Wykonał: Bystryk Dominik	Wydział: Informatyka Techniczna Filia JG Rok: 2 Nr grupy: 1 Rok akadem.: 2023/2024
Organizacja i architektura komputerów (laboratorium)	
Data zakończenia projektu: **26.03.2024 r.**	**Temat ćwiczenia laboratoryjnego:** ***Kalkulator dużych liczb heksadecymalnych***

Streszczenie – Ćwiczenie polegało na przygotowaniu programu, który dla dwóch dużych liczb A i B w pozycyjnym zapisie szesnastkowym wykonywałby operacje dodawania, odejmowania i mnożenia. Program napisany w języku assembly na procesor x86

1. Wykorzystane oprogramowanie

- Virtual Box ver. 7.014
- Ubuntu 23.10
- Assembler NASM ver 2.16.01
- Text Editor wbudowany w OS

2. Cel i zakres ćwiczenia oraz opis sposobu wykonania ćwiczenia.

Celem ćwiczenia było przygotowanie programu w języku assembly, który miał, dla dwóch dużych (do 100 bajtów) liczb w zapisie pozycyjnym szesnastkowym wykonywać działania algebraiczne t.j. dodawanie, odejmowanie oraz mnożenie.

Dodatkowo program miał pobierać argumentu z wiersza poleceń i działać na architekturze x86 (32-bit).
Kompilacja (assemblowanie) programu miało zostać zautomatyzowane za pomocą pliku Makefile.

Ze względu na brak wcześniejszego kontaktu z językiem assembly, po wstępnym zaznajomieniu się z podstawami języka przygotowano następujący sposób wykonania ćwiczenia:
a. Przygotować program pobierający argumentu z wiersza poleceń i wypisujący je na ekranie
b. Do wyżej wymienionego programu przygotować możliwość wpisania tychże argumentów (tylko liczb A i B oraz znaku) do odpowiednich tablic, alokowanych w pamięci przydzielanej programowi.
c. Dopisać podstawowe porównanie, od którego będzie zależało dalsze działanie programu (porównanie wartości argumentu „znak”
do odpowiedniego znaku algebraicznego)
d. Po prawidłowym przyrównaniu znaku przekonwertować „półbajt po półbajcie” tablice zawierające liczby z ASCII na hex
e. Wykonanie działania arytmetycznego zgodnie z wprowadzonym wcześniej znakiem arytmetycznym i wpisanie wyniku do tabeli „wynik”
f. Przekonwertowanie otrzymanego wyniku z systemu Hex na ASCII i wypisanie ich w terminalu.

g. Wprowadzenie ewentualnych możliwości kontroli błędów użytkownika przy wprowadzaniu argumentów (nieprawidłowy znak arytmetyczny/błędna dana podana przez użytkownika)

3. **Wykonanie zadania**

Ze względu na fakt, iż nie miałem wcześniej kontaktu z językiem assembly stosunkowo ciężko było mi nie pogubić się w gąszczu nowej nomenklatury oraz szeregu „ograniczeń" jakie nakłada na użytkowniku praca na rejestrach.

W związku z tym, oraz szeregiem nieznanych mi i niezrozumiałych błędów podczas „kompilacji", udało mi się wykonać następujące elementy zadania (elementy zamieszczone w kodzie programu):
- program pobiera argumenty z wiersza poleceń oraz wypisuje je na ekranie
- program przepisuje podane argumenty do odpowiednich tablic (pomijając znak, do którego odwołuję się bezpośrednio do stosu);
- program porównuje i wybiera odpowiednie działanie arytmetyczne
- program ma możliwość sprawdzenia czy został podany właściwy znak arytmetyczny
- program wykonuje prawidłowo dodawanie i odejmowanie
- program wykonuje prawidłowo mnożenie pierwszej liczby przez **jedną cyfrę/literę** drugiej liczby podanej przez użytkownika – algorytm działa dla cyfr w systemie hexadecymalnym od 0 do 9 oraz dla litery a;
przy literach B-F dochodzi do zjawiska gubienia przeniesienia
- zaimplementowano algorytm dzielenia, który przeprowadza dzielenie i potrafi wypisać prawidłowy wynik z resztą z dzielenia; Wymaga on jednak jeszcze poprawy oraz uwzględnienia sytuacji, w której, przy 4 bajtowym dzielniku, 4 bajty dzielnej są mniejsze od dzielnika. (w obecnym stanie programu dojdzie jedynie do zjawiska overflow na rejestrze eax)
- program prawidłowo wykonuje konwersję ASCII do Hex oraz konwersję odwrotną Hex do ASCII

4. **Algorytmy oraz ich opis**

a) pobieranie argumentów z wiersza poleceń:
- krótki opis

./nazwa_programu arg1 arg2 arg3
Argumenty, z którymi wywołany jest nasz program są umieszczane na stacku, aby się do nich dostać korzystam z możliwości jakie dają wskaźniki stosu (EBP oraz ESP). Poprzez odpowiednie adresowanie mam możliwość pobrać argumenty, umieścić ich wartości w rejestrach, by następnie skorzystać z innych możliwości operacji na argumentach.
W przypadku naszego programu mamy 3 argumenty:
Arg 1 to liczba nr 1
Arg 2 to znak arytmetyczny określający jakie działanie chcemy wykonać
Arg 3 to liczba nr 2

b) pobieranie długości string

Ponieważ działania arytmetyczne chciałem przeprowadzić w sposób „bajt po bajcie" z tablic chciałem wiedzieć ile liczb ma każdy string (żeby poprawnie wpisać je do tablic – w sposób jaki przygotowałem wg algorytmów)
- opis algorytmu:
> wyczyść rejestr ebx (będzie potrzebny jako licznik i jako wyznacznik długości podanej

liczby)
> porównaj każdy znak stringa od miejsca wskazywanego przez eax+ebx (w eax mamy de facto offset wpisanego stringa) i sprawdź czy nie jest równy 0 (NULL/koniec stringa)
>jeżeli nie to wróć do początku po inkrementacji wskaźnika
>jeżeli tak przerzuć znalezioną długość liczby znajdującą się w ebx do pamięci; do zmiennej, która jest powiązana z daną liczbą.

Po pobraniu długości każdej liczby i wprowadzeniu ich do odpowiednich „zmiennych" - strlen1 dla liczby 1 i strlen2 dla liczby 2 – wypisujemy je na ekranie terminala.

c) Wpisywanie stringa do tablicy
Aby ułatwić sobie operacje na stringach wpisuję obie liczby do odpowiadających im tablic Num1_arr oraz Num2_arr przesuwając je o 1 półbajt w prawo, by mieć na początkach tablic „zero wiodące", które przyda się w późniejszym etapie.
- opis algorytmu:
> Czyszczę wszystkie rejestry (na wszelki wypadek)
> Rejest wskaźnikowy ESI ustawiam na odpowiednią liczbę, którą chcę umieścić w tablicy
> EDI ustawiony jest na 1. element tablicy, do której chcę przepisać liczbę ze stosu
> ECX, stanowiący licznik potrzebnych przejść pętli, zawiera długość liczby, którą chcemy wpisać
> pętla wpisująca liczbę pobiera bajt (półbajt) liczby wprowadzonej i umieszcza go w tablicy Numx (x – numer wprowadzanej liczby) na pozycji +1 względem oryginalnego miejsca w liczbie.
Rezultatem, np. dla liczby AEFFD9 jest 0x00, 0x0A, 0x0E, 0x0F, 0x0F, 0x0D, 0x09 w tablicy Numx.

d) Wypisanie znaku i sprawdzenie go
Program działa na zasadzie: Przygotuj arg 1, sprawdź arg 2, przygotuj arg 3, dlatego w kodzie, mimo iż wspomniałem wyżej o przepisywaniu do tablic i wyszukiwaniu długości obu liczb pomiędzy nimi znajduje się sprawdzenie oraz wypisanie znaku podanego przez użytkownika.
Program rozpoznaje następujące znaki:
- „+" dla dodawania
- „-" dla odejmowania
- „ '*'" dla mnożenia (podawane w terminalu jako '*' ze względu na OS)
- „ '/'" dla dzielenia (podawane w terminalu jako '/' ze względu na OS)
W przypadku podania znaku, który nie został wyżej wymieniony wyświetli się informacja o wpisaniu błędnego znaku i pogram zakończy działalność.
Po porównaniu zostaje wypisany odpowiedni znak algebraiczny i następuje przejście do wypisania i opracowania liczby 2.

e) konwersja ASCII do HEX

Po wpisaniu liczby do tablicy program konwertuje liczbę wpisaną przez użytkownika do systemu hexadecymalnego.
Odbywa się to w następujący sposób:
> Ustawiamy wskaźnik na tablicę z liczbą (rejestr EDI) oraz ustawiamy nasz licznik na długość liczby w tablicy (rejestr ECX).
> Sprawdzamy półbajt po półbajcie liczby hexadecymalnej czym jest – cyfrą, dużą literą,

małą literą (uwzględniamy 0 na początku, które pochodzi z przepisania; w rezultacie sprawdzenie zaczynamy od EDI + 1).
> Za pomocą odpowiedniej funkcji, bazując na tablicy kodów ASCII przeprowadzamy konwersję do HEX i nadpisujemy sprawdzany element tablicy na wejściu.
> Pętla trwa do czasu aż ecx nie osiągnie 0.
Dodatkowo w tym miejscu sprawdzana jest poprawność danych – czy użytkownik nie podał przypadkiem cyfry/znaku, który nie należy do systemu hexadecymalnego.

f) wybór odpowiedniej operacji arytmetycznej
Następnie program ponownie pobiera wartość argumentu 2 ze stosu (znaku) i ponownie sprawdza, już bez kontroli błędów, który znak został wybrany.
Po sprawdzeniu przeskakuje do odpowiedniego działania arytmetycznego.
g) Dodawania – liczsum
Jeżeli zostanie wykryty znak „+" program wykona dodawanie.
Program jest ustawiony tak, by mógł dodawać dowolnej długości liczby (maksymalnie 100 bajtowe hexadecymalne) niezależnie od tego, która z nich jest dłuższa/krótsza.
Dlatego na samym początku dochodzi do sprawdzenia długości podanych liczb i wybraniu odpowiedniego ustawienia wskaźników.
Po wyborze odpowiedniego ustawienia wskaźników dochodzi do przepisania tablic Numx_arr do tablic Wynik_arr oraz PrzepisanaLiczba_arr, celem dostosowania obu liczb do najdłuższej z nich.
Np. przy dodawaniu liczb 4E + 4 dodawane były tablice: 0x04, 0x0E + 0x00, 0x04 a nie 0x04, 0x0E + 0x04 (4E + 40).
Po przepisaniu następuje wywołanie za pomocą instrukcji call faktycznej funkcji rozpoczynającej proces dodawania obu liczb.

-opis algorytmu:
> Pobierz ostatni półbajt z 1. liczby (do al) oraz ostatni półbajt 2. liczby (do bl);
> dodaj je (wynik znajduje się w al) i sprawdź czy nie doszło do przeniesienia
> jeżeli w al mamy liczbę większą lub równą 0x10 tzn. że doszło do przeniesienia → funkcja do_reszty; jeżeli nie doszło do przeniesienia w Wynik_arr dochodzi do nadpisania ostatniego półbajt zawartością al i przejście do głównej pętli dodawania
> funkcja do_reszty sprowadza się do końcowego elementu poprzedzającej funkcji sumator – odejmuje nadmiar od wyniku dodawania ostatnich półbajtów oraz wpisuje zawartość rejestru AL do tablicy. Po jej wykonaniu przechodzimy do funkcji, która zajmuje się przeniesieniem → is_carry
> funkcja is_carry dodaje przeniesienie (0x01) do kolejnego półbajtu liczby, o ile nie jest on ostatnim bajtem cyfry lub nie ma wartości litery F.
Funkcja sprawdza i działa tak długo aż:
* nie znajdzie bajtu (półbajtu), do którego może dodać liczbę – np. FFFFF + 1; co spowoduje dodanie do naszego „zera wiodącego" w tablicy 1 i zamianę pozostałych F na 0 - liczba FFFFF zamieni się w 100000, po pierwszej operacji dodawania
* znajdzie półbajtu do którego może dodać 1 z przeniesienia – np. AEF + 1 = AF0
Po zakończeniu działania funkcja ta przechodzi do głównej pętli dodawania.
> add_loop – główna pętla dodawania.
Główna pętla dodawania działa na praktycznie takiej samej zasadzie jak dodawanie pierwszego półbajta, z tą jednak różnicą, że przywraca wskaźnik tablicy na Wynik_arr, by prawidłowo pobierał i dodawał półbajty (ponieważ operujemy na tym arrayu w przypadku wystąpienia przeniesienia).
Jeżeli pętla wykryje, że doszliśmy do ostatniego bajtu tablicy – tego, który pozostawiliśmy podczas przepisywania liczb z pamięci, przeniesie się do funkcji done

kończącej proces dodawania.
> done:
Funkcja done ma za zadanie przekonwertować otrzymany wynik z HEX na ASCII oraz wypisać go z uwzględnieniem 1. miejsca w tabeli, które może zawierać przeniesienie wynikające z dodania Num1 do Num2.
Przykładowe działanie programu przy dodawaniu dwóch, wygenerowanych za pomocą strony random.org, 100 bajtowych liczb hexadecymalnych:

```
gotha@gotha-1-2:~/Desktop/Kalkulator/Bytestyle$ ./calc a6760b85d32c5a0f84a5c62e8
2205637495214eb6c17327c4598ff6935b9c4a6c51a86e8e126b9d7eaefa6a956f14955149a23322
a3d46a3f51f2bc8f169dc6e07ba3d7d9dea2148b34dd9362365b222af407d7a3bda7a8e9b01de827
7d9f5b8096edb5b + cf653e3db5d72a6d064e8029e738c879e30c289119d85adc2de14af2e57a2b
4fe5567109e44cadd4bf3951803e249dc69b90868dd8d082e62d63d7b483877cd075a5f16fae4d91
a1b7b402651670c1e07599002a318b687a2af9a1b8cb109d2f060721c6
a6760b85d32c5a0f84a5c62e82205637495214eb6c17327c4598ff6935b9c4a6c51a86e8e126b9d7
eaefa6a956f14955149a23322a3d46a3f51f2bc8f169dc6e07ba3d7d9dea2148b34dd9362365b222
af407d7a3bda7a8e9b01de8277d9f5b8096edb5b
+
cf653e3db5d72a6d064e8029e738c879e30c289119d85adc2de14af2e57a2b4fe5567109e44cadd4
bf3951803e249dc69b90868dd8d082e62d63d7b483877cd075a5f16fae4d91a1b7b402651670c1e0
7599002a318b687a2af9a1b8cb109d2f060721c6

175DB49C38903847C8AF4465869591EB12C5E3D7C85EF8D58737A4A5C1B33EFF6AA70F7F2C57367A
CAA28F8299515E71BB02AA9C0030DC98A2283037D74F1593E7D602EED4C37B2EA6B01DB9B39D6740
324D97DA46D65E308C5FB803B42EA92E70F75FD21
```

h) odejmowanie – **liczroz**
Jeżeli program wykryje znak „-” przeprowadzi odejmowanie.
Algorytm odejmowania jest bardzo podobny do algorytmu dodawania z kilkoma znaczącymi różnicami.
>funkcja sub_with_borrow „dopisze” do odejmowanych wartości 0x10, by móc przeprowadzić odejmowanie i wpisać wartość AL do tablicy
>funkcja sub_carry będzie szukała półbajta od którego może pożyczyć 0x01, tak długo aż nie natrafi na ostatni bajt (jeżeli wszystkie inne też są zerami) lub nie natrafi na bajt z którego może pobrać zapożyczenie.
Przykładowe działanie algorytmu:

```
gotha@gotha-1-2:~/Desktop/Kalkulator/Bytestyle$ ./calc a6760b85d32c5a0f84a5c62e8
2205637495214eb6c17327c4598ff6935b9c4a6c51a86e8e126b9d7eaefa6a956f14955149a23322
a3d46a3f51f2bc8f169dc6e07ba3d7d9dea2148b34dd9362365b222af407d7a3bda7a8e9b01de827
7d9f5b8096edb5b - cf653e3db5d72a6d064e8029e738c879e30c289119d85adc2de14af2e57a2b
4fe5567109e44cadd4bf3951803e249dc69b90868dd8d082e62d63d7b483877cd075a5f16fae4d91
a1b7b402651670c1e07599002a318b687a2af9a1b8cb109d2f060721c6
a6760b85d32c5a0f84a5c62e82205637495214eb6c17327c4598ff6935b9c4a6c51a86e8e126b9d7
eaefa6a956f14955149a23322a3d46a3f51f2bc8f169dc6e07ba3d7d9dea2148b34dd9362365b222
af407d7a3bda7a8e9b01de8277d9f5b8096edb5b
-
cf653e3db5d72a6d064e8029e738c879e30c289119d85adc2de14af2e57a2b4fe5567109e44cadd4
bf3951803e249dc69b90868dd8d082e62d63d7b483877cd075a5f16fae4d91a1b7b402651670c1e0
7599002a318b687a2af9a1b8cb109d2f060721c6

FD710CD481D552FA27E5746049AE78DBD6645EC5A523ED7A017B7B476503F9956DFC415DEFCDA0C0
32BB6552918CCAB8E79099CA4516CC3BDC7BB54146DE25F9D92144C0DEF9C8FA6FB99D6D10CF4F04
239A77D500A4F121470083CC9ACC958890367B995
```

i) mnożenie – **liczmnoz**:
- obecnie mnożenie jest zaimplementowane TYLKO dla liczb, które są albo równej długości, albo liczba 1 jest dłuższa od liczby 2.
- nie udało mi się zaimplementować mnożenia przez więcej niż 1 cyfrę

- w obecnym stanie programu mnożenie działa tylko dla cyfr od 0-9 oraz dla litery a; Od litery B do F gubi pierwsze przeniesienie (i kolejne) z nieznanych mi przyczyn. Algorytm mnożenia działa bardzo podobnie do algorytmu dodawania, z tą jednak różnicą, że sprawdzenie przeniesienia odbywa się za pomocą pętli (by sprawdzić ile wynosi) oraz jest zapisywane do zmiennej *carr*, by zostać później dodanym do wyniku mnożenia kolejnego półbajtu.
Po zakończeniu operacji mnożenia program kieruje się do funkcji done_mul, która odpowiada za przekazanie przeniesienia na ostatni półbajt wyniku oraz wypisanie wyniku w terminalu.

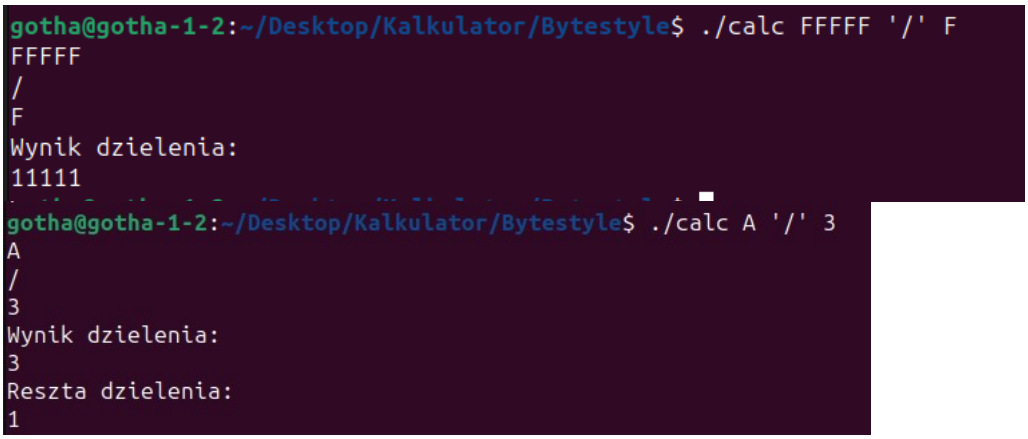
j) dzielenie – liczdziel:

Algorytm dzielenia rozpoczyna się od przeprowadzenia 3 testów:
- sprawdzana jest długość wprowadzanego dzielnika; kalkulator miał obsługiwać dzielenie przez maksymalnie 4 bajtową liczbę hexadecymalną; jeżeli podano dłuższą, program zakończy działanie
- sprawdzana jest wartość dzielnika, czy nie jest zerem; jeżeli jest program zakończy działanie z odpowiednim komunikatem błędu
- sprawdzana jest długość liczby 1. względem liczby 2.; jeżeli liczba 1. jest krótsza od liczby 2. program wykonuje „dzielenie” i wypisuje wynik wraz z resztą z dzielenia. Program nie wykonuje dzielenia „z przecinkiem”, więc jako wynik zostanie wypisane 0 a jako reszta zostanie wypisana liczba nr 1.

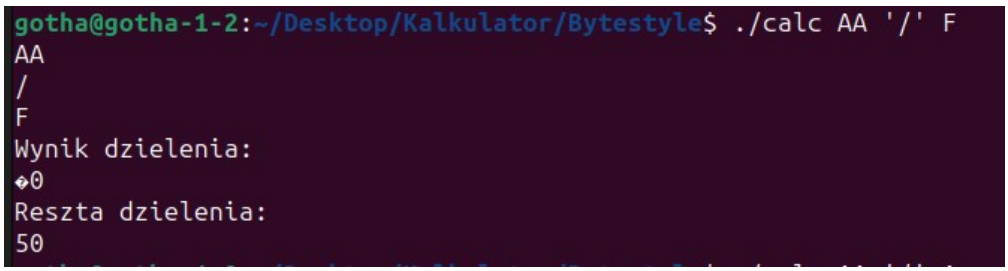
Następnym etapem jest określenie jakiej długości dzielnik podał użytkownik i poprawne przepisanie go z Num2_arr do rejestru EBX.
Po nim następuje dzielenie bajt po bajcie z uwzględnieniem wystąpienia reszty z dzielenia. Ze względu na fakt użycia instrukcji *mul ebx* nasz wynik jest zapisywany do rejestru EAX (dokładnie do AL), a reszta z dzielenia zapisywana jest do rejestru EDX (DL).
Procedura dzielenia bierze pod uwagę czy powstała reszta czy nie oraz czy wpisana wartość w rejestrze EAX jest większa bądź równa wartości w rejestrze EBX.

Niestety algorytm wymaga jeszcze dopracowania, w obecnym stanie rzeczy jest w stanie wykonać dzielenie 1 półbajt przez 1 półbajt i czasem potrafi podać prawidłowy wynik przy podziale większej liczby przez 1 półbajt, jednakże w większości przypadków wypisuje błędny wynik, czasem wypisuje prawidłową resztę mimo wypisania błędnego wyniku.

Przykładowe, poprawne działanie algorytmu dzielenia:



Błędny wynik, prawidłowa reszta (reszta jest zapisywana w ten sposób ze względu na sposób wpisywania reszty do jej własnej tablicy, nie biorę pod uwagę „zera wiodącego”



5. **Uwagi końcowe:**
   Zabrakło mi czasu na pełne przygotowanie algorytmów mnożenia i dzielenia oraz optymalizację kodu.

6. **Optymalizacja:**
   Optymalizacja w moim programie jest możliwa w każdym jego aspekcie działania.
   Od ograniczenia funkcji, które się duplikują, poprzez odpowiednie ustawienie wskaźników do optymalizacji wykorzystanych algorytmów, które pochodzą z prostego „weź kartkę, weź liczby i sprawdź krok po kroku, co masz w danym momencie zrobić” i implementacji takich algorytmów.
   Dodatkowo mój kod wymaga gruntownej restrukturyzacji oraz dopisania większej ilości komentarzy.

7. **Krótko o Makefile**
   >automatyzuje kompilację (asemblację) programu
   Zawartość programu Makefile:
   all:
   nasm -felf32 -o calc.o calc.asm
   ld -m elf_i386 -o calc calc.o
   nasm -felf32 -o calc.o calc.asm
   > asembler nasm
   -felf32 - wymuszenie trybu 32 -bit
   -o calc.o calc.asm – plik_wynikowy_asemblacji.o plik_źródłowy.asm
   ld -m elf_i386 -o calc calc.o
   > linker GNU

-m elf_i386 – wymuszenie emulacji i386
-o calc calc.o → linkowanie do pliku wynikowego z pliku źródłowego

8. Literatura

[1] notatki i materiały z wykładu „Organizacja i architektura komputerów”, źródło niepublikowane
[2] https://pl.wikibooks.org/wiki/Asembler_x86/Architektura, WikiBooks, książka o architekturze i instrukcjach assembly dla procesorów x86
[3] „NASM – The Netwide Assembly version 2.12.02”, The NASM Development Team, 1996-2016
[4] „Język asembler dla każdego”, Bogdan Drozdowski
[5] Youtube i wszelkiej tematyki filmiki instruktażowe/wykłady związane z assembly
[6] StackOverflow – przykłady użycia instrukcji w różnych elementach programowania w języku assembly
[7] Dokumentacja instrukcji Assembly x86 dla procesorów Intela