

Assembler



HUMAN LANGUAGE = C is assigned the sum of A times B plus 10



HIGH LEVEL LANGUAGE = C is assigned the sum of A times B plus 10



ASSEMBLY LANGUAGE



MACHINE LANGUAGE



BINARY CODE

Intel Assembly Language:

```
mov  eax, A
mul  B
add  eax, 10
mov  D, eax
```

Intel Machine Language:

```
A1 00404000
F7 25 00404004
83 C0 0A
A3 00404008
```

Asembler i rejestry procesora

Asembler to język programowania, należący do języków niskiego poziomu. Znaczy to tyle, że **jednej komendzie asemblera odpowiada dokładnie jeden rozkaz procesora**.

Assembler operuje na rejestrach procesora.

Rejestr procesora to zespół układów elektronicznych, mogący przechowywać informacje (=pamięć wewnętrzna procesora).

Rejestry ogólnego użytku

akumulator:

AX = **AH** (starsze 8 bitów) + **AL** (młodsze 8 bitów)

Rejestr ten najczęściej służy do wykonywania działań matematycznych, w tym rejestrze (lub jego części AX, AH) będziemy mówić SO i BIOS-owi, co od niego chcemy.

EAX (32 bity) = **AX** (młodsze 16 bitów) + starsze 16 bitów, **RAX** (64 bity) = **EAX** (młodsze 32 bity) + starsze 32 bity,

rejestr bazowy:

BX = **BH** + **BL**

Ten rejestr jest używany n.p. przy dostępie do tablic.

licznik:

CX = **CH** + **CL**

Tego rejestru używamy na przykład do określania ilości powtórzeń pętli.

rejestr danych:

DX = **DH** + **DL**

W tym rejestrze przechowujemy adresy różnych zmiennych.

Rejestry indeksowe

indeks źródłowy:

SI (16 bitów) = SIL (młodsze 8 bitów) + starsze 8 bitów (tylko 64-bit)

RSI (64 bity) = ESI (młodsze 32 bity) + starsze 32 bity,

ESI (32 bity) = SI (młodsze 16 bitów) + starsze 16 bitów,

indeks docelowy:

DI (16 bitów) = DIL (młodsze 8 bitów) + starsze 8 bitów (tylko 64-bit)

RDI (64 bity) = EDI (młodsze 32 bity) + starsze 32 bity,

EDI (32 bity) = DI (młodsze 16 bitów) + starsze 16 bitów,

Rejestry indeksowe najczęściej służą do operacji na długich łańcuchach danych, w tym napisach i tablicach.

Rejestry wskaźnikowe

wskaźnik bazowy:

RBP (64 bity) = EBP (młodsze 32 bity) + starsze 32 bity,
EBP (32 bity) = BP (młodsze 16 bitów) + starsze 16 bitów.

BP (16 bit) BPL (młodsze 8 bitów) + starsze 8 bitów (tylko 64-bit)

Najczęściej służy do dostępu do zmiennych lokalnych.

wskaźnik stosu:

RSP (64 bity) = ESP (młodsze 32 bity) + starsze 32 bity,
ESP (32 bity) = SP (młodsze 16 bitów) + starsze 16 bitów.

SP (16 bitów) = SPL (młodsze 8 bitów) + starsze 8 bitów (tylko 64-bit)

Służy do dostępu do stosu.

wskaźnik instrukcji:

RIP (64 bity) = EIP (młodsze 32 bity) + starsze 32 bity,

EIP (32 bity) = IP (młodsze 16 bitów) + starsze 16 bitów.

Mówi procesorowi, skąd ma pobierać instrukcje do wykonywania.

Rejestry segmentowe (tylko 16-bit)

segment

- kodu **CS** - mówi procesorowi, gdzie są dla niego instrukcje.
- danych **DS** - najczęściej pokazuje z naszymi zmiennymi.
- stosu **SS** - wskazuje na segment z naszym stos-em.
- dodatkowy **ES** - często używany, gdy chcemy coś napisać lub narysować na ekranie bez pomocy Windows, DOSa czy BIOSu.
- **FS i GS** (dostępne dopiero od 80386) - nie mają specjalnego przeznaczenia. Są tu na wypadek, gdyby zabrakło nam innych rejestrów segmentowych.

Rejestr stanu procesora

FLAGI (16-bit), E-FLAGI (32-bit) lub R-FLAGI (64-bit).

służą przede wszystkim do badania wyniku ostatniej operacji
(np. czy nie wystąpiło przepełnienie, czy wynik jest zerem, itp.).

Najważniejsze flagi:

CF (**carry** flag - flaga przeniesienia),

OF (**overflow** flag - flaga przepełnienia),

SF (**sign** flag - flaga znaku),

ZF (**zero** flag - flaga zera),

IF (**interrupt** flag - flaga przerwań),

PF (**parity** flag - flaga parzystości),

DF (**direction** flag - flaga kierunku).

Program w wersji TASM

```
.model tiny
.code
org 100h
start:
    mov ah, 9
    mov dx, offset info
    int 21h
    mov ah, 0
    int 16h
    mov ax, 4C00h
    int 21h
info db „Hallo.$”
end start
```

.model tiny wskazuje kompilatorowi rodzaj programu. Jest kilka takich dyrektyw:
tiny: kod i dane mieszczą się w jednym 64kB segmencie. Typowy dla programów typu .com
small: kod i dane są w różnych segmentach, ale obydwa są mniejsze od 64kB i dalej: **medium**:
compact: **large**: **huge**:

.code Wskazuje początek segmentu, gdzie znajduje się kod programu. Inne dyrektywy:
.data, deklarująca początek segmentu z danymi
.stack, deklarująca segment stosu

org 100h Ta linia mówi kompilatorowi, że nasz kod będzie znajdował się pod adresem 100h w swoim segmencie. To jest typowe dla programów .com. DOS,

start: (z dwukropkiem) i **end start** (bez dwukropka)
Mówią kompilatorowi, gdzie są odpowiednio: początek i koniec programu.

Program w wersji TASM

```
.model tiny
.code
org 100h
start:
    mov     ah, 9
    mov     dx, offset info
    int     21h
    mov     ah, 0
    int     16h
    mov     ax, 4C00h
    int     21h
info db     "Czesc.$"
end start
```

INT = interrupt = przerwanie.

Wywołując przerwanie 21h uruchamiamy jedną z funkcji DOSa. Którą? O tym zazwyczaj mówi rejestr AX. W spisie przerwań mamy:

INT 21

- DOS 1+

- **WRITE STRING TO STANDARD OUTPUT**
AH = 09h
- **DS:DX -> \$-terminated string**

Już widzimy, czemu w AH jest wartość 9. Chcieliśmy uruchomić funkcję, która wypisuje na ekran ciąg znaków zakończony znakiem dolara. Adres tego ciągu musi się znajdować w parze rejestrów: DS wskazuje segment, w którym znajduje się ten ciąg, a DX - jego adres w tym segmencie. Dlatego było

mov dx, offset info

Program to display a message

```
; This is a simple program to display a message
.MODEL SMALL                      ; Ignore the first 4 four lines
.STACK 200h
.CODE                             ; starts code segment
ASSUME CS:@CODE, DS:@CODE
START:                          ;generally a good name to use as an entry point
    JMP BEGIN                   ; jump to BEGIN:
Message db "This is a message!$"
BEGIN:                           ;code starts here
    mov dx, OFFSET Message      ;display a message on screen
    mov ax, SEG     Message
    mov ds, ax
    mov ah, 9
    int 21h
    mov ax, 4c00h              ;move the value of 4c00 hex into AX
    int 21h                    ; subroutine 4C returns control to dos
END START                      ;end here
```

Kolejne przerwania

- **INT 16h - KEYBOARD - GET KEYSTROKE AH = 00h**

- **Return: AH = BIOS scan code AL = ASCII character**

Ta funkcja pobiera znak z klawiatury i zwraca go w rejestrze AL. Jeśli nie naciśnięto nic, poczeka, aż użytkownik naciśnie.

- **mov ax, 4c00h** ;Do rejestru AX wpisujemy wartość 4c00 szesnastkowo.

- **int 21h "EXIT" - TERMINATE WITH RETURN CODE AH = 4Ch AL = return code**

- Jak widzimy, ta funkcja powoduje wyjście z powrotem do DOSa, z numerem błędu (errorlevel) w AL równym 0. Przyjmuje się, że 0 oznacza, iż program zakończył się bez błędów. Jak widać po rozmiarze rejestru AL (8 bitów), program może wyjść z $2^8=256$ różnymi numerami błędu.

- **info db „Hallo! \$"**

- Etykietą info opisujemy kilka bajtów, w tym przypadku zapisanych jako ciąg znaków. A po co znak dolara \$? Jak sobie przypomnimy, funkcja 9. przerwania DOSa wypisuje ciąg znaków zakończony właśnie na znak dolara \$. Gdyby tego znaczka nie było, DOS wypisywałby różne śmieci z pamięci, aż trafiłby na przypadkowy znak dolara \$ nie wiadomo gdzie.

Kolejne przerwania

int 10h	przerwanie karty graficznej
int 13h	obsługa dysków
int 15h	część BIOS-u
int 16h	obsługa klawiatury
int 21h	DOS

int 10h

przerwanie karty graficznej

funkcja 0 - ustaw tryb graficzny:

Argumenty:

- AH = 0
- AL = żądany tryb graficzny (patrz niżej)
- Podstawowe tryby graficzne i ich rozdzielczości:
 - 0 - tekstowy, 40x25, segment 0B800
 - 1 - tekstowy, 40x25, segment 0B800
 - 2 - tekstowy, 80x25, segment 0B800
 - 3 - tekstowy (tradycyjny), 80x25, segment 0B800
 - 12h - graficzny, 640x480 w 16/256tys. kolorach, segment 0A000
 - 13h - graficzny, 320x200 w 256 kolorach, segment 0A000

funkcja 2 - ustaw pozycję kursora tekstowego:

Argumenty:

- AH = 2
- BH = numer strony, zazwyczaj 0
- DH = wiersz (0 oznacza górny)
- DL = kolumna (0 oznacza lewą)

int 10h

przerwanie karty graficznej

funkcja 3 - pobierz pozycję kursora tekstowego i jego rozmiar:

Argumenty:

- AH = 3
- BH = numer strony, zazwyczaj 0
- Zwraca:
 - CH = początkowa linia skanowania (górna granica kursora)
 - CL = końcowa linia skanowania (dolna granica kursora)
 - DH = wiersz (0 oznacza górny)
 - DL = kolumna (0 oznacza lewą)

funkcja 0Eh - wypisz znak na ekran:

Argumenty:

- AH = 0Eh
- AL = kod ASCII znaku do wypisania
- BH = numer strony, zazwyczaj 0
- BL = kolor (tylko w trybach graficznych)

funkcja 0Fh - pobierz tryb graficzny:

Argumenty:

- AH = 0Fh
- Zwraca:
 - AH = liczba kolumn znakowych
 - AL = bieżący tryb graficzny
 - BH = aktywna strona

funkcja 2 - czytaj sektory dysku do pamięci:

Argumenty:

- AH = 2
- AL = liczba sektorów do odczytania (musi być niezerowa)
- CH = najmłodsze 8 bitów numeru cylindra
- CL =
bity 0-5: numer sektora (1-63)
bity 6-7: najstarsze 2 bity numeru cylindra (tylko dla twardych dysków)
- DH = numer głowicy
- DL = numer dysku, dla dysków twardych bit7=1 (0=dysk A:, 1=B:, 80h=C:, 81h=D:, ...)
- ES:BX = adres miejsca, gdzie będą zapisane dane odczytane z dysku

- Zwraca:

- flaga CF=1, jeśli wystąpił błąd; CF=0, gdy nie było błędu
- AH=status (patrz niżej)
- AL=liczba przeczytanych sektorów (nie zawsze prawidłowy)

- Podstawowe wartości statusu:

- 0 - operacja zakończyła się bez błędów
- 3 - dysk jest chroniony przed zapisem
- 4 - sektor nie znaleziony / błąd odczytu
- 6 - zmiana dyskietki. Najczęściej spowodowany tym, że napęd nie zdążył się rozpędzić. Ponowić próbę.
- 80h - przekroczony limit czasu operacji. Dysk nie jest gotowy.

int 13h

obsługa dysków

Przykład (czytanie bootsektora):

```
mov ax, 0201h ; funkcja czytania sektorów
xor  dx, dx    ; głowica 0, dysk 0 = A:
mov  cx, 1     ; numer sektora
mov  bx, bufor ; dokąd czytać
int  13h       ; czytaj
jnc  czyt_ok   ; sprawdź, czy błąd
```

DIFFERENCE BETWEEN RISC AND CISC

RISC means	CISC means
Reduced Instruction Set Computer. A RISC system has reduced number of instructions	Complex Instruction Set Architecture. A CISC system has complex instructions such as direct addition between data in two memory locations.
Uniform instruction format , using a single word with the opcode in the same bit positions in every instruction, demanding less decoding	Complex instruction, more addressing mode
RISC architecture is not widely used	CISC architecture is widely used At least 75% of the processor use CISC architecture
RISC puts a greater burden on the software. Software developers need to write more lines for the same tasks	CISC, software developers no need to write more lines for the same tasks
Mainly used for real time applications	Mainly used for real time applications Mainly used in PC's, WS's & servers

8086 Instruction Set

- **Data Transfer** Instructions
- **Logical** Instructions
- **Shift and Rotate** Instructions
- **Arithmetic** Instructions
- **Transfer** Instructions
- **String** Instructions
- **Subroutine and Interrupt** Instructions
- **Processor Control** Instructions

8086 Instruction Set

- **Data Transfer** Instructions

MOV	Move byte or word to register or memory
IN, OUT	Input byte or word from port, output word to port
LEA	Load effective address
LDS, LES	Load pointer using data segment, extra segment
PUSH, POP	Push word onto stack, pop word off stack
XCHG	Exchange byte or word
XLAT	Translate byte using look-up table

- **Logical** Instructions

NOT	Logical NOT of byte or word (one's complement)
AND	Logical AND of byte or word
OR	Logical OR of byte or word
XOR	Logical exclusive-OR of byte or word
TEST	Test byte or word (AND without storing)

8086 Instruction Set (2)

- **Shift and Rotate Instructions**

- **SHL, SHR** Logical shift left, right byte or word by 1 or CL
- **SAL, SAR** Arithmetic shift left, right byte or word by 1 or CL
- **ROL, ROR** Rotate left, right byte or word by 1 or CL
- **RCL, RCR** Rotate left, right through carry byte or word by 1 or CL

- **Arithmetic Instructions**

- **ADD, SUB** Add, subtract byte or word
- **ADC, SBB** Add, subtract byte or word and carry (borrow)
- **INC, DEC** Increment, decrement byte or word
- **NEG** Negate byte or word (two's complement)
- **CMP** Compare byte or word (subtract without storing)
- **MUL, DIV** Multiply, divide byte or word (unsigned)
- **IMUL, IDIV** Integer multiply, divide byte or word (signed)
- **CBW, CWD** Convert byte to word, word to dword (useful before mul/div)
- **DAA, DAS** Decimal adjust for add, sub (binary coded dec numbers)

8086 Instruction Set (3)

- **Transfer Instructions**

- **JMP** Unconditional jump

- *Conditional jumps:*

- **JA (JNBE)** Jump if above (not below or equal), +127, -128 range only
- **JAE (JNB)** Jump if above or equal (not below), +127, -128 range only
- **JB (JNAE)** Jump if below (not above or equal), +127, -128 range only
- **JBE (JNA)** Jump if below or equal (not above), +127, -128 range only
- **JE (JZ)** Jump if equal (zero), +127, -128 range only
- **JG (JNLE)** Jump if greater (not less or equal), +127, -128 range only

- *Loop control:*

- **LOOP** Loop unconditional, count in CX, short jump to target address
- **LOOPE (LOOPZ)** Loop if equal (zero), count in CX, short jump to target address
- **LOOPNE (LOOPNZ)** Loop if not equal (not zero), count in CX, short jump to targetaddress
- **JCXZ** Jump if CX equals zero (used to skip code in loop)

8086 Instruction Set (4)

- **String Instructions**

- **MOVS** Move byte or word string
- **MOVSB, MOVSW** Move byte, word string
- **CMPS** Compare byte or word string
- **SCAS** Scan byte or word string (comparing to A or AX)
- **LODS, STOS** Load, store byte or word string to AL or AX
- ***Repeat instructions*** (*placed in front of other string operations*):
 - **REP** Repeat
 - **REPE, REPZ** Repeat while equal, zero
 - **REPNE, REPNZ** Repeat while not equal (zero)

- **Subroutine and Interrupt Instructions**

- **CALL, RET** Call, return from procedure (inside or outside current segment)
- **INT, INTO** Software interrupt, interrupt if overflow
- **IRET** Return from interrupt

8086 Instruction Set (5)

- **Processor Control** Instructions

- *Flag manipulation:*

- **STC, CLC, and CMC** Set, clear, complement carry flag
- **STD, CLD** Set, clear direction flag
- **STI, CLI** Set, clear interrupts enable flag
- **LAHF, SAHF** Load AH from flags, store AH into flags
- **PUSHF, POPF** Push flags onto stack, pop flags off stack

- *Coprocessor, multiprocessor interface:*

- **ESC** Escape to external processor interface
- **LOCK** Lock bus during next instruction

- *Inactive states:*

- **NOP** No operation
- **WAIT** Wait for TEST pin activity
- **HLT** Halt processor