Język Pseudo-Assembler
Instrukcja Obsługi

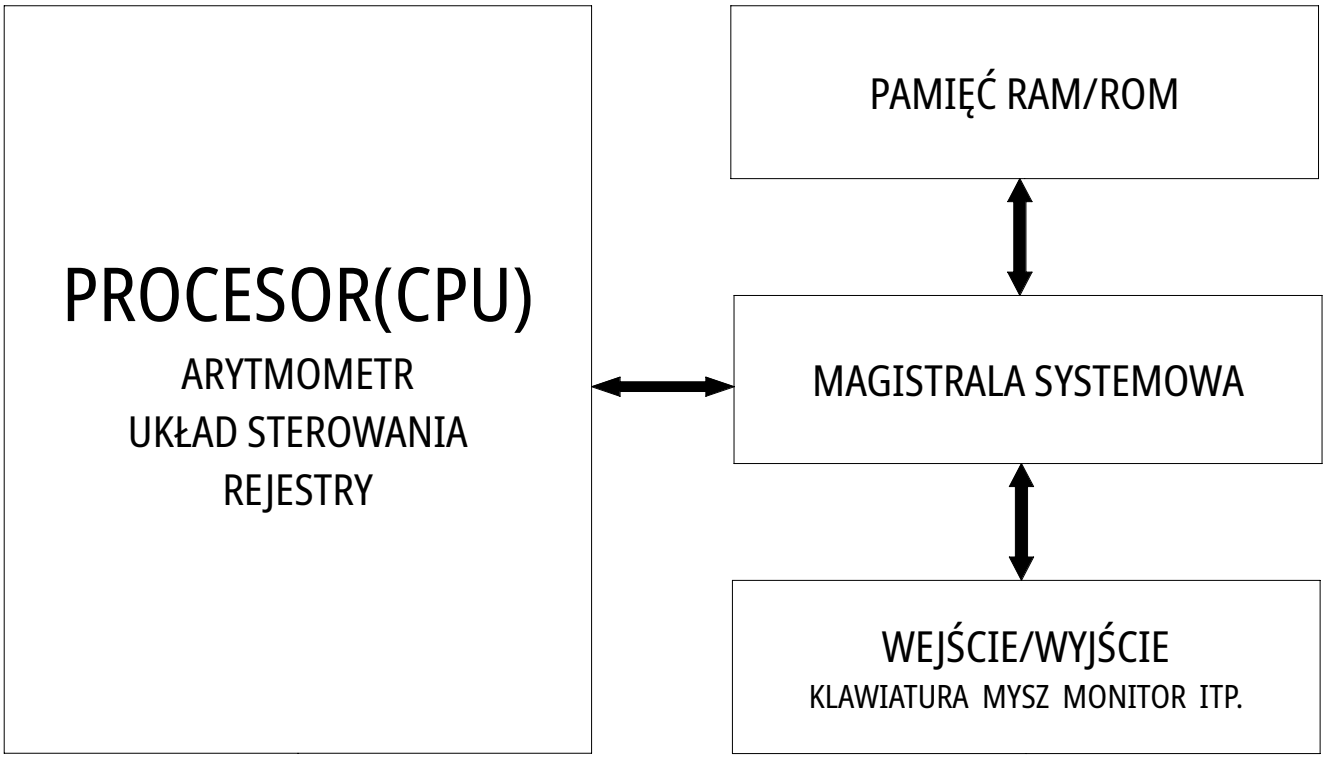
MIT License

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 budowane w celu rozwiązywania konkretnego problemu.
Jeśli za pomocą komputera miało być rozwiązane inne zadanie, należało zbudować inny komputer lub w najlepszym razie zmienić fizyczną budowę już istniejącego. Sytuacja zmieniła się od momentu opracowania przez Johna von Neumanna w 1945 roku modelu komputera. jest to, że 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 sekwencynie
- 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 z rodziny x86 być zaprezentowana w ten sposób: B0|B1|B2|B3

Przykładowo 32-bitowa wartość 0x1BA583D4 (prefiks 0x w Pseudo-Asemblerze 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			
RND	REJESTR 1	---	Losuje wartość i zapisuje ją w rejestrze 1.
RND	ADRES PAMIĘCI	---	Losuje wartość i zapisuje ją pod wskazanym adresem.
NOP	---	---	Nic nie robi.
EXIT	---	---	Kończy działanie komputera.

Przykładowe programy

Zadanie. 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

Zadanie. 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