Asembler to tzw. niskopoziomowy język programowania (a w zasadzie, grupa języków), w którym pojedyncze instrukcje w kodzie odpowiadają pojedynczym instrukcjom procesora. Programy pisane w asemblerze mają bezpośredni dostęp do pamięci i rejestrów procesora. W zasadzie brakuje jakiejkolwiek kontroli błędów – każda instrukcja jest dozwolona.

Program napisany w asemblerze jest kompilowany do postaci wykonywalnej, np. do pliku .exe, który to plik posiada określoną strukturę. Ta struktura pozwala systemowi operacyjnemu rozpoznać, jakie fragmenty kodu należy umieścić w pamięci. System operacyjny odpowiada też za pewne modyfikacje wczytywanego kodu, np. za zagwarantowanie, że adresacja będzie odnosić się do tego miejsca w pamięci, w którym program faktycznie został umieszczony. To wiąże się np. z modyfikacją początkowej wartości niektórych rejestrów procesora (o rejestrach dalej).

Różne urządzenia (procesory, mikrokontrolery, sterowniki) mają różne asemblery, czyli zestawy instrukcji, narzędzia do kompilacji itd., niemniej składnia programów napisanych pod różne urządzenia jest podobna.

Procesor to ten element komputera, który realizuje programy, czyli wykonuje instrukcje wczytywane z pamięci. Procesor, od strony programisty, składa się z rejestrów, czyli specjalnych, szybkich komórek pamięci, które są wykorzystywane jako argumenty podstawowych instrukcji procesora.

Typowe rejestry procesora w architekturze x86 (32 bitowej) to:

- 32 bitowe rejestry ogólnego przeznaczenia: EAX, EBX, ECX, EDX
  każdy jest podzielony na część 16 bitową i 2 części 8 bitowe np. AX, AH, AL
  są podstawowymi argumentami większości instrukcji; można do nich/z nich kopiować komórki w pamięci
- 32 bitowe rejestry pomocnicze: ESI (źródło danych do niektórych instrukcji), EDI (docelowe miejsce danych), ESP (wskaźnik stosu), EBP (offset), EIP (wskaźnik instrukcji)

- 16 bitowe rejestry segmentowe: CS, DS, SS, ES, FS, GS
  w trybie rzeczywistym wskazują adres segmentu (co 16 bitów) względem którego obliczany jest adres, np. DS:CX wskazywał na adres CX + 16 * DS
  w trybie chronionym (większość dzisiejszych zastosowań) rejestry wskazują na pozycję w tablicy deskryptorów (lokalnej lub globalnej); deskryptor opisuje takie elementy pamięci jak rzeczywisty adres
  w trybie płaskiej pamięci (flat memory) każdy proces ma dostęp do ciągłego bloku pamięci i rejestry te nie są wykorzystywane
- rejestry kontrolne: CR0 (32bit np. bit 0: czy system jest w real/protected), CR2, CR3, CR4
- rejestry debugowania: DR0, DR1, DR2, DR3
- rejestr FLAG: EFLAGS

Pamięć to ciąg bajtów o rosnących adresach typowo zapisywanych w sposób szesnastkowy

```
0x00000000
0x00000001
0x00000002
...
0x0000000A
...
0x0000000F
0x00000010
0x00000011
...
```

Adresacja to wybór komórki lub komórek w pamięci, do których należy zapisać/z których należy odczytać wartości. Kluczowy jest tutaj rozmiar adresowanego obszaru, typowo może to być: bajt, 2 bajty (słowo/word), 4 bajty (podwójne słowo/dword) itd.

Np. instrukcja asembera mov a, b przenosi wartość z miejsca a do miejsca b:

```
mov EAX, 0x00000002 ;; zapisze zawartość rejestru EAX
  ;; do komórek 0x0...02 do 0x0...05
mov AX, 0x00000002 ;; zapisze zawartość rejestru AX
  ;; do komórek 0x0...02 i 0x0...03
```

Stos jest specjalnym miejscem w pamięci. Jest to kolejka FIFO, do której dokłada się dane poleceniem `push X`, a pobiera poleceniem `pop X`. Z każdym dodaniem/pobraniem zmienia się wartość wskaźnika stosu ESP: zmniejsza przy dodaniu, zwiększa przy pobraniu. Np.

```
ESP = 0x000000010
0x00000010 11
0x00000011 12
0x00000012 13
0x00000013 14
pop EAX ;; zapisze wartość 14131211 do EAX
  ;; i zmieni ESP = 0x000000014
push 01020304 ;; zmieni ESP = 0x00000000C
```

Typowo, na stosie przechowywane są argumenty wywoływanych procedur i adresy powrotu (do kodu, który wywołał procedurę).

Są trzy standardowe sygnatury instrukcji asemblera:

```
operand
operand argument1
operand argument1, argument2
```

Przykładowo:

```
nop ;; nie robi nic przez jeden 'cykl' (pomijając
  ;; mechanizmy jak wielopotokowość) procesora
jmp 500 ;; przenosi wykonywanie kodu pod względny adres 500
  ;; skoki to podstawowy mechanizm realizacji instrukcji
  ;; warunkowych, pętli i wywołań procedur
cmp ax, bx ;; porównuje zawartość rejestrów ax, bx
  ;; rezultat porównania jest zapisywany w rejestrze flagowym
  ;; z wartości może odczytać inna instrukcja
```

Kod jest dzielony na segmenty, zależnie od przeznaczenia zawartych w nim treści

```
.text ;; właściwy kod - instrukcje w pliku typu
  ;; Portable Executable (alternatywa .code)
.data ;; właściwe dane - zapisane komórki w pamięci
.bss ;; dane tworzone dopiero przy ładowania programu
```

Portable Executable to jeden z format plików wykonywalnych w Windowsie. Posiada segmenty: code (text), data, idata (importowane funkcje z bibliotek), bss.

`kernel32.dll` i `user32.dll` to dynamicznie łączone biblioteki w systemie Windows. Zawierają funkcje do obsługi wejść, wyjść, wątków, okien i konsoli.

Spis funkcji
`https://www.pinvoke.net/`

```
[DllImport("user32.dll")]
static extern IntPtr CreateMenu();

[DllImport("user32.dll", CharSet=CharSet.Auto)]
static extern uint CharLowerBuff([In,Out] StringBuilder lpsz
  uint cchLength);
```

Dodatkowo (C):
`https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-api-list`

Netwide Assembler: http://www.nasm.us/
Anthony's Linker: http://alink.sourceforge.net/
OllyDbg: http://www.ollydbg.de/

Kompilacja i linkowanie:

```
nasm -fobj filename.asm
```

* -fobj : plik wynikowy to obj

```
alink -oPE filename.obj
```

* -oPE : plik wynikowy to Portable Executable

Jeśli mamy strukturę katalogów

```
..\
..\alink
..\nasm
..\inny
```

To z katalogu `inny` możemy kompilować makrem

```
;; _make.bat
for %%f in (*.asm) do (
    ..\nasm\nasm -fobj "%%~nf.asm"
    ..\alink\alink -oPE "%%~nf.obj"
    "%%~nf.exe"
)
```