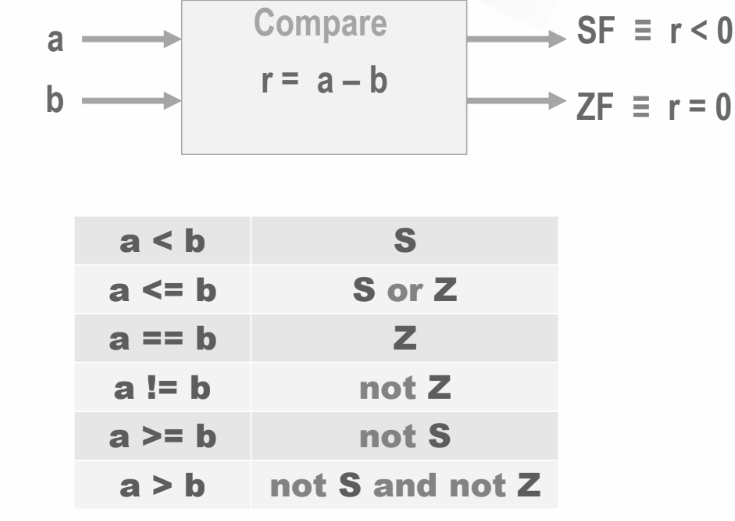
E-wykładnik (do faktycznego wykładnika ze wzoru dodajemy 127 żeby obsłużyć ujemne)

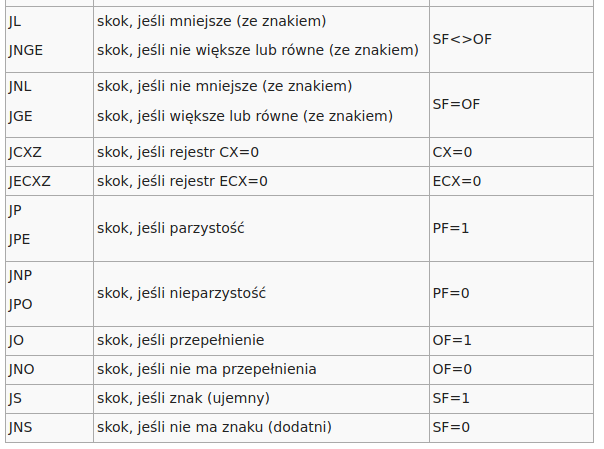
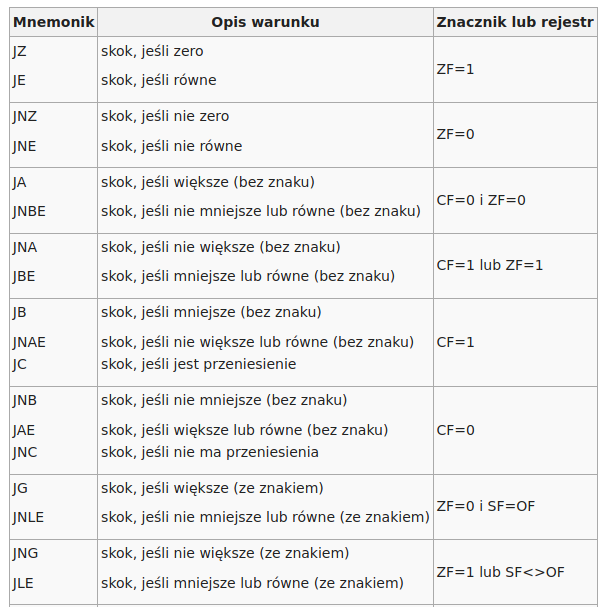
# Assembler

mov d,s <> d=s add d,s <> d=d+s sub d,s <>d=d-s neg c c=-c

**mul z (dx:ax)=ax\*z <> dx=0, ax<=16bit else dx liczby overflow ax**

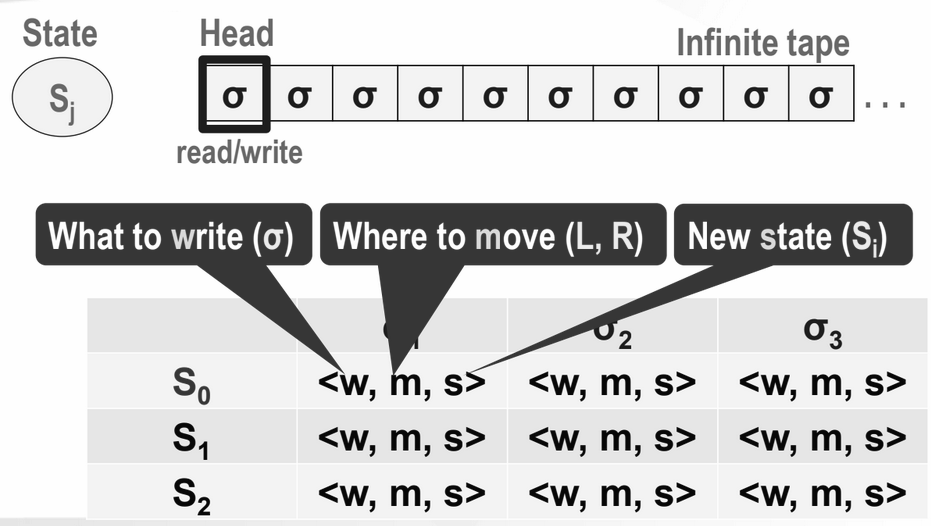
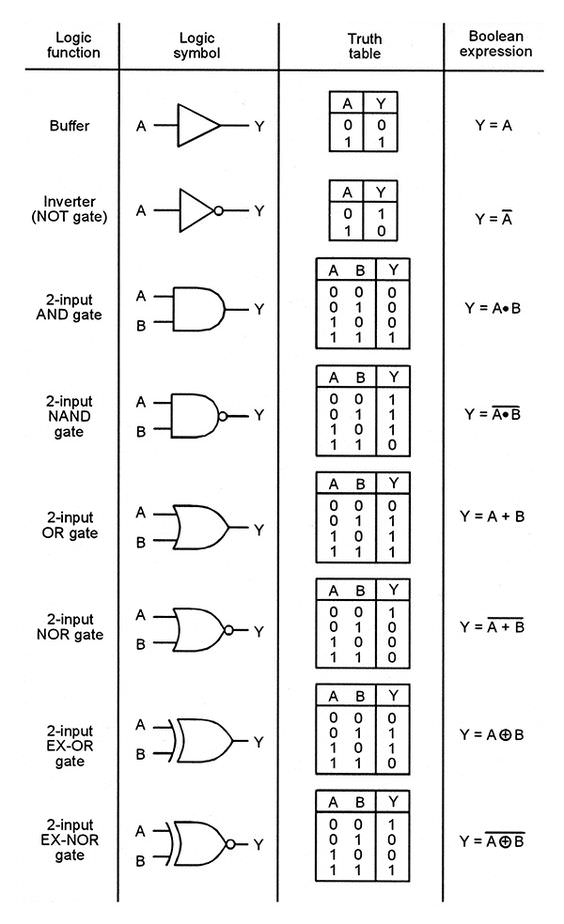
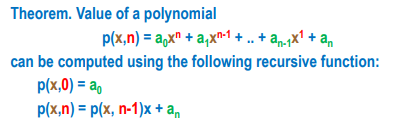
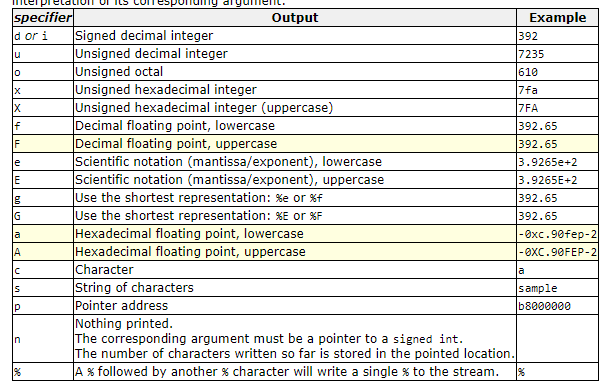
**div z <> {ax=(dx:ax)/z, dx=(dx:ax)%z}**

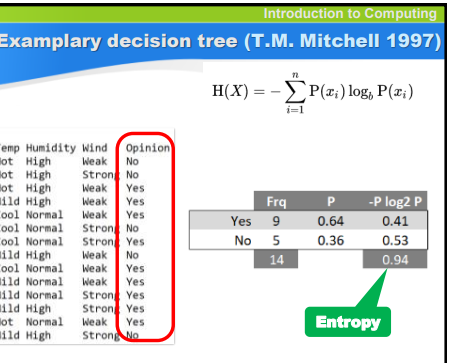
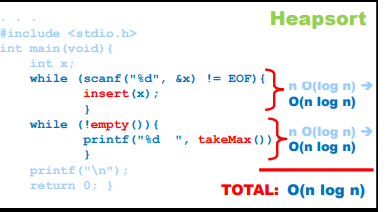


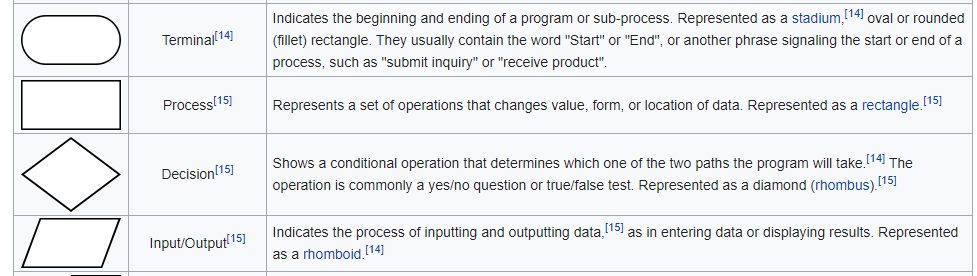


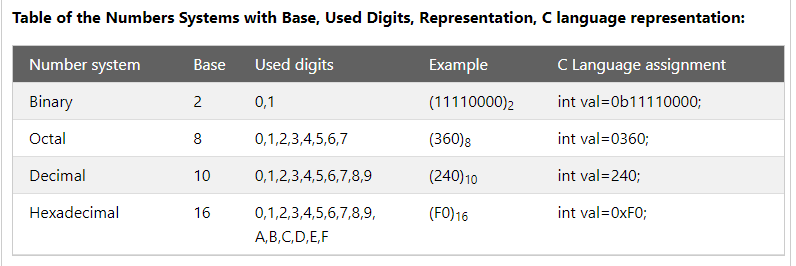
Rejestry: EAX, EBX, ECX, EDX 32-bitowe,

Mniej znaczące 16 bitów rejestrów tworzy rejestry 16 bitowe AX, BX, CX, DX, BP, DI, SI, SP o tym samym przeznaczeniu. 8 bitów mniej znaczących rejestrów AX, BX, CX, DX tworzy 8 bitowe rejestry AL, BL, CL, DL, zaś 8 bitów bardziej znaczących tworzy 8 bitowe rejestry AH, BH, CH, DH.







The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems in a concurrent system such as a multitasking operating system. A trivial semaphore is a plain variable that is changed (for example, incremented or decremented, or toggled) depending on programmer-defined conditions.

A useful way to think of a semaphore as used in a real-world system is as a record of how many units of a particular resource are available, coupled with operations to adjust that record safely (i.e., to avoid race conditions) as units are acquired or become free, and, if necessary, wait until a unit of the resource becomes available.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called counting semaphores, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called binary semaphores and are used to implement locks.

Bitwise operations

------------------

Bitwise operators treat every bit in a word as a Boolean (two-value)

variable, apply a column-wise Boolean operator, and generate a result.

Unlike binary math, there is no carry or borrow - every column is

independent.

Examples of 4-bit bitwise AND and OR (written in C or C++ as "&" and "|"):

0110 = 6h 0110 = 6h

AND 1010 = Ah OR 1010 = Ah

---- ----

EQUALS 0010 = 2h EQUALS 1110 = Eh

In C language:

int foo = 0x6 & 0xA; /\* bitwise AND gives: foo <-- 2h \*/

int foo = 0x6 | 0xA; /\* bitwise OR gives: foo <-- Eh \*/

Examples of 4-bit binary bitwise "NOT" (C language uses a tilde operator "~"):

NOT 0110 equals 1001 -or- ~6h equals 9h

NOT 1010 equals 0101 -or- ~Ah equals 5h

NOT 0000 equals 1111 -or- ~0h equals Fh

NOT 1111 equals 0000 -or- ~Fh equals 0h

You will recoginze the above values from the 4-bit "hex bit flip" table.

The NOT operator flips every bit.

Note that the word length (number of bits) is critical when performing

a NOT operation, since \*every bit\* in the word flips, even the bits you

don't see:

In C language:

char x = ~0; /\* result is 0xFF \*/

short x = ~0; /\* result is 0xFFFF \*/

int x = ~0; /\* result is 0xFFFFFFFF for 32-bit int \*/

int x = ~0; /\* result is 0xFFFFFFFFFFFFFFFF for 64-bit \*/

char x = ~0xF; /\* result is 0xF0 \*/

short x = ~0xF; /\* result is 0xFFF0 \*/

int x = ~0xF; /\* result is 0xFFFFFFF0 for 32-bit int \*/

int x = ~0xF; /\* result is 0xFFFFFFFFFFFFFFF0 for 64-bit \*/

int x = ~0x1; /\* result is 0xFFFFFFFE for 32-bit int \*/

int x = ~0xFFFF; /\* result is 0xFFFF0000 for 32-bit int \*/

int x = ~0xFFFF0000; /\* result is 0x0000FFFF for 32-bit int \*/

Note that in C language the bit flip of character 0xF is 0xF0 and not 0x0,

since character 0xF is really stored as 0x0F (8 bits) and all 8 bits flip.

If you store 0xF in a 32-bit "int", all 32 bits flip, so ~0xF = 0xFFFFFFF0.

Labki WDI emaile

1)

1. mov dx, 258 - przypisuje wartość 258 do 16-bitowego rejestru dx

2. printReg dx - wywołuje makro printReg z rejestrem dx

3. return0 - wywołuje makro kończące program

makro printReg - wyświetla nazwę rejestru przekazanego do makra

oraz 4-cyfrową wartość w systemie heksadecymalnym

przy przekazanej liczbie ujemnej -1 wyświetla się jako FFFF (max), -2 jako FFFE, etc.

65535 wyświetla się również jako FFFF

2)

mov ax, -1: ax = X

mov bx, ax: bx = X

add ax, ax: ax = X + X

add ax, bx: ax = X + X + X

dla wartości ujemnej -1 program wyświetla FFFD (-3)

3)

cmp 0, 0 - zero

cmp 1, 0 - --

cmp 1, -1 - carry

cmp -1, -1 - zero

cmp -100, -200 - --

cmp -100, 200 - sign

cmp 100, 200 - carry, sign

cmp 100, -200 - carry

CF carry flag w wyniku dodawania/odejmowania przekroczono możliwy zakres wartości zmiennej

ZF zero flag wynik ostatniego działania wyniósł 0

SF sign flag bit znaku jest równy 1 (liczba jest ujemna)

OF overflow flag bit znaku zmienił swoją wartość, ale nie doszło do przeniesienia poza zakres (tzn. CF=0)

\_start:

mov ax, -1

mov bx, 3

; program odejmuje wieksza liczbe od mniejszej, dopoki nie beda sobie rowne

; jesli jedna z liczb = 0, oryginalny program wpadnie w nieskonczona petle

; dla -1 i 3 program wpadnie w nieskonczona petle

zabezpieczenie:

cmp ax, 0

je koniec

cmp bx, 0

je koniec

start:

cmp ax, bx ; porownuje ax i bx

je koniec ; jesli ax == bx: konczy program

jle else ; jesli ax < bx: bx -= ax

sub ax, bx ; jesli ax > bx: ax -= bx

jmp petla

else:

sub bx, ax ; bx -= ax

petla:

jmp start

koniec:

int 3

5)

section .text

global \_start

\_start:

mov ax, 1234

mov bx, 4

mov cx, 0

\_loop:

cmp ax, bx

jl \_end ; jeśli ax < bx: koniec

push dx ; tymczasowe przechowanie DX

call \_div\_and\_save ; CX += reszta z AX/BX. AX = wynik AX/BX

pop dx

jmp \_loop

\_div\_and\_save: ; DX:AX jest dzielone przez BX

mov dx, 0 ; wynik trafia do AX, reszta do DX

div bx

add cx, dx ; CX += reszta z AX/BX

ret

; BX - system

; 1234 10 0

; 123 10 4

; 12 10 7

; 1 10 9

; ---- koniec ---

; dodana reszta, czyli 1 = 10 = 000A przy 1234

\_end:

add cx, ax

printReg cx

return0 ;return 0;