Język Pseudo-Assembler
Instrukcja Obsługi

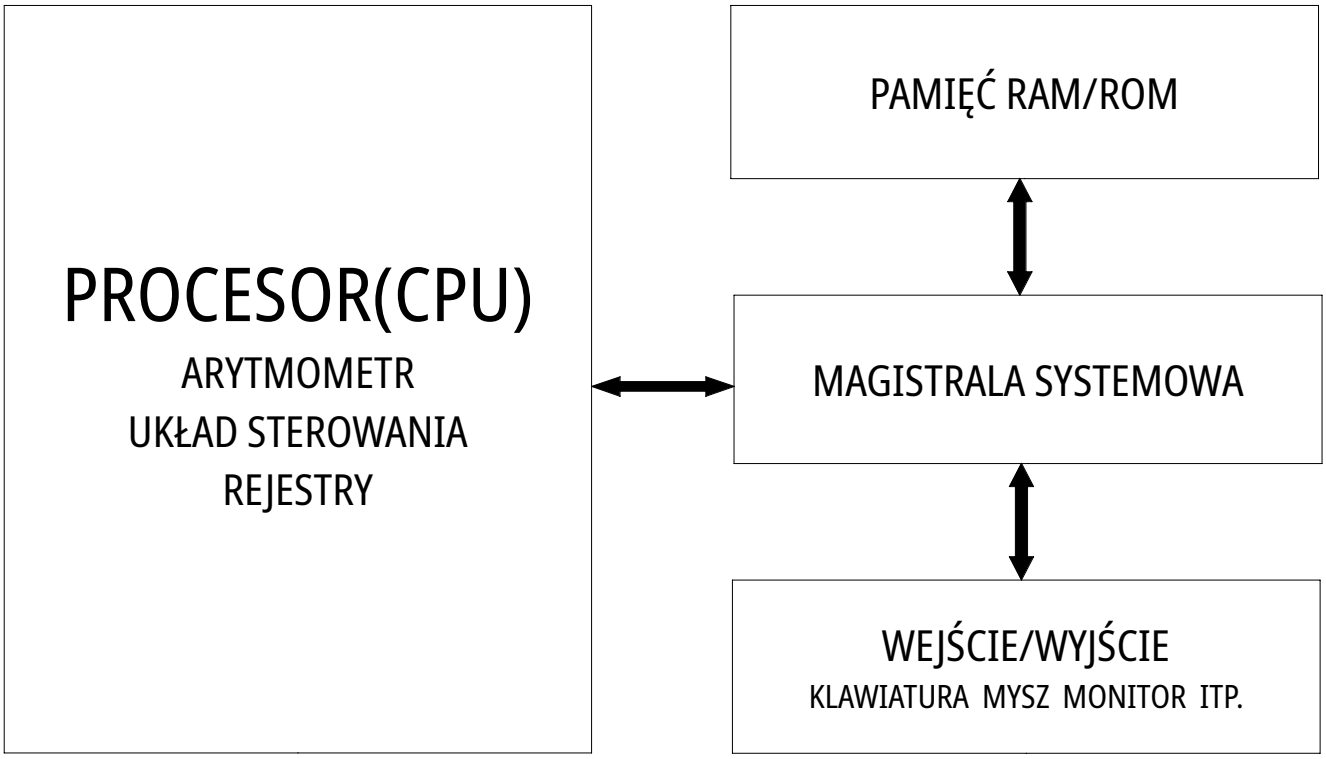
MIT License

Przedmowa

Ta Instrukcja Obsługi jest dokumentacją składni języka Pseudo-Assembler. Wyjaśnia ona także sposób działania uproszczonego modelu komputera. Jest ona skierowana dla niedoświadczonych programistów rozpoczynających swoją przygodę z programowaniem, jako wstęp do języków niskiego poziomu takich jak języki asemblerowe lub języków średniego poziomu takich jak C.

Model Komputera

Pierwsze komputery były konstruowane w celu rozwiązywania konkretnego problemu. Dla każdego problemu należało w zasadzie zbudować inny komputer lub w najlepszym wypadku fizycznie zmodyfikować już istniejący. Zmieniło się to z momentem opracowania przez Johna von Neumanna w 1945 roku modelu komputera, w którym dane przechowywane są wspólnie z instrukcjami, co sprawia, że są kodowane w ten sam sposób najczęściej binarnie.



W architekturze tej komputer składa się z:

- pamięci komputerowej(RAM/ROM) przechowującej dane programu oraz instrukcje programu; każda komórka pamięci ma unikatowy identyfikator nazywany jej adresem
- układu sterowania pobierającego dane i instrukcje z pamięci i przetwarzający je sekwencyjnie
- arytmometru wykonującego operacje arytmetyczne i logiczne
- rejestrów czyli bardzo szybkiej pamięci podręcznej procesora; wyróżnia się rejestry:
  - 16 rejestrów ogólnego zastosowania do przechowywania liczb stało-pozycyjnych i adresów pamięci; rejestry 0-7 są przeznaczone do modyfikacji, natomiast rejestry 8-15 mają specjalne zastosowanie
  - 16 rejestrów zmiennopozycyjnych; wszystkie przeznaczone do modyfikacji
  - rejestr flag zawierający informacje o ostatnio wykonanej operacji logicznej lub arytmetycznej
  - rejestr zawierający adres obecnie wykonywanej instrukcji
- magistrali systemowej odpowiedzialnej za komunikację między komponentami komputera
- urządzeń wejścia/wyjścia służących do interakcji z użytkownikiem

„System komputerowy zbudowany w oparciu o architekturę von Neumanna powinien:

- mieć skończoną i funkcjonalnie pełną listę rozkazów
- mieć możliwość wprowadzenia programu do systemu komputerowego poprzez urządzenia zewnętrzne i jego przechowywanie w pamięci w sposób identyczny jak danych
- dane i instrukcje w takim systemie powinny być jednakowo dostępne dla procesora
- informacja jest tam przetwarzana dzięki sekwencyjnemu odczytywaniu instrukcji z pamięci komputera i wykonywaniu tych instrukcji w procesorze.

Podane warunki pozwalają przełączać system komputerowy z wykonania jednego zadania (programu) na inne bez fizycznej ingerencji w strukturę systemu, a tym samym gwarantują jego uniwersalność.

System komputerowy von Neumanna nie posiada oddzielnych pamięci do przechowywania danych i instrukcji. Instrukcje jak i dane są zakodowane w postaci liczb. Bez analizy programu trudno jest określić czy dany obszar pamięci zawiera dane czy instrukcje. Wykonywany program może się sam modyfikować traktując obszar instrukcji jako dane, a po przetworzeniu tych instrukcji – danych – zacząć je wykonywać."

PAMIĘĆ RAM/ROM

W Pseudo-Assemblerze wykorzystywany jest segmentowy model pamięci. Oznacza to że program ma wydzielone 4 segmenty: kodu, 2 x danych i stosu. Każdy segment ma 2^{16} bajtów pamięci, czyli 64kiB. „W segmencie kodu przechowywany jest kod naszego programu, w segmencie danych, jak mówi nazwa, program posiada własny obszar, gdzie przechowuje najróżniejsze dane; zaś stos jest to obszar pamięci, który zasługuje na dokładniejszy opis. Jest on opisany niżej, jeszcze w tym rozdziale.
Odwoływanie się do konkretnego adresu (w tym modelu organizacji pamięci) odbywa się przy użyciu dwóch liczb, których rozmiar wynosi w obu przypadkach $2^{16.}$ Pierwsza z nich wskazuje na adres segmentu, w którym znajduje się komórka; zaś druga liczba jest przesunięciem, czyli odległością pomiędzy adresem komórki i adresem początku segmentu, w który się znajduje. Wartość tę będziemy nazywać *offsetem*. Aby określić konkretny fizyczny adres, używa się zapisu - x:y. Gdzie x to adres segmentu, zaś y to offset, np. 003F:0004."

W architekturze Pseudo-Assemblera formą zapisu bajtów jest *Little Endian*. „Oznacza to, że wielobajtowe wartości są zapisane w kolejności od najmniej do najbardziej znaczącego (patrząc od lewej strony), bardziej znaczące bajty będą miały "wyższe" (rosnące) adresy. Należy mieć na uwadze, że odwrócona zostaje kolejność **bajtów** a nie **bitów**.

Zatem 32-bitowa wartość B3|B2|B1|B0 mogłaby by na procesorze maszyny Pseudo-Assemblera być zaprezentowana w ten sposób: B0|B1|B2|B3

Przykładowo 32-bitowa wartość 0x1BA583D4 (prefiks 0x w Pseudo-Assemblerze oznacza liczbę w systemie szesnastkowym, tak jak w C/C++) mogłaby zostać zapisana w pamięci mniej więcej tak: D4|83|A5|1B."

REJESTRY

Rejestry to bardzo szybka pamięć podręczna o czasie dostępu wielokrotnie mniejszym niż czas dostępu do pamięci RAM.

W procesorze maszyny Pseudo-Assemblera wyróżniamy następujące rejestry:

- 16 rejestrów ogólnego przeznaczenia; jak sama nazwa wskazuje, służą do wykonywania przeróżnych z góry nieokreślonych czynności takich jak: przechowywanie wyników obliczeń arytmetycznych, liczniki czy przechowywanie adresu komórki pamięci RAM; rejestry 0 - 7 są przeznaczone do modyfikacji, natomiast rejestry 8 - 15 mają specjalne zastosowanie i nie powinny być modyfikowane:
  - rejestr 8 przechowuje resztę z ostatnio wykonywanego dzielenia
  - rejestr 9 przechowuje adres pierwszego niezapisanego bajtu sekcji danych
  - rejestr 10 przechowuje adres ramki stosu
  - rejestr 11 przechowuje adres elementu znajdującego się na szczycie stosu
  - rejestr 12 przechowuje adres segmentu stosu
  - rejestr 13 przechowuje adres dodatkowej sekcji danych
  - rejestr 14 przechowuje adres sekcji danych
  - rejestr 15 przechowuje adres sekcji kodu
- 16 rejestrów zmiennopozycyjnych(0 - 15); wszystkie przeznaczone do modyfikacji służące do wykonywania obliczeń arytmetycznych na liczbach zmiennopozycyjnych.
- rejestr flag zawierający informacje o ostatnio wykonanej operacji logicznej lub arytmetycznej, takie jak czy nastąpiło przepełnienie i/lub przeniesienie, czy wynik jest ujemny albo jest zerem; wyróżnia się następujące flagi:
  - *carry flag* równy 1 gdy w wyniku dodawania/odejmowania nastąpiło przeniesienie na 32. bitu na 33. bit
  - *parity flag* równy 1 gdy liczba jedynek w najmłodszym bajcie jest parzysta
  - *zero flag* równy 1 gdy wynik ostatniego działania wyniósł 0
  - *sign flag* równy 1 gdy bit znaku jest równy 1 (liczba jest ujemna)
  - *overflow flag* równy 1 gdy w wyniku dodawania/odejmowania przekroczono możliwy zakres wartości zmiennej
- rejestr zawierający adres obecnie wykonywanej instrukcji(EIP)

STOS*

„Jak sama nazwa wskazuje, działa na zasadzie podobnej do stosu jakichś rzeczy. Elementy na stosie kładziemy "jeden na drugim". Tzn. aby nałożyć element nr. 3, najpierw kładziemy elementy 1 i 2. Gdy chcemy się pozbyć 2. elementu ze stosu, nie możemy tego zrobić dopóki nie pozbędziemy się 3.

Jednakże trzeba koniecznie wspomnieć, że stos ma specyficzną budowę. Mianowicie rośnie w dół. Zatem jeśli chcesz go sobie wyobrazić jako poukładane książki jedna na drugiej jak robiłeś do tej pory to należy wprowadzić poprawkę na te wyobrażenie. Wyobraź sobie że stos tworzysz wbrew wszelkim prawom

grawitacji na suficie. Jeśli coś dokładasz to rośnie on w dół czyli w kierunku podłogi pomieszczenia. Odwrotnie się dzieje gdy coś ze stosu zdejmujesz. W grupie rejestrów procesora Pseudo-Assemblera wyróżniamy trzy rejestry związane ze stosem: 12(SS) - przechowuje adres stosu, 11(ESP) - będący adresem wierzchołka stosu oraz 10(EBP) – nazywany wskaźnikiem ramki stosu, którego funkcję omówimy poniżej. Wywołania funkcji ingerują w specyficzny sposób w strukturę stosu. Opiszę ją poniżej.

Przed wywołaniem funkcji musimy najpierw umieścić na stosie argumenty dla niej, w odwrotnej kolejności niż są wymienione w pliku nagłówkowym/dokumentacji. Gdy już to zrobimy, mamy na stosie kilka argumentów ułożonych jeden na drugim. Następnie wywołujemy naszą funkcję. Po wywołaniu, jeszcze przed przeskokiem w miejsce pamięci (gdzie znajduje się kod naszej funkcji), automatycznie na stos odkładana jest wartość rejestru EIP (opisanego już wcześniej), która po tej operacji "leży" bezpośrednio na naszych argumentach. W następnej kolejności, na stos wrzucona zostaje zawartość rejestru 10(EBP), po czym nadana zostaje mu nowa wartość (jego rola w całym procesie za chwilę zostanie opisana). Zapamiętaną wartość określamy skrótem SFP (ang. *Stack Frame Pointer* - wyjaśnienie poniżej). Na koniec, na szczyt naszego stosu wrzucane są kolejno wszystkie zmienne lokalne wykorzystywane przez naszą funkcję i cały proces kończy się. Końcowy efekt obrazuje grafika po prawej (oczywiście ilość argumentów i lokalnych zmiennych, czy ewentualnie buforów, jest tutaj zupełnie przykładowa). Jak widać na niej, rejestr 10(EBP) wskazuje na adres SFP. Cały opisany obszar na rysunku nazywamy *ramką stosu*. Gdy funkcja kończy swoje działanie, przywracana jest wartość rejestrów EIP oraz EBP, a następnie cała ramka zostaje wyrzucona ze szczytu stosu.
Jak wspomniałem należy również pamiętać, że stos rozpina się od górnych obszarów pamięci ku dolnym. Dołożenie czegokolwiek na stos powoduje zmniejszenie wartości rejestru ESP, zaś zabranie czegoś powoduje jego zwiększenie.”

Składnia

Program napisany w Pseudo-Assemblerze składa się z ciągu wyrażeń, dzielących się na deklaracje i instrukcje. Każde wyrażenie składa się ze słów opcjonalnie poprzedzonych etykietą i/lub zakończonych komentarzem.

Komentarze

Komentarz może być dodany na końcu każdego wyrażenia. Komentarz składa się ze znaku średnika (;) i następującego po nim ciągu znaków. Komentarz obowiązuje aż do znaku nowej linii.

Etykiety

Etykieta może być umieszczona na początku wyrażenia. Składa się ona z identyfikatora i następującego po nim znaku dwukropka (:). Identyfikator może się składać ze znaków alfanumerycznych i znaku podkreślenia (_), przy czym nie może się on zaczynać cyfrą. Podczas asemblacji etykiecie nadawana jest wartość odpowiadająca adresowi instrukcji lub deklaracji, której ona dotyczy. Wiele etykiet z tym samym identyfikatorem jest niedozwolone.

SŁOWA

Wyróżnia się następujące rodzaje słów:

- słowa kluczowe
- deklaracje
- numer rejestru
- adres pamięci
- identyfikatory

Słowa kluczowe

Słowa kluczowe to kody instrukcji i deklaracji(„*opcodes*") używane żeby nadać całemu wyrażeniu znaczenie. Każde wyrażenie zaczyna się od słowa kluczowego i zawiera dokładnie jedno słowo kluczowe.

Deklaracje

Deklaracje powodują umieszczenie wskazanego rodzaju zmiennej do sekcji danych jeszcze przed uruchomieniem programu. Każdą deklarację musi poprzedzać słowo kluczowe DC lub DS. Deklaracje są jedynie dozwolone na początku każdego programu. Deklaracje, które następują po słowie kluczowym instrukcji są niedozwolone.

Numer rejestru

Numer rejestru to liczba od 0 do 15 używana jako argument instrukcji. W zależności od kontekstu wskazuje ona na rejestr ogólnego przeznaczenia lub rejestr zmiennopozycyjny.

Adres pamięci

Adresy pamięci to zapis postaci x(y), gdzie x oznacza rejestr z którego zostanie pobrany adres sekcji natomiast y oznacza przesunięcie względem tego adresu. Są używane jako argumenty funkcji. Na przykład: 11(-4) oznacza weź adres wierzchołka stosu i dodaj do niego -4. Niech adres wierzchołka stosu wynosi 0x00FF00AA. Przesunięcie wynosi 0xFFFC czyli 16-bitowa wartość -4 w systemie szesnastkowym. Żeby obliczyć rzeczywisty adres trzeba dodać do siebie 0x00AA i 0xFFFC jako 16-bitowe wartości i do wyniku dodać 0x00FF0000. Ostatecznie otrzymujemy więc adres 0x00FF00A6.

Identyfikatory

Identyfikator może się składać ze znaków alfanumerycznych i znaku podkreślenia (_), przy czym nie może się on zaczynać cyfrą. Podczas asemblacji jest on zamieniany na adres pamięci odpowiadającej etykiecie, której on dotyczy. Jego użycie jest równoważne z zastosowaniem odpowiedniego adresu pamięci, jednak zastosowanie identyfikatorów ułatwia czytanie i zrozumienie kodu.

DEKLARACJE

Wyrażenia można podzielić na dwie grupy: instrukcje i deklaracje. Program zazwyczaj składa się z ciągu deklaracji po którym występuje ciąg instrukcji. Same deklaracje składają się z kodu deklaracji i następującego po nim słowa deklaracji, a opcjonalnie może zawierać etykietę i/lub komentarz:

```
[ETYKIETA] <KOD_DEKLARACJI> <SŁOWO_DEKLARACJI> [KOMENTARZ]
```

Deklaracja w zależności od użytego kodu deklaracji może mieć dwa znaczenia:

- DC - zadeklaruj stałą, miejsce w pamięci będzie zarezerwowane, wymagane będzie podanie tej stałej w deklaracji, która zostanie umieszczona w pamięci jeszcze przed wykonaniem programu
- DS - zadeklaruj przestrzeń, miejsce w pamięci będzie zarezerwowane, ale nie zostanie tam umieszczona żadna wartość

Słowo deklaracji jest postaci:
`[LICZBA*]<TYP_ZMIENNEJ>[(WARTOŚĆ)]`

- LICZBA oznacza ile kopii zmiennej trzeba zarezerwować w pamięci, domyślnie 1 kopia jeżeli nie zostało to podane
- TYP_ZMIENNEJ oznacza jakiego typu będzie to zmienna
- WARTOŚĆ jaka zostanie wpisana do pamięci w stosownym formacie zależnie od podanego typu; jeżeli kod deklaracji to DC podanie wartości jest obowiązkowe, jeżeli DS to podanie wartości jest niedozwolone

Deklaracja liczby stało-pozycyjnej

Liczby stało-pozycyjne typu INTEGER zajmują 4 bajty w pamięci i mogą być przedstawione w następujących bazach:

- dziesiętnej; zaczynają się opcjonalnie znakiem plus (+) albo minus (-) reprezentowane są jako ciąg cyfr (0 - 9) zaczynając od niezerowej lub jako zero
- binarnej; zaczynają się prefiksem 0b reprezentowane są jako ciąg binarnych cyfr (0 - 1)
- szesnastkowej; zaczynają się prefiksem 0x reprezentowane są jako ciąg cyfr (0 - 9, A – F)

Deklaracja liczby zmiennopozycyjnej*

Liczby zmiennopozycyjne typu FLOAT zajmują 4 bajty w pamięci i mogą być przedstawione następująco:

- opcjonalny znak plus (+) albo minus (-), następująca część całkowita będąca cyframi od (0 – 9) rozpoczynająca się niezerową cyfrą lub będąca zerem, znak kropki (.) i część ułamkowa czyli zero lub więcej cyfr (0 – 9); wymagana jest obecność albo części całkowitej w postaci jednej lub więcej cyfr (0 - 9) albo kropki i części ułamkowej w postaci jednej lub więcej cyfr (0 – 9)
- binarnie; zaczynając się prefiksem 0b reprezentowane są jako ciąg binarnych cyfr (0 – 1);
- szesnastkowo; zaczynają się prefiksem 0x reprezentowane są jako ciąg cyfr (0 - 9, A – F)

Deklaracja bajtu*

Liczby typu BAJT zajmują 1 bajt w pamięci i mogą być przedstawione następująco:

- dziesiętnie; jako liczba z przedziału [0 – 255]
- binarnie; zaczynając się prefiksem 0b reprezentowane są jako ciąg binarnych cyfr (0 – 1);
- szesnastkowo; zaczynają się prefiksem 0x reprezentowane są jako ciąg cyfr (0 - 9, A – F)

Deklaracja znaku*

Liczby typu CHAR zajmują 1 bajt w pamięci i mogą być przedstawione następująco:

- dziesiętnie; jako liczba z przedziału [0 – 255]
- binarnie; zaczynając się prefiksem 0b reprezentowane są jako ciąg binarnych cyfr (0 – 1);
- szesnastkowo; zaczynają się prefiksem 0x reprezentowane są jako ciąg cyfr (0 - 9, A – F)
- znak zawarty w pojedynczych cudzysłowach (')
- sekwencja poprzedzona znakiem (\) zawarta w pojedynczych cudzysłowach ('); sekwencja może być wartością dziesiętną z przedziału [0 – 255] lub znakiem specjalnym (n/t/') oznaczającym odpowiednio nową linię znak tabulacji i pojedynczy cudzysłów

Deklaracja napisu*

Zmienne napisu typu STRING zabraniają podania liczby kopii deklarowanej wartości. Wczytane do pamięci są zakończone bajtem o wartości 0. Są przedstawiane następująco:

- ciąg znaków lub sekwencji poprzedzonych znakiem (\) zawartych w podwójnych cudzysłowach ("); sekwencja może być wartością dziesiętną z przedziału [0 – 255] lub znakiem specjalnym (n/t/') oznaczającym odpowiednio nową linię znak tabulacji i podwójny cudzysłów

INSTRUKCJE

Instrukcja to wyrażenie wykonywane podczas pracy programu. W Pseudo-Assemblerze na instrukcję mogą składać się cztery części:

- Etykieta (opcjonalnie)
- Kod instrukcji (wymagane)
- Argumenty (zależnie od instrukcji od 0 do 2)(gdy są 2 muszą być oddzielone przecinkiem (,))
- Komentarz (opcjonalnie)

```
[ETYKIETA] <KOD_INSTRUKCJI> {ARGUMENT} {,ARGUMENT} [KOMENTARZ]
```

Najważniejsze instrukcje

KOD	ARGUMENT 1	ARGUMENT 2	SKRÓCONY OPIS
INSTRUKCJE ARYTMETYCZNE			
ADD	REJESTR 1	REJESTR 2	Dodaje rejestr 1 do rejestru 2. Wynik zapisywany jest w rejestrze 1.
ADD	REJESTR 1	ADRES PAMIĘCI	Dodaje rejestr 1 do wartości pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
SUB	REJESTR 1	REJESTR 2	Odejmuje rejestr 1 od rejestru 2. Wynik zapisywany jest w rejestrze 1.
SUB	REJESTR 1	ADRES PAMIĘCI	Odejmuje rejestr 1 od wartości pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
MUL	REJESTR 1	REJESTR 2	Mnoży rejestr 1 i rejestr 2. Wynik zapisywany jest w rejestrze 1.
MUL	REJESTR 1	ADRES PAMIĘCI	Mnoży rejestr 1 i wartość pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
IDIV	REJESTR 1	REJESTR 2	Dzieli rejestr 1 przez rejestr 2. Wynik zapisywany jest w rejestrze 1. Reszta zapisywana jest w *ósmym rejestrze*. Jest to dzielenie liczb ze znakiem.
IDIV	REJESTR 1	ADRES PAMIĘCI	Dzieli rejestr 1 przez wartość pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1. Reszta zapisywana jest w *ósmym rejestrze*. Jest to dzielenie liczb ze znakiem.
DIV	REJESTR 1	REJESTR 2	Dzieli rejestr 1 przez rejestr 2. Wynik zapisywany jest w rejestrze 1. Reszta zapisywana jest w *ósmym rejestrze*. Jest to dzielenie liczb bez znaku.
DIV	REJESTR 1	ADRES PAMIĘCI	Dzieli rejestr 1 przez wartość pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.

			Reszta zapisywana jest w *ósmym rejestrze*. Jest to dzielenie liczb bez znaku.
INC	REJESTR 1	---	Dodaje jedynkę do rejestru 1. Wynik zapisywany jest w rejestrze 1.
INC	ADRES PAMIĘCI	---	Dodaje jedynkę do wartości pod wskazanym adresem. Wynik zapisywany jest pod wskazanym adresem.
DEC	REJESTR 1	---	Odejmuje jedynkę od rejestru 1. Wynik zapisywany jest w rejestrze 1.
DEC	ADRES PAMIĘCI	---	Odejmuje jedynkę od wartości pod wskazanym adresem. Wynik zapisywany jest pod wskazanym adresem.
CMP	REJESTR 1	REJESTR 2	Odejmuje rejestr 1 od rejestru 2. Wynik nie jest zapisywany.
CMP	REJESTR 1	ADRES PAMIĘCI	Odejmuje rejestr 1 od wartości pod wskazanym adresem. Wynik nie jest zapisywany.
INSTRUKCJE LOGICZNE			
AND	REJESTR 1	REJESTR 2	Koniunkcja rejestru 1 z rejestrem 2. Wynik zapisywany jest w rejestrze 1.
AND	REJESTR 1	ADRES PAMIĘCI	Koniunkcja rejestru 1 z wartością pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
OR	REJESTR 1	REJESTR 2	Alternatywa rejestru 1 i rejestrem 2. Wynik zapisywany jest w rejestrze 1.
OR	REJESTR 1	ADRES PAMIĘCI	Alternatywa rejestru 1 i wartości pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
XOR	REJESTR 1	REJESTR 2	Alternatywa wykluczająca rejestru 1 i rejestrem 2. Wynik zapisywany jest w rejestrze 1.
XOR	REJESTR 1	ADRES PAMIĘCI	Alternatywa wykluczająca rejestru 1 i wartości pod wskazanym adresem. Wynik zapisywany jest w rejestrze 1.
NOT	REJESTR 1	---	Negacja rejestru 1. Wynik zapisywany jest w rejestrze 1.
NOT	ADRES PAMIĘCI	---	Negacja wartości pod wskazanym adresem. Wynik zapisywany jest pod wskazanym adresem.
TEST	REJESTR 1	REJESTR 2	Koniunkcja rejestru 1 z rejestrem 2. Wynik nie jest zapisywany.
TEST	REJESTR 1	ADRES PAMIĘCI	Koniunkcja rejestru 1 z wartością pod wskazanym adresem. Wynik nie jest zapisywany.

SHL	REJESTR 1	REJESTR 2	Przesuwa wartość w rejestrze 1 w lewo o liczbę bitów będącą wartością w rejestrze 2. Wynik zapisywany jest w rejestrze 1.
SHR	REJESTR 1	ADRES PAMIĘCI	Przesuwa wartość w rejestrze 1 w prawo o liczbę bitów będącą wartością w rejestrze 2. Wynik zapisywany jest w rejestrze 1.
INSTRUKCJE SKOKU			
JMP	ADRES PAMIĘCI	---	Skacze bezwarunkowo pod wskazany adres pamięci.
JE/JZ	ADRES PAMIĘCI	---	Skacze pod wskazany adres pamięci jeżeli wynikiem ostatniego działania było zero.
JG	ADRES PAMIĘCI	---	Skacze pod wskazany adres pamięci jeżeli wynikiem ostatniego działania była liczba większa od zera.
JL	ADRES PAMIĘCI	---	Skacze pod wskazany adres pamięci jeżeli wynikiem ostatniego działania była liczba mniejsza od zera.
LOOP	REJESTR 1	ADRES PAMIĘCI	Odejmuje jedynkę od rejestru 1. Skacze pod wskazany adres pamięci jeżeli wynikiem nie było zero.
CALL	ADRES PAMIĘCI	---	Odkłada na stos adres następnej instrukcji. Skacze bezwarunkowo pod wskazany adres pamięci.
RET	---	---	Zdejmuje ze stosu adres. Skacze pod ten adres.
INSTRUKCJE PRZESYŁU DANYCH			
LD	REJESTR 1	REJESTR 2	Zapisuje wartość rejestru 2 w rejestrze 1.
LD	REJESTR 1	ADRES PAMIĘCI	Zapisuje wartość pod wskazanym adresem pamięci w rejestrze 1.
LDB	REJESTR 1	ADRES PAMIĘCI	Zapisuje bajt pod wskazanym adresem pamięci w rejestrze 1.
ST	REJESTR 1	ADRES PAMIĘCI	Zapisuje wartość rejestru 1 pod wskazanym adresem pamięci.
LDA	REJESTR 1	ADRES PAMIĘCI	Zapisuje wskazany adres pamięci w rejestrze 1.
XCHG	REJESTR 1	REJESTR 2	Zamienia miejscami wartości rejestru 1 i rejestru 2.
XCHG	REJESTR 1	ADRES PAMIĘCI	Zamienia miejscami wartości rejestru 1 i wartości pod wskazanym adresem pamięci.
PUSH	REJESTR 1	---	Wrzuca wartość rejestru 1 na szczyt stosu.
PUSH	ADRES PAMIĘCI	---	Wrzuca wartość rejestru 1 na szczyt stosu.
POP	REJESTR 1	---	Zdejmuje wartość ze stosu i zapisuje ją w rejestrze 1.
POP	ADRES PAMIĘCI	---	Zdejmuje wartość ze stosu i zapisuje ją pod wskazanym adresem pamięci.

INSTRUKCJE WEJŚCIA/WYJŚCIA			
IN	REJESTR 1	---	Wczytuje znak z klawiatury. Zapisuje go w rejestrze 1.
IN	ADRES PAMIĘCI	---	Wczytuje znak z klawiatury. Zapisuje go pod wskazanym adresem.
OUT	REJESTR 1	---	Wypisuje wartość rejestru 1 na ekran jako wartość bez znaku.
OUT	ADRES PAMIĘCI	---	Wypisuje wartość spod wskazanego adresu pamięci na ekran jako wartość bez znaku.
IOUT	REJESTR 1	---	Wypisuje wartość rejestru 1 na ekran jako wartość ze znakiem.
IOUT	ADRES PAMIĘCI	---	Wypisuje wartość spod wskazanego adresu pamięci na ekran jako wartość ze znakiem.
COUT	REJESTR 1	---	Wypisuje wartość rejestru 1 na ekran jako znak.
COUT	ADRES PAMIĘCI	---	Wypisuje wartość spod wskazanego adresu pamięci na ekran jako znak.
INNE INSTRUKCJE			
RAND	REJESTR 1	---	Losuje wartość i zapisuje ją w rejestrze 1.
RAND	ADRES PAMIĘCI	---	Losuje wartość i zapisuje ją pod wskazanym adresem.
NOP	---	---	Nic nie robi.
EXIT	---	---	Kończy działanie komputera.

Przykładowe programy

Program 1. Napisać program do wyznaczania liczby rozwiązań równania kwadratowego $2x^2 - 8x + 8 = 0$.

```
;quadratic_equation.asm
;PROGRAM DO WYZNACZANIA LICZBY ROZWIĄZAŃ RÓWNANIA KWADRATOWEGO 2x^2-8x+8=0
A:          DC      INTEGER(2)
B:          DC      INTEGER(-8)
C:          DC      INTEGER(8)
WYNIK:      DS      INTEGER
ZERO:       DC      INTEGER(0)
JEDEN:      DC      INTEGER(1)
DWA:        DC      INTEGER(2)
CZTERY:     DC      INTEGER(4)

MAIN:
        LD      0, B                ;REJESTR 0 -> B
        MUL     0, 0                ;REJESTR 0 -> B * B
        LD      1, CZTERY           ;REJESTR 1 -> CZTERY
        MUL     1, A                ;REJESTR 1 -> CZTERY * A
        MUL     1, C                ;REJESTR 1 -> CZTERY * A * C
        SUB     0, 1                ;REJESTR 0 -> B * B - CZTERY * A * C
        JG      DWA_ROZWIAZANIA     ;B * B - CZTERY * A * C >? 0
        JZ      JEDNO_ROZWIAZANIE   ;B * B - CZTERY * A * C =? 0
        JL      ZERO_ROZWIAZAN      ;B * B - CZTERY * A * C <? 0

DWA_ROZWIAZANIA:                    ;B * B - CZTERY * A * C > 0
        LD      2, DWA              ;REJESTR 2 -> DWA
        ST      2, WYNIK            ;ZAPISZ W PAMIĘCI WYNIK
        JMP     KONIEC_PROGRAMU
JEDNO_ROZWIAZANIE:                  ;B * B - CZTERY * A * C = 0
        LD      2, JEDEN            ;REJESTR 2 -> JEDEN
        ST      2, WYNIK            ;ZAPISZ W PAMIĘCI WYNIK
        JMP     KONIEC_PROGRAMU
ZERO_ROZWIAZAN:                     ;B * B - CZTERY * A * C < 0
        LD      2, ZERO             ;REJESTR 2 -> ZERO
        ST      2, WYNIK            ;ZAPISZ W PAMIĘCI WYNIK
        JMP     KONIEC_PROGRAMU

KONIEC_PROGRAMU:
        OUT     WYNIK               ;WYPISZ WYNIK NA EKRAN
        EXIT                        ;KONIEC
```

Wynik programu: 1

Program 2. Napisać program do wyznaczania sumy dodatnich liczb ze 100-elementowego wektora.

```
;vector_sum.asm
;PROGRAM DO WYZNACZANIA SUMY DODATNICH LICZB ZE 100-ELEMENTOWEGO WEKTORA
JEDEN:      DC      INTEGER(1)
STO:        DC      INTEGER(100)
WEKTOR:     DS      100*INTEGER
SUMA:       DS      INTEGER

MAIN:
        CALL    INIT_TAB        ;ZAINICJALIZUJ TABLICĘ
        SUB     0, 0            ;REJESTR 0 - WARTOŚĆ ZERO
        LD      1, 0            ;REJESTR 1 - TU OBLICZAMY SUMĘ
        LD      2, 0            ;REJESTR 2 - ITERATOR
        LDA     3, WEKTOR       ;REJESTR 3 - WSKAŹNIK NA OBECNIE
ANALIZOWANY ELEMENT WEKTORA

LOOP:   CMP     0, 3(0)         ;ZERO ? I-TY ELEMENT WEKTORA
        JGE     NEGATIVE        ;JEŚLI ZERO JEST WIĘKSZE LUB RÓWNE I-TEMU
ELEMENTOWI WEKTORA TO NIE DODAWAJ DO SUMY
        ADD     1, 3(0)         ;W.P.P. DODAWAJ
NEGATIVE:
        LDA     3, 3(4)         ;PRZESUŃ WSKAŹNIK WEKTORA NA NASTĘPNY
ELEMENT
        ADD     2, JEDEN        ;ZWIĘKSZ ITERATOR O JEDEN
        CMP     2, STO          ;ITERATOR ? STO
        JZ      KONIEC_PROGRAMU ;JEŚLI ITERATOR RÓWNY STO TO SKOŃCZYL SIĘ
WEKTOR
        JMP     LOOP            ;W.P.P. PRZEJDŹ DO NASTĘPNEGO ELEMENTU
WEKTORA
KONIEC_PROGRAMU:
        ST      1, SUMA         ;ZAPISZ WYNIK W PAMIĘCI
        OUT     SUMA            ;WYPISZ WYNIK NA EKRAN
        EXIT                    ;KONIEC PROGRAMU


;INICJALIZUJE WEKTOR LOSOWYMI WARTOŚCIAMI Z PRZEDZIAŁU <-100, 99>
INIT_TAB:
...                             ;PEWNE INSTRUKCJE WYPEŁNIAJĄCE WEKTOR
...                             ;LOSOWYMI WARTOŚCIAMI Z PRZEDZIAŁU
...                             ;<-100, 99>
...
...
...
        RET
```

Wynik programu: 2542

Kod Maszynowy

„Kod maszynowy to postać programu komputerowego (zwana postacią wykonywalną lub binarną) przeznaczona do bezpośredniego lub prawie bezpośredniego wykonania przez procesor. Jest ona dopasowana do konkretnego typu procesora i wyrażona w postaci rozumianych przez niego kodów rozkazów i ich argumentów. Kod maszynowy może być generowany w procesie kompilacji (w przypadku języków wysokiego poziomu) lub asemblacji (w przypadku języków niskiego poziomu).”

Asemblacja

Kod maszynowy Pseudo-Assemblera jest podzielony na dwie części. W pierwszej części zawarte są informacje o zadeklarowanych zmiennych. Ta część kodu nie jest wykonywana. Jest to tylko informacja dla komputera, gdzie, w jakiej kolejności i jakie wartości mają być zadeklarowane. Druga część składa się z instrukcji i ich argumentów. Ta część jest przeznaczona do bezpośredniego wykonania przez procesor. Instrukcje w tej części umieszczone są w postaci binarnej jedna po drugiej. Instrukcje dotyczące odwołania do pamięci zajmują cztery bajty każda, natomiast wszystkie pozostałe po dwa bajty.

Struktura Pliku Binarnego*

Każdy plik binarny Pseudo-Assemblera zaczyna się 16-bitową liczbą określającą liczbę deklaracji. Następują po niej deklaracje zakodowane binarnie. Każda deklaracja zaczyna się bajtem określającym jej rodzaj. Następna jest 16-bitowa liczba określająca ile kopii zmiennej należy zadeklarować, o ile rodzaj deklaracji tego wymaga. Ostatnią częścią deklaracji jest wartość która ma być zadeklarowana w postaci binarnej, również opcjonalna w zależności od rodzaju deklaracji. Po wyczerpaniu puli deklaracji rozpoczyna się sekcja instrukcji. Każda instrukcja rozpoczyna się 1-bajtowym kodem, na podstawie którego można określić co ta instrukcja robi oraz czy odwołuje się ona do pamięci RAM. Następny bajt to dwa numery rejestrów. Jeżeli instrukcja odwołuje się do pamięci, to dwa kolejne bajty są *offsetem.* Jeżeli więc funkcja nie odwołuje się do pamięci to dwa rejestry są argumentami funkcji, a jeżeli odwołuje się do pamięci to argumentami są pierwszy rejestr oraz wartość przechowywana pod adresem w drugim rejestrze przesunięta o *offset* lub sam adres. Koniec sekcji instrukcji jest równocześnie końcem pliku binarnego. Należy pamiętać że dla pliku binarnego formą zapisu bajtów jest również *Little Endian.*

Przykładowy Plik Binarny*

Struktura **programu 1**:

```
     | 08 00           |  8 DEKLARACJI:
0000 | 01 02 00 00 00  |  A:          DC      INTEGER(2)
0004 | 01 F8 FF FF FF  |  B:          DC      INTEGER(-8)
0008 | 01 08 00 00 00  |  C:          DC      INTEGER(8)
000C | 06              |  WYNIK:      DS      INTEGER
0010 | 01 00 00 00 00  |  ZERO:       DC      INTEGER(0)
0014 | 01 01 00 00 00  |  JEDEN:      DC      INTEGER(1)
0018 | 01 02 00 00 00  |  DWA:        DC      INTEGER(2)
001C | 01 04 00 00 00  |  CZTERY:     DC      INTEGER(4)
---- |                 |  MAIN:
0000 | FF 0E 04 00     |              LD      0, B
0004 | 1F 00           |              MUL     0, 0
0006 | FF 1E 1C 00     |              LD      1, CZTERY
000A | E6 1E 00 00     |              MUL     1, A
000E | E6 1E 08 00     |              MUL     1, C
0012 | 1E 01           |              SUB     0, 1
0014 | CA 0F 20 00     |              JG      DWA_ROZWIAZANIA
0018 | CC 0F 2C 00     |              JZ      JEDNO_ROZWIAZANIE
001C | C8 0F 38 00     |              JL      ZERO_ROZWIAZAN
     |                 |  DWA_ROZWIAZANIA:
0020 | FF 2E 18 00     |              LD      2, DWA
0024 | FB 2E 0C 00     |              ST      2, WYNIK
0028 | CD 0F 44 00     |              JMP     KONIEC_PROGRAMU
     |                 |  JEDNO_ROZWIAZANIE:
002C | FF 2E 14 00     |              LD      2, JEDEN
0030 | FB 2E 0C 00     |              ST      2, WYNIK
0034 | CD 0F 44 00     |              JMP     KONIEC_PROGRAMU
     |                 |  ZERO_ROZWIAZAN:
0038 | FF 2E 10 00     |              LD      2, ZERO
003C | FB 2E 0C 00     |              ST      2, WYNIK
0040 | CD 0F 44 00     |              JMP     KONIEC_PROGRAMU
     |                 |  KONIEC_PROGRAMU:
0044 | F7 0E 0C 00     |              OUT     WYNIK
0048 | 00 00           |              EXIT
```

Po lewej stronie zaznaczono w którym miejscu w sekcji danych albo kodu znajdowała by się zadeklarowana zmienna albo instrukcja. Odpowiednimi kolorami zaznaczono powiązania z kodem znajdującym się po prawej stronie. Jak można zauważyć odwołania do pamięci składają się z odpowiedniego rejestru(E - sekcja danych, F - sekcja kodu) i odpowiedniego przesunięcia. Na przykład etykieta CZTERY odwołuje się do sekcji danych (E) z przesunięciem 001C, natomiast etykieta JEDNO_ROZWIAZANIE odwołuje się do sekcji kodu (F) z przesunięciem 002C.

Opisy Poszczególnych Instrukcji

Ten rozdział przeznaczony jest szczegółowemu opisowi wszystkich dostępnych instrukcji
z uwzględnieniem najdrobniejszych szczegółów dotyczących ich działania, czy binarnego kodu instrukcji.

Opis podzielony jest na cztery sekcje:

- tabela wymieniająca warianty instrukcji wraz z kodami instrukcji, możliwą listą argumentów oraz skróconym opisem
- dokładny opis działania instrukcji
- poglądowy algorytm wyjaśniający działanie instrukcji
- flagi na które oddziaływa dana instrukcja lub nie

Każdej instrukcji dedykowana jest jedna strona, a instrukcje posortowane są alfabetycznie celem ułatwienia szukania danej instrukcji.

ADD

KOD	INSTRUKCJA	SKRÓCONY OPIS
1D	ADD *reg1, reg2*	Dodaje reg1 do reg2.
E8	ADD *reg1, mem2*	Dodaje reg1 do mem2.

Opis

Dodaje pierwszy argument(wynikowy) do drugiego argumentu(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci.

Instrukcja ADD wykonuje dodawanie całkowitoliczbowe. Wykonuje obliczenia zarówno dla liczb całkowitoliczbowych ze znakiem jak i bez znaku i ustawia flagi CF oraz OF żeby zasygnalizować przeniesienie (przepełnienie) odpowiednio w znakowym jak i bez znakowym wyniku. Flaga SF sygnalizuje znak wyniku.

Operacja

ARG1 = ARG1 + ARG2;

Modyfikowane Flagi

Flagi OF, SF, ZF, CF i PF są ustawione w zależności od wyniku.

AND

KOD	INSTRUKCJA	SKRÓCONY OPIS
32	AND *reg1, reg2*	Koniunkcja reg1 i reg2.
D8	AND *reg1, mem2*	Koniunkcja reg1 i mem2.

Opis

Wykonuje bitową koniunkcję pierwszego argumentu(wynikowy) i drugiego argumentu(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Każdy bit wyniku ustawiany jest wartością 1 jeżeli w obu wynikowym i źródłowym argumencie odpowiedni bit miał wartość 1; w przeciwnym przypadku ustawiany jest wartością 0.

Operacja

ARG1 = ARG1 and ARG2;

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

BOUT

KOD	INSTRUKCJA	SKRÓCONY OPIS
08	BOUT *reg1*	Wypisuje reg1 na ekran jako bajt.
F3	BOUT *mem1*	Wypisuje mem1 na ekran jako bajt.

Opis

Wypisuje najmłodszy bajt argumentu na ekran. Argumentem może być rejestr albo miejsce w pamięci.

Operacja

OUT (BYTE)ARG1;

Modyfikowane Flagi

Żadne.

CALL

KOD	INSTRUKCJA	SKRÓCONY OPIS
BB	CALL *mem1*	Wywołuje procedurę spod podanego miejsca w pamięci mem1.

Opis

Zapisuje adres następnej po tej instrukcji(rejestr EIP) na stosie(dla późniejszego powrotu instrukcją RET), następnie skacze w miejsce wskazane w argumencie źródłowym, wskazującym na pierwszą instrukcję procedury. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

STACK.PUSH(EIP);

EIP = ADDRESS(ARG1);

Modyfikowane Flagi

Żadne.

CMP

KOD	INSTRUKCJA	SKRÓCONY OPIS
22	CMP *reg1, reg2*	Porównuje reg1 z reg2.
E3	CMP *reg1, mem2*	Porównuje reg1 z mem2.

Opis

Porównuje pierwszy źródłowy argument z drugim źródłowym argumentem i ustawia flagi w zależności od wyniku porównania. Porównanie jest wykonywane przez odejmowanie drugiego argumentu od pierwszego w podobny sposób jak w instrukcji SUB, jednak wynik nie jest nigdzie zapisywany. Pierwszy argument może być wyłącznie rejestrem. Drugi argument może być rejestrem albo miejscem w pamięci.

Warunki w instrukcjach skoku odnoszą się do wyniku porównania instrukcji CMP.

Operacja

TEMP = ARG1 – ARG2;
MODIFY_FLAGS;

Modyfikowane Flagi

Flagi OF, SF, ZF, CF i PF są ustawione w zależności od wyniku.

COUT

KOD	INSTRUKCJA	SKRÓCONY OPIS
07	COUT *reg1*	Wypisuje reg1 na ekran jako znak.
F4	COUT *mem1*	Wypisuje mem1 na ekran jako znak.

Opis

Wypisuje najmłodszy bajt argumentu na ekran jako znak. Używane jest kodowanie ASCII. Argumentem może być rejestr albo miejsce w pamięci.

Operacja

OUT (CHAR)ARG1

Modyfikowane Flagi

Żadne.

DEC

KOD	INSTRUKCJA	SKRÓCONY OPIS
24	DEC *reg1*	Zmniejsza reg1 o jeden.
E1	DEC *mem1*	Zmniejsza mem1 o jeden.

Opis

Odejmuje jeden od argumentu(wynikowy), jednocześnie zachowując stan flagi CF. Wynikowy argument może być rejestrem albo miejscem w pamięci. Instrukcja pozwala zmniejszenie licznika pętli bez modyfikacji flagi CF.(Żeby zmniejszyć licznik z modyfikacją flagi CF należy użyć odpowiednio instrukcji SUB)

Operacja

ARG1 = ARG1 -1;

Modyfikowane Flagi

Flaga CF nie jest modyfikowana. Flagi OF, SF, ZF i PF są ustawione w zależności od wyniku.

DIV

KOD	INSTRUKCJA	SKRÓCONY OPIS
20	`DIV` *`reg1, reg2`*	Dzieli bez znaku reg1 przez reg2.
E5	`DIV` *`reg1, mem2`*	Dzieli bez znaku reg1 przez mem2.

Opis

Dzieli bez znaku pierwszy argument(wynikowy) przez drugi argument(źródłowy) a następnie przechowuje wynik w argumencie wynikowym, reszta z dzielenia przechowywana jest w rejestrze ósmym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci.

Niecałkowita część wyniku jest przycinana do zera. Reszta z dzielenia jest zawsze mniejsza od dzielnika. Błąd dzielenia jest sygnalizowany przez ustawienie flag CF i OF na 1, przy braku błędu flagi te ustawiane są na 0.

Operacja

ARG1 = ARG1 / ARG2;

REG8 = ARG1 mod ARG2;

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 1, gdy dzielnik to 0, w przeciwnym przypadku ustawiane na 0. Flagi SF, ZF i PF nie są modyfikowane.

ENTER

KOD	INSTRUKCJA	SKRÓCONY OPIS
17	ENTER	Tworzy ramkę stosu dla procedury.

Opis

Tworzy ramkę stosu dla procedury. Instrukcje ENTER i LEAVE są przeznaczone do organizacji kodu w bloki podobnie do języków wyższego poziomu. Typowo instrukcja ENTER jest pierwszą instrukcją procedury, tworzącą dla niej nową ramkę stosu. Instrukcja LEAVE jest zwykle używane na końcu procedury(tuż przed instrukcją RET), żeby zwolnić stworzoną ramkę.

Operacja

STACK.PUSH(EBP);
EBP = ESP;

Modyfikowane Flagi

Żadne.

EXIT

KOD	INSTRUKCJA	SKRÓCONY OPIS
00	EXIT	Kończy działanie komputera.

Opis

Kończy działanie komputera.

Operacja

Modyfikowane Flagi

Żadne.

FABS

KOD	INSTRUKCJA	SKRÓCONY OPIS
2C	FABS *fpr1*	Oblicza wartość bezwzględną z fpr1.

Opis

Zeruje bit znaku pierwszego argumentu tworząc tym samym wartość bezwzględną. Argumentem może być wyłącznie rejestr zmiennoprzeicnkowy.

Operacja

ARG1 = |ARG1|;

Modyfikowane Flagi

Żadne.

FADD

KOD	INSTRUKCJA	SKRÓCONY OPIS
26	FADD *fpr1, fpr2*	Dodaje fpr1 do fpr2.
DF	FADD *fpr1, mem2*	Dodaje fpr1 do mem2.

Opis

Dodaje pierwszy argument(wynikowy) do drugiego argumentu(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = ARG1 + ARG2;

Modyfikowane Flagi

Żadne.

FCMP

KOD	INSTRUKCJA	SKRÓCONY OPIS
2A	FCMP *fpr1, fpr2*	Porównuje fpr1 z fpr2.
DB	FCMP *fpr1, mem2*	Porównuje fpr1 z mem2.

Opis

Porównuje pierwszy argument źródłowy z drugim i ustawia flagi ZF, CF i PF w zależności od wyniku (patrz tabela poniżej). Flagi ZF i CF są ustawiane podobnie jak gdyby użyta była instrukcja CMP dla liczb całkowitych bez znaku (Instrukcje JA, JE i JB powinny zostać użyte do sprawdzenia warunku). Flaga CF jest ustawiana, gdy jeden z argumentów nie jest liczbą. Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci. Znak 0.0 jest ignorowany, więc +0.0 jest równe -0.0.

Warunek	ZF	CF	PF
ARG1 > ARG2	0	0	0
ARG1 < ARG2	0	1	0
ARG1 = ARG2	1	0	0
ARG1 ? ARG2	1	1	1

Operacja

```
CASE(RELATION)
        ARG1 > ARG2:  ZF=0, CF=0, PF=0;
        ARG1 < ARG2:  ZF=0, CF=1, PF=0;
        ARG1 = ARG2:  ZF=1, CF=0, PF=0;
        DEFAULT:      ZF=1, CF=1, PF=1;
```

Modyfikowane Flagi

Flagi OF, SF nie są modyfikowane. Dla flag ZF, CF, PF patrz tabela wyżej.

FCOS

KOD	INSTRUKCJA	SKRÓCONY OPIS
2E	FCOS *fpr1*	Oblicza cosinus z fpr1.

Opis

Oblicza przybliżoną wartość cosinusa kąta podanego argumentu. Argument musi być podany w radianach i może być wyłącznie rejestrem zmiennoprzecinkowym.

Operacja

ARG1 = COS(ARG1);

Modyfikowane Flagi

Żadne.

FDIV

KOD	INSTRUKCJA	SKRÓCONY OPIS
29	`FDIV` *`fpr1, fpr2`*	Dzieli fpr1 przez fpr2.
DC	`FDIV` *`fpr1, mem2`*	Dzieli fpr1 przez mem2.

Opis

Dzieli pierwszy argument(wynikowy) przez drugi argument(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = ARG1 / ARG2;

Modyfikowane Flagi

Żadne.

FILD

KOD	INSTRUKCJA	SKRÓCONY OPIS
ED	`FILD` *`reg1, mem2`*	Wczytuje liczbę całkowitą mem2 jako zmiennoprzecinkową do fpr1.

Opis

Wczytuje liczbę całkowitą podaną w drugim argumencie(źródłowy), konwertuje ją do liczby zmiennoprzecinkowej i umieszcza ją w rejestrze zmiennoprzecinkowym podanym w drugim argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być wyłącznie miejscem w pamięci.

Operacja

ARG1 = (FLOAT)ARG2

Modyfikowane Flagi

Żadne.

FIST

KOD	INSTRUKCJA	SKRÓCONY OPIS
EE	`FIST` *`fpr1, mem2`*	Umieszcza liczbę zmiennoprzecinkową fpr1 jako całkowitą w mem2.

Opis

Konwertuje liczbę zmiennoprzecinkową podaną w pierwszym argumencie(źródłowy) na liczbę całkowitą i umieszcza ją w drugim argumencie(wynikowy). Źródłowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Wynikowy argument może być wyłącznie miejscem w pamięci.

Jeżeli źródłowy argument nie jest liczbą całkowitą, to wartość niecałkowita jest ucinana.

Operacja

ARG2 = (INT)ARG1

Modyfikowane Flagi

Żadne.

FLD

KOD	INSTRUKCJA	SKRÓCONY OPIS
03	FLD *fpr1, fpr2*	Wczytuje fpr2 do fpr1.
FC	FLD *fpr1, mem2*	Wczytuje mem2 do fpr1.

Opis

Wczytuje liczbę podaną w drugim argumencie(źródłowy) i umieszcza ją w rejestrze zmiennoprzecinkowym podanym w drugim argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = ARG2;

Modyfikowane Flagi

Żadne.

FMUL

KOD	INSTRUKCJA	SKRÓCONY OPIS
28	MUL *fpr1, fpr2*	Mnoży fpr1 przez fpr2.
DD	MUL *fpr1, mem2*	Mnoży fpr1 przez mem2.

Opis

Mnoży pierwszy argument(wynikowy) przez drugi argument(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = ARG1 · ARG2;

Modyfikowane Flagi

Żadne.

FOUT

KOD	INSTRUKCJA	SKRÓCONY OPIS
06	FOUT *fpr1*	Wypisuje fpr1 na ekran jako liczbę zmiennoprzecinkową.
F5	FOUT *mem1*	Wypisuje mem1 na ekran jako liczb zmiennoprzecinkową.

Opis

Wypisuje argument na ekran jako liczbę zmiennoprzecinkową. Argumentem może być rejestr zmiennoprzecinkowy albo miejsce w pamięci.

Operacja

OUT (FLOAT)ARG1

Modyfikowane Flagi

Żadne.

FPOP

KOD	INSTRUKCJA	SKRÓCONY OPIS
0F	FPOP *fpr1*	Zdejmuje wartość ze stosu i umieszcza ją w fpr1.
F1	FPOP *mem1*	Zdejmuje wartość ze stosu i umieszcza ją w mem1.

Opis

Zdejmuje wartość ze stosu i umieszcza ją w argumencie wynikowym. Wynikowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = STACK.POP();

Modyfikowane Flagi

Żadne.

FPOPA

KOD	INSTRUKCJA	SKRÓCONY OPIS
12	FPOPA	Zdejmuje ze stosu wszystkie rejestry zmiennoprzecinkowe.

Opis

Po kolei zdejmuje ze stosu i umieszcza w rejestrach zmiennoprzecinkowych liczby w kolejności 15, 14, ..., 2, 1, 0.

Operacja

FPR15 = STACK.POP();

FPR14 = STACK.POP();

...

FPR2 = STACK.POP();

FPR1 = STACK.POP();

FPR0 = STACK.POP();

Modyfikowane Flagi

Żadne.

FPUSH

KOD	INSTRUKCJA	SKRÓCONY OPIS
0A	FPUSH *fpr1*	Wrzuca fpr1 na stos.
F2	FPUSH *mem1*	Wrzuca mem1 na stos.

Opis

Wrzuca na stos argumentu źródłowy. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

STACK.PUSH(ARG1);

Modyfikowane Flagi

Żadne.

FPUSHA

KOD	INSTRUKCJA	SKRÓCONY OPIS
0D	FPUSHA	Wrzuca wszystkie rejestry zmiennoprzecinkowe na stos.

Opis

Wrzuca wszystkie rejestry zmiennoprzecinkowe na stos w kolejności 0, 1, 2, ..., 14, 15.

Operacja

STACK.PUSH(FPR0);
STACK.PUSH(FPR1);
STACK.PUSH(FPR2);
...
STACK.PUSH(FPR14);
STACK.PUSH(FPR15);

Modyfikowane Flagi

Żadne.

FSIN

KOD	INSTRUKCJA	SKRÓCONY OPIS
2D	`SIN` *`fpr1`*	Oblicza sinus z fpr1.

Opis

Oblicza przybliżoną wartość sinusa kąta podanego argumentu. Argument musi być podany w radianach i może być wyłącznie rejestrem zmiennoprzecinkowym.

Operacja

ARG1 = SIN(ARG1);

Modyfikowane Flagi

Żadne.

FSQRT

KOD	INSTRUKCJA	SKRÓCONY OPIS
2B	FSQRT *fpr1*	Oblicza pierwiastek kwadratowy z fpr1.

Opis

Oblicza wartość pierwiastka kwadratowego podanego argumentu. Argument może być wyłącznie rejestrem zmiennoprzecinkowym.

Operacja

ARG1 = SQRT(ARG1);

Modyfikowane Flagi

Żadne.

FST

KOD	INSTRUKCJA	SKRÓCONY OPIS
F9	FST *fpr1, mem2*	Umieszcza fpr1 w mem2.

Opis

Umieszcza liczbę zmiennoprzecinkową podaną w pierwszym argumencie(źródłowy) w drugim argumencie(wynikowy). Źródłowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Wynikowy argument może być wyłącznie miejscem w pamięci.

Operacja

ARG2 = ARG1;

Modyfikowane Flagi

Żadne.

FSUB

KOD	INSTRUKCJA	SKRÓCONY OPIS
27	FSUB *fpr1, fpr2*	Odejmuje fpr2 od fpr1.
DE	FSUB *fpr1, mem2*	Odejmuje mem2 od fpr1.

Opis

Odejmuje drugi argument(źródłowy) do pierwszego argumentu(wynikowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

ARG1 = ARG1 - ARG2;

Modyfikowane Flagi

Żadne.

FTAN

KOD	INSTRUKCJA	SKRÓCONY OPIS
2F	FTAN *fpr1*	Oblicza tangens z fpr1

Opis

Oblicza przybliżoną wartość tangensa kąta podanego argumentu. Argument musi być podany w radianach i może być wyłącznie rejestrem zmiennoprzecinkowym.

Operacja

ARG1 = TAN(ARG1);

Modyfikowane Flagi

Żadne.

FTST

KOD	INSTRUKCJA	SKRÓCONY OPIS
31	FTST *fpr1*	Porównuje fpr1 z 0.0.
D9	FTST *mem1*	Porównuje mem1 z 0.0.

Opis

Porównuje pierwszy argument źródłowy z wartością 0.0 i ustawia flagi ZF, CF i PF w zależności od wyniku (patrz tabela poniżej). Flagi ZF i CF są ustawiane podobnie jak gdyby użyta była instrukcja CMP dla liczb całkowitych bez znaku (Instrukcje JA, JE i JB powinny zostać użyte do sprawdzenia warunku). Flaga CF jest ustawiana, gdy argument nie jest liczbą. Źródłowy argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci. Znak 0.0 jest ignorowany, więc +0.0 jest równe -0.0.

Warunek	ZF	CF	PF
ARG1 > 0.0	0	0	0
ARG1 < 0.0	0	1	0
ARG1 = 0.0	1	0	0
ARG1 ? 0.0	1	1	1

Operacja

```
CASE(RELATION)
        ARG1 > 0.0:     ZF=0, CF=0, PF=0;
        ARG1 < 0.0:     ZF=0, CF=1, PF=0;
        ARG1 = 0.0:     ZF=1, CF=0, PF=0;
        DEFAULT:        ZF=1, CF=1, PF=1;
ESAC;
```

Modyfikowane Flagi

Flagi OF, SF nie są modyfikowane. Dla flag ZF, CF, PF patrz tabela wyżej.

FXAM

KOD	INSTRUKCJA	SKRÓCONY OPIS
30	FXAM *fpr1*	Klasyfikuje wartość fpr1.
DA	FXAM *mem1*	Klasyfikuje wartość mem1.

Opis

Bada zawartość argumentu źródłowego i ustawia odpowiednio flagi ZF, CF i PF. Ustawienie flag odpowiada odpowiednim klasom(patrz tabela niżej). Flaga SF jest zawsze ustawiana względem bitu znaku liczby. Argumentem może być rejestr zmiennoprzecinkowy lub miejsce w pamięci.

Klasa	ZF	CF	PF
Nieskończoność	0	1	1
Nie-liczba	0	1	0
Zero	1	0	0
Nieznormalizowana	1	0	1
Zwykła wartość	0	0	1
Niewspierana wartość	0	0	0

Operacja

```
CASE(CLASS)
        INFINITY:       ZF=0, CF=1, PF=1;
        NAN:            ZF=0, CF=1, PF=0;
        ZERO:           ZF=1, CF=0, PF=0;
        SUBNORMAL:      ZF=1, CF=0, PF=1;
        FINITE:         ZF=0, CF=0, PF=1;
        DEFAULT:        ZF=0, CF=0, PF=0;
ESAC;
SF = 0x80000000 & ARG1;
```

Modyfikowane Flagi

Flaga OF, nie jest modyfikowana. Flaga SF jest ustawiana w zależności od znaku. Dla flag ZF, CF, PF patrz tabela wyżej.

FXCH

KOD	INSTRUKCJA	SKRÓCONY OPIS
14	FXCH *fpr1, fpr2*	Zamienia wartości między fpr1 a fpr2.
EF	FXCH *fpr1, mem2*	Zamienia wartości między fpr1 a mem2.

Opis

Zamienia miejscami wartości między pierwszym argumentem źródłowym a drugim argumentem. Pierwszy argument może być wyłącznie rejestrem zmiennoprzecinkowym. Drugi argument może być rejestrem zmiennoprzecinkowym albo miejscem w pamięci.

Operacja

TEMP = ARG1;

ARG1 = ARG2;

ARG2 = TEMP;

Modyfikowane Flagi

Żadne.

HALT

KOD	INSTRUKCJA	SKRÓCONY OPIS
1A	HALT	Wstrzymuje pracę komputera.

Opis

Wstrzymuje pracę komputera. Praca jest wznawiana wraz z naciśnięciem klawisza RETURN.

Operacja

HALT;

Modyfikowane Flagi

Żadne.

IDIV

KOD	INSTRUKCJA	SKRÓCONY OPIS
21	`IDIV` *`reg1, reg2`*	Dzieli reg1 przez reg2.
E4	`IDIV` *`reg1, mem2`*	Dzieli reg1 przez mem2.

Opis

Dzieli ze znakiem pierwszy argument(wynikowy) przez drugi argument(źródłowy) a następnie przechowuje wynik w argumencie wynikowym, reszta z dzielenia przechowywana jest w rejestrze ósmym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci.

Niecałkowita część wyniku jest przycinana do zera. Reszta z dzielenia jest zawsze mniejsza od dzielnika. Błąd dzielenia jest sygnalizowany przez ustawienie flag CF i OF na 1, przy braku błędu flagi te ustawiane są na 0.

Operacja

ARG1 = ARG1 / ARG2;

REG8 = ARG1 mod ARG2;

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 1, gdy dzielnik to 0, w przeciwnym przypadku ustawiane na 0. Flagi SF, ZF i PF nie są modyfikowane.

IN

KOD	INSTRUKCJA	SKRÓCONY OPIS
19	`IN` *reg1*	Wczytuje znak z klawiatury do reg1.
EB	`IN` *mem1*	Wczytuje znak z klawiatury do mem1.

Opis

Wczytuje znak z klawiatury i umieszcza go w najmłodszym bicie argumentu. Argumentem może być rejestr lub miejsce w pamięci. Jeżeli argumentem jest rejestr to pozostałe bajty są ustawiane znakiem najmłodszego bajtu.

Operacja

ARG1 = INPUT;

Modyfikowane Flagi

Żadne.

INC

KOD	INSTRUKCJA	SKRÓCONY OPIS
23	INC *reg1*	Zwiększa reg1 o jeden.
E2	INC *mem1*	Zwiększa mem1 o jeden.

Opis

Dodaje jeden do argumentu(wynikowy), jednocześnie zachowując stan flagi CF. Wynikowy argument może być rejestrem albo miejscem w pamięci. Instrukcja pozwala zwiększenie licznika pętli bez modyfikacji flagi CF.(Żeby zwiększyć licznik z modyfikacją flagi CF należy użyć odpowiednio instrukcji ADD)

Operacja

ARG1 = ARG1 + 1;

Modyfikowane Flagi

Flaga CF nie jest modyfikowana. Flagi OF, SF, ZF i PF są ustawione w zależności od wyniku.

IOUT

KOD	INSTRUKCJA	SKRÓCONY OPIS
05	IOUT *reg1*	Wypisuje reg1 na ekran jako liczbę całkowitą ze znakiem.
F6	IOUT *mem1*	Wypisuje mem1 na ekran jako liczbę całkowitą ze znakiem.

Opis

Wypisuje argument na ekran jako liczbę całkowitą ze znakiem. Argumentem może być rejestr albo miejsce w pamięci.

Operacja

OUT (SIGNED)ARG1;

Modyfikowane Flagi

Żadne.

JA

KOD	INSTRUKCJA	SKRÓCONY OPIS
C6	JA *mem1*	Skocz do mem1 jeśli powyżej(CF=0 i ZF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=0 i ZF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF CF=0 i ZF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JAE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C5	JAE *mem1*	Skocz do mem1 jeśli powyżej lub równe(CF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JB

KOD	INSTRUKCJA	SKRÓCONY OPIS
C4	JB *mem1*	Skocz do mem1 jeśli poniżej(CF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF CF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JBE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C3	JBE *mem1*	Skocz do mem1 jeśli poniżej lub równe(CF=1 lub ZF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=1 lub ZF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=1 lub ZF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JC

KOD	INSTRUKCJA	SKRÓCONY OPIS
C4	JC *mem1*	Skocz do mem1 jeśli przeniesienie(CF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JE

KOD	INSTRUKCJA	SKRÓCONY OPIS
CC	JE *mem1*	Skocz do mem1 jeśli równe(ZF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JG

KOD	INSTRUKCJA	SKRÓCONY OPIS
CA	JG *mem1*	Skocz do mem1 jeśli większe(ZF=0 i SF=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=0 i SF=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=0 i SF=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JGE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C9	JGE *mem1*	Skocz do mem1 jeśli większe lub równe(SF=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  SF=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JL

KOD	INSTRUKCJA	SKRÓCONY OPIS
C8	JL *mem1*	Skocz do mem1 jeśli mniejsze(SF!=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF!=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  SF!=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JLE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C7	JLE *mem1*	Skocz do mem1 jeśli mniejsze lub równe(ZF=1 lub SF!=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=1 lub SF!=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=1 lub SF!=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JMP

KOD	INSTRUKCJA	SKRÓCONY OPIS
CD	JMP *mem1*	Skocz do mem1.

Opis

Wykonuje skok do miejsca w pamięci podanego w argumencie. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

EIP = ADDRESS(ARG1);

Modyfikowane Flagi

Żadne.

JNA

KOD	INSTRUKCJA	SKRÓCONY OPIS
C3	JNA *mem1*	Skocz do mem1 jeśli nie powyżej(CF=1 lub ZF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=1 lub ZF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=1 lub ZF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNAE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C4	JNAE *mem1*	Skocz do mem1 jeśli nie powyżej i nie równe(CF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNB

KOD	INSTRUKCJA	SKRÓCONY OPIS
C5	JAE *mem1*	Skocz do mem1 jeśli nie powyżej(CF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNBE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C6	JNBE *mem1*	Skocz do mem1 jeśli nie poniżej i nie równe(CF=0 i ZF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=0 i ZF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF CF=0 i ZF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNC

KOD	INSTRUKCJA	SKRÓCONY OPIS
C5	JNC *mem1*	Skocz do mem1 jeśli nie przeniesienie(CF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(CF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  CF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNE

KOD	INSTRUKCJA	SKRÓCONY OPIS
CB	JNE *mem1*	Skocz do mem1 jeśli nie równe(ZF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF CF=0 i ZF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNG

KOD	INSTRUKCJA	SKRÓCONY OPIS
C7	JLE *mem1*	Skocz do mem1 jeśli nie większe(ZF=1 lub SF!=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=1 lub SF!=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

IF CF=0 i ZF=0

    EIP = ADDRESS(ARG1);

FI;

Modyfikowane Flagi

Żadne.

JNGE

KOD	INSTRUKCJA	SKRÓCONY OPIS
C8	JL *mem1*	Skocz do mem1 jeśli nie większe i nie równe(SF!=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF!=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

IF CF=0 i ZF=0

    EIP = ADDRESS(ARG1);

FI;

Modyfikowane Flagi

Żadne.

JNL

KOD	INSTRUKCJA	SKRÓCONY OPIS
C9	JNL *mem1*	Skocz do mem1 jeśli nie mniejsze(SF=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  SF=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNLE

KOD	INSTRUKCJA	SKRÓCONY OPIS
CA	JNLE *mem1*	Skocz do mem1 jeśli nie mniejsze i nie równe(ZF=0 i SF=OF).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=0 i SF=OF), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=0 i SF=OF
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNO

KOD	INSTRUKCJA	SKRÓCONY OPIS
C1	JNO *mem1*	Skocz do mem1 jeśli nie przepełnienie(OF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(OF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  OF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNP

KOD	INSTRUKCJA	SKRÓCONY OPIS
BD	JNP *mem1*	Skocz do mem1 jeśli nie parzystość(PF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(PF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  PF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNS

KOD	INSTRUKCJA	SKRÓCONY OPIS
BF	JNS *mem1*	Skocz do mem1 jeśli bez znaku(SF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  SF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JNZ

KOD	INSTRUKCJA	SKRÓCONY OPIS
CB	JNZ *mem1*	Skocz do mem1 jeśli nie zero(ZF=0).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=0), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=0
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JO

KOD	INSTRUKCJA	SKRÓCONY OPIS
C2	JO *mem1*	Skocz do mem1 jeśli przepełnienie(OF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(OF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  OF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JP

KOD	INSTRUKCJA	SKRÓCONY OPIS
BE	JP *mem1*	Skocz do mem1 jeśli parzystość(PF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(PF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  PF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JS

KOD	INSTRUKCJA	SKRÓCONY OPIS
C0	JS *mem1*	Skocz do mem1 jeśli znak(SF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(SF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  SF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

JZ

KOD	INSTRUKCJA	SKRÓCONY OPIS
CC	JE *mem1*	Skocz do mem1 jeśli zero(ZF=1).

Opis

Sprawdza czy flagi rejestru flag są w ustalonym stanie, jeśli tak, czyli warunek jest spełniony(ZF=1), to wykonywany jest skok do miejsca w pamięci podanego w argumencie. Jeśli warunek nie jest spełniony to skok nie jest wykonywany, a egzekucja programu kontynuuje sekwencyjnie zaraz po instrukcji skoku. Argumentem może być wyłącznie miejsce w pamięci.

Operacja

```
IF  ZF=1
        EIP = ADDRESS(ARG1);
FI;
```

Modyfikowane Flagi

Żadne.

LD

KOD	INSTRUKCJA	SKRÓCONY OPIS
02	LD *reg1, reg2*	Wczytuje reg2 do reg1.
FF	LD *reg1, mem2*	Wczytuje mem2 do reg1.

Opis

Wczytuje liczbę podaną w drugim argumencie(źródłowy) i umieszcza ją w rejestrze podanym w pierwszym argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem. Źródłowy argument może być rejestrem albo miejscem w pamięci.

Operacja

ARG1 = ARG2;

Modyfikowane Flagi

Żadne.

LDA

KOD	INSTRUKCJA	SKRÓCONY OPIS
F8	LDA *reg1, mem2*	Wczytuje adres mem2 do reg1.

Opis

Wczytuje adres podany w drugim argumencie(źródłowy) i umieszcza go w rejestrze podanym w pierwszym argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem. Źródłowy argument może być rejestrem albo miejscem w pamięci.

Operacja

ARG1 = ADDRESS(ARG2);

Modyfikowane Flagi

Żadne.

LDB

KOD	INSTRUKCJA	SKRÓCONY OPIS
FE	LDB *reg1, mem2*	Wczytuje bajt ze znakiem z mem2 do reg1.

Opis

Wczytuje najmłodszy liczby podanej w drugim argumencie(źródłowy) i umieszcza ją w rejestrze podanym w drugim argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem. Źródłowy argument może być wyłącznie miejscem w pamięci. Pozostałe bajty są uzupełniane znakiem wczytanego bajtu.

Operacja

ARG1 = (BYTE)ARG2;

Modyfikowane Flagi

Żadne.

LDBU

KOD	INSTRUKCJA	SKRÓCONY OPIS
FD	LDBU *reg1, mem2*	Wczytuje bajt bez znaku z mem2 do reg1.

Opis

Wczytuje najmłodszy liczby podanej w drugim argumencie(źródłowy) i umieszcza ją w rejestrze podanym w drugim argumencie(wynikowy) . Wynikowy argument może być wyłącznie rejestrem. Źródłowy argument może być wyłącznie miejscem w pamięci. Pozostałe bajty są uzupełniane zerem.

Operacja

ARG1 = (BYTE)ARG2;

Modyfikowane Flagi

Żadne.

LEAVE

KOD	INSTRUKCJA	SKRÓCONY OPIS
18	LEAVE	Zwalnia ramkę stosu.

Opis

Zwalnia ramkę stosu stworzoną wcześniej przez instrukcję ENTER. Instrukcja LEAVE kopiuje ramkę stosu(EBP) do rejstru wkaźnika stosu(ESP), co zwalnia tym samym miejsce rezerwowane przez ramkę. Stara ramka stosu jest następnie zdejmowana z nowego szczytu stosu, co przywraca oryginalną ramkę stosu.

Zazwyczaj następną instrukcją po LEAVE jest instrukcja RET, przywracająca sterowanie do wywołującej procedury.

Operacja

ESP = EBP;

EBP = STACK.POP();

Modyfikowane Flagi

Żadne.

LOOP

KOD	INSTRUKCJA	SKRÓCONY OPIS
BC	LOOP *reg1, mem2*	Zmniejsza reg1 o jeden. Skacze do mem2 jeżeli reg1 różne od zera.

Opis

Z każdym wywołanie funkcji LOOP pierwszy argument(licznik) jest zmniejszany o jeden, a następnie wartość w pierwszym argumencie porównywana jest z zerem. Jeżeli licznik jest różny od zera, to wykonywany jest skok w miejsce sprecyzowane w drugim argumencie. Jeżeli licznik wynosi 0 skok nie jest wykonywany, egzekucja kontynuuje sekwencyjnie.

Operacja

```
ARG1 = ARG1 – 1;
IF(ARG1=0)
        EIP = ADDRESS(ARG2);
FI;
```

Modyfikowane Flagi

Żadne.

MUL

KOD	INSTRUKCJA	SKRÓCONY OPIS
1F	`MUL` *`reg1, reg2`*	Mnoży reg1 przez reg2.
E6	`MUL` *`reg1, mem2`*	Mnoży reg1 przez mem2.

Opis

Mnoży pierwszy argument(wynikowy) przez drugi argument(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci.

Przepełnienie jest sygnalizowane przez ustawienie flag CF i OF na 1, przy braku przepełnienia flagi te ustawiane są na 0.

Operacja

ARG1 = ARG1 · ARG2;

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 1, gdy przepełnienie, w przeciwnym przypadku ustawiane na 0. Flagi SF, ZF i PF nie są modyfikowane.

NEG

KOD	INSTRUKCJA	SKRÓCONY OPIS
25	NEG *reg1*	Neguje reg1 w U2.
E0	NEG *mem1*	Neguje mem1 w U2.

Opis

Zamienia wynikowy argument jego uzupełnieniem w U2. (Ta operacja jest równoważna z odjęciem argumentu od zera) Argumentem może być rejestr lub miejsce w pamięci.

Operacja

IF(ARG1 = 0)

THEN: CF=0;

ELSE: CF=1;

FI;

ARG1 = -ARG1;

Modyfikowane Flagi

Flaga CF jest ustawiana na 0 jeżeli argument to zero, W przeciwnym wypadku ustawiana na 1. Flagi ZF, SF, OF, PF ustawiane w zależności od wyniku;

NOP

KOD	INSTRUKCJA	SKRÓCONY OPIS
15	NOP	Nic nie robi.

Opis

Nic nie robi.

Operacja

Modyfikowane Flagi

Żadne.

NOT

KOD	INSTRUKCJA	SKRÓCONY OPIS
36	NOT *reg1*	Neguje reg1 w U1(odwraca każdy bit).
D4	NOT *mem1*	Neguje mem1 w U1(odwraca każdy bit).

Opis

Wykonuje bitową negację argumentu wynikowego, wynik jest zapisywany w argumencie wynikowym. Argument wynikowy może być rejestrem albo miejscem w pamięci. Każdy bit wyniku ustawiany jest wartością 1 jeżeli w wynikowym argumencie odpowiedni bit miał wartość 0; w przeciwnym przypadku ustawiany jest wartością 0.

Operacja

ARG1 = not ARG1

Modyfikowane Flagi

Żadne.

OR

KOD	INSTRUKCJA	SKRÓCONY OPIS
33	OR *reg1, reg2*	Alternatywa reg1 i reg2
D7	OR *reg1, mem2*	Alternatywa reg1 i mem2

Opis

Wykonuje bitową alternatywę pierwszego argumentu(wynikowy) i drugiego argumentu(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Każdy bit wyniku ustawiany jest wartością 1 jeżeli w wynikowym lub źródłowym argumencie odpowiedni bit miał wartość 1; w przeciwnym przypadku ustawiany jest wartością 0.

Operacja

ARG1 = ARG1 or ARG2

Modyfikowane Flagi

Żadne.

OUT

KOD	INSTRUKCJA	SKRÓCONY OPIS
04	OUT *reg1*	Wypisuje reg1 na ekran jako liczbę całkowitą bez znaku.
F7	OUT *mem1*	Wypisuje mem1 na ekran jako liczbę całkowitą bez znaku.

Opis

Wypisuje argument na ekran jako liczbę całkowitą bez znaku. Argumentem może być rejestr albo miejsce w pamięci.

Operacja

OUT (UNSIGNED)ARG1

Modyfikowane Flagi

Żadne.

POP

KOD	INSTRUKCJA	SKRÓCONY OPIS
0E	POP *reg1*	Zdejmuje ze stosu wartość i umieszcza ją w reg1.
F1	POP *mem1*	Zdejmuje ze stosu wartość i umieszcza ją w mem1.

Opis

Zdejmuje wartość ze stosu i umieszcza ją w argumencie wynikowym. Wynikowy argument może być rejestrem albo miejscem w pamięci.

Operacja

ARG1 = STACK.POP();

Modyfikowane Flagi

Żadne.

POPA

KOD	INSTRUKCJA	SKRÓCONY OPIS
11	POPA	Zdejmuje wszystkie rejestry ogólnego zastosowania ze stosu.

Opis

Zdejmuje wszystkie rejestry ogólnego zastosowania ze stosu w kolejności 15, 14, 13, 12, 10, 9, ..., 2, 1, 0. Wartość rejestru ESP jest ignorowana, zamiast zdjęcia jej ze stosu, ESP jest zwiększane o cztery.

Operacja

```
REG15 = STACK.POP();
REG14 = STACK.POP();
REG13 = STACK.POP();
REG12 = STACK.POP();
ESP = ESP + 4;              (*Pomiń 4 bajty stosu*)
REG10 = STACK.POP();
...
REG2 = STACK.POP();
REG1 = STACK.POP();
REG0 = STACK.POP();
```

Modyfikowane Flagi

Żadne.

POPF

KOD	INSTRUKCJA	SKRÓCONY OPIS
10	POPF	Zrzuca ze stosu wartość i umieszcza ją w rejestrze flag.

Opis

Zrzuca ze stosu wartość i umieszcza ją w rejestrze flag. Rejestr flag to liczba 4 bajtowa w której poszczególne bity oznaczają flagi(patrz tabela niżej).

Flaga	CF	PF	ZF	SF	OF
Numer bitu	1	3	7	8	12

Operacja

FLAGS = STACK.POP();

Modyfikowane Flagi

Wszystkie, patrz opis.

PUSH

KOD	INSTRUKCJA	SKRÓCONY OPIS
09	PUSH *reg1*	Wrzuca wartość reg1 na stos.
F2	PUSH *mem1*	Wrzuca wartość mem1 na stos.

Opis

Wrzuca na stos argumentu źródłowy. Źródłowy argument może być rejestrem albo miejscem w pamięci.

Operacja

STACK.PUSH(ARG1);

Modyfikowane Flagi

Żadne.

PUSHA

KOD	INSTRUKCJA	SKRÓCONY OPIS
0C	PUSHA	Wrzuca wszystkie rejestry ogólnego zastosowania na stos.

Opis

Wrzuca wszystkie rejestry ogólnego zastosowania na stos w kolejności 0, 1, 2, ..., 11(stara wartość), 12, 13, 14, 15. Zamiast ESP na stos wrzucana jest wartość sprzed wywołania Instrukcji PUSHA..

Operacja

TEMP = ESP;
STACK.PUSH(REG0);
STACK.PUSH(REG1);
STACK.PUSH(REG2);
...
STACK.PUSH(REG10);
STACK.PUSH(TEMP);
STACK.PUSH(REG12);
STACK.PUSH(REG13);
STACK.PUSH(REG14);
STACK.PUSH(REG15);

Modyfikowane Flagi

Żadne.

PUSHF

KOD	INSTRUKCJA	SKRÓCONY OPIS
0B	PUSHF	Wrzuca wartość rejestru flag na stos.

Opis

Wrzuca wartość rejestru flag na stos. Rejestr flag to liczba 4 bajtowa w której poszczególne bity oznaczają flagi(patrz tabela niżej).

Flaga	CF	PF	ZF	SF	OF
Numer bitu	1	3	7	8	12

Operacja

STACK.PUSH(FLAGS);

Modyfikowane Flagi

Zadne.

RET

KOD	INSTRUKCJA	SKRÓCONY OPIS
01	RET	Powraca z procedury.

Opis

Skacze do miejsca w pamięci zlokalizowanego na szczycie stosu. Adres na szczycie zazwyczaj jest umieszczony przez wywołanie instrukcji CALL. Adres ten jest wtedy adresem następnej instrukcji po instrukcji CALL.

Operacja

EIP = STACK.POP();

Modyfikowane Flagi

Żadne.

RAND

KOD	INSTRUKCJA	SKRÓCONY OPIS
16	RAND *reg1*	Losuje liczbę całkowitą i umieszcza ją w reg1.
EC	RAND *mem1*	Losuje liczbę całkowitą i umieszcza ją w mem1.

Opis

Losuje liczbę całkowitą i umieszcza ją w argumencie wynikowym. Argument może być rejestrem albo miejscem w pamięci.

Operacja

ARG1 = RAND;

Modyfikowane Flagi

Żadne.

ROL

KOD	INSTRUKCJA	SKRÓCONY OPIS
3C	ROL *reg1, reg2*	Obraca w lewo bity reg1 o wartość podaną w reg2.
CE	ROL *reg1, mem2*	Obraca w lewo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa(obraca) w lewo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z prawej strony są uzupełniane bitami przesuniętymi z lewej strony. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Operacja

ARG1 = ROTATE_LEFT(ARG1, ARG2);

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

ROR

KOD	INSTRUKCJA	SKRÓCONY OPIS
3B	ROR *reg1, reg2*	Obraca w prawo bity reg1 o wartość podaną w reg2.
CF	ROR *reg1, mem2*	Obraca w prawo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa(obraca) w prawo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z lewej strony są uzupełniane bitami przesuniętymi z prawej strony. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Operacja

ARG1 = ROTATE_RIGHT(ARG1, ARG2);

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

SAL

KOD	INSTRUKCJA	SKRÓCONY OPIS
38	SAL *reg1, reg2*	Przesuwa w lewo bity reg1 o wartość podaną w reg2.
D2	SAL *reg1, mem2*	Przesuwa w lewo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa w lewo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z prawej strony są uzupełniane zerami. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Instrukcje SAL i SHL wykonują dokładnie te same operacje.

Operacja

ARG1 = ARG1 << ARG2

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

SAR

KOD	INSTRUKCJA	SKRÓCONY OPIS
39	SAR *reg1, reg2*	Przesuwa w prawo bity reg1 o wartość podaną w reg2.
D1	SAR *reg1, mem2*	Przesuwa w prawo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa w prawo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z lewej strony są uzupełniane znakiem argumentu wynikowego. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Operacja

ARG1 = ARG1 >> ARG2

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

SHL

KOD	INSTRUKCJA	SKRÓCONY OPIS
38	SHL *reg1, reg2*	Przesuwa w lewo bity reg1 o wartość podaną w reg2.
D2	SHL *reg1, mem2*	Przesuwa w lewo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa w lewo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z prawej strony są uzupełniane zerami. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Instrukcje SAL i SHL wykonują dokładnie te same operacje.

Operacja

ARG1 = ARG1 << ARG2

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

SHR

KOD	INSTRUKCJA	SKRÓCONY OPIS
37	SHR *reg1, reg2*	Przesuwa w prawo bity reg1 o wartość podaną w reg2.
D3	SHR *reg1, mem2*	Przesuwa w prawo bity reg1 o wartość podaną w mem2.

Opis

Przesuwa w prawo bity wartości podanej w pierwszym argumencie(wynikowy) o wartość podaną w drugim argumencie(źródłowy). Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Bity z lewej strony są zerami. Jedynie pierwsze 5 bitów argumentu źródłowego brane są pod uwagę.

Operacja

ARG1 = ARG1 >>> ARG2

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

SLEEP

KOD	INSTRUKCJA	SKRÓCONY OPIS
1C	`SLEEP` *reg1*	Czeka reg1 milisekund.
E9	`SLEEP` *mem1*	Czeka mem1 milisekund.

Opis

Czeka liczbę sprecyzowaną w argumencie źródłowym milisekund. Argument może być rejestrem albo miejscem w pamięci.

Operacja

SLEEP(ARG1);

Modyfikowane Flagi

Żadne.

ST

KOD	INSTRUKCJA	SKRÓCONY OPIS
FB	ST *reg1, mem2*	Umieszcza reg1 w mem2.

Opis

Umieszcza liczbę podaną w pierwszym argumencie(źródłowy) w drugim argumencie(wynikowy). Źródłowy argument może być wyłącznie rejestrem. Wynikowy argument może być wyłącznie miejscem w pamięci.

Operacja

ARG2 = ARG1;

Modyfikowane Flagi

Żadne.

STB

KOD	INSTRUKCJA	SKRÓCONY OPIS
FA	STB *reg1, mem2*	Umieszcza bajt z reg1 w mem2.

Opis

Umieszcza najmłodszy bajt liczby podanej w pierwszym argumencie(źródłowy) w drugim argumencie(wynikowy). Źródłowy argument może być wyłącznie rejestrem. Wynikowy argument może być wyłącznie miejscem w pamięci.

Operacja

ARG2 = (BYTE)ARG1;

Modyfikowane Flagi

Żadne.

SUB

KOD	INSTRUKCJA	SKRÓCONY OPIS
1E	SUB *reg1, reg2*	Odejmuje reg2 od reg1.
E7	SUB *reg1, mem2*	Odejmuje mem2 od reg1.

Opis

Odejmuje drugi argument(źródłowy) od pierwszego argumentu(wynikowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci.

Instrukcja SUB wykonuje odejmowanie całkowitoliczbowe. Wykonuje obliczenia zarówno dla liczb całkowitoliczbowych ze znakiem jak i bez znaku i ustawia flagi CF oraz OF żeby zasygnalizować przeniesienie (przepełnienie) odpowiednio w znakowym jak i bez znakowym wyniku. Flaga SF sygnalizuje znak wyniku.

Operacja

ARG1 = ARG1 - ARG2

Modyfikowane Flagi

Flagi OF, SF, ZF, CF i PF są ustawione w zależności od wyniku.

TEST

KOD	INSTRUKCJA	SKRÓCONY OPIS
35	TEST *reg1, reg2*	Koniunkcja reg1 i reg2.
D5	TEST *reg1, mem2*	Koniunkcja reg1 i mem2.

Opis

Wykonuje bitową koniunkcję pierwszego argumentu(źródłowy) i drugiego argumentu(źródłowy), wynik nie jest nigdzie zapisywany, zmieniają się wyłącznie flagi. Pierwszy argument może być wyłącznie rejestrem. Drugi argument może być rejestrem albo miejscem w pamięci. Każdy bit wyniku ustawiany jest wartością 1 jeżeli w obu wynikowym i źródłowym argumencie odpowiedni bit miał wartość 1; w przeciwnym przypadku ustawiany jest wartością 0.

Operacja

TEMP = ARG1 and ARG2

Modyfikowane Flagi

Flagi OF i CF są ustawiane na 0; Flagi SF, ZF, PF są ustawiane w zależności od wyniku.

TIME

KOD	INSTRUKCJA	SKRÓCONY OPIS
1B	`TIME` *reg1*	Zapisuje czas w reg1.
EA	`TIME` *mem1*	Zapisuje czas w mem1.

Opis

Zapisuje czas, który upłynął od 01.01.1970 roku mierzony w sekundach w argumencie wynikowym. Dodatkowo w rejestrze ósmym przechowywana jest liczba milisekund która upłynęła od tego dnia modulo 1000. Argument może być rejestrem albo miejscem w pamięci.

Operacja

ARG1 = TIME_IN_SECONDS;

REG8 = TIME_IN_MILLIS mod 1000;

Modyfikowane Flagi

Żadne.

XCHG

KOD	INSTRUKCJA	SKRÓCONY OPIS
13	XCHG *reg1, reg2*	Zamienia miejscami wartości reg1 i reg2.
F0	XCHG *reg1, mem2*	Zamienia miejscami wartości reg1 i mem2.

Opis

Zamienia miejscami wartości między pierwszym argumentem źródłowym a drugim argumentem. Pierwszy argument może być wyłącznie rejestrem. Drugi argument może być rejestrem albo miejscem w pamięci.

Operacja

TEMP = ARG1;

ARG1 = ARG2;

ARG2 = TEMP;

Modyfikowane Flagi

Żadne.

XOR

KOD	INSTRUKCJA	SKRÓCONY OPIS
34	XOR *reg1, reg2*	Alternatywa wykluczająca reg1 i reg2.
D6	XOR *reg1, mem2*	Alternatywa wykluczająca reg1 i mem2.

Opis

Wykonuje bitową alternatywę pierwszego argumentu(wynikowy) i drugiego argumentu(źródłowy) a następnie przechowuje wynik w argumencie wynikowym. Wynikowy argument może być wyłącznie rejestrem; źródłowy argument może być rejestrem albo miejscem w pamięci. Każdy bit wyniku ustawiany jest wartością 1 jeżeli w wynikowym albo źródłowym argumencie odpowiedni bit miał wartość 1; w przeciwnym przypadku ustawiany jest wartością 0.

Operacja

ARG1 = ARG1 xor ARG2

Modyfikowane Flagi

Żadne.

Źródła

Wikibooks contributors, "Asembler x86/Architektura," *Wikibooks,* https://pl.wikibooks.org/w/index.php?title=Asembler_x86/Architektura&oldid=218923 (accessed wrzesień 16, 2019).

Edytorzy Wikipedii, "Architektura von Neumanna," *Wikipedia, wolna encyklopedia,* pl.wikipedia.org/w/index.php?title=Architektura_von_Neumanna&oldid=57461159 (accessed wrzesień 16, 2019).

Edytorzy Wikipedii, "Język maszynowy," *Wikipedia, wolna encyklopedia,* pl.wikipedia.org/w/index.php?title=J%C4%99zyk_maszynowy&oldid=47481004 (accessed wrzesień 16, 2019).