# Interpreter Pseudoasemblera Specyfikacja

BARTOSZ BŁACHUT, GR. H1

# Spis treści

# Wymagania systemowe

Program przystosowany jest do środowiska wiersza poleceń systemu Windows 10 (testowany na wersji 1903) oraz Windows 7, ze względu na użycie biblioteki *windows.h* w celu estetycznego wyświetlania stanu emulowanej maszyny.

Program należy uruchamiać w wierszu poleceń systemu Windows, korzystając z pliku wykonywalnego *interpreter.exe*, który znajduje się w folderze *_executable*. Program najlepiej uruchomić przy **zmaksymalizowanym** oknie konsolowym.

# Działanie programu

Po uruchomieniu programu w wierszu poleceń, użytkownik jest proszony o podanie ścieżki do pliku tekstowego, zawierającego instrukcje w pseudoassemblerze.

```
_______ PSEUDO ASSEMBLER INTERPRETER _______

Please provide the path for the file containing the assembler instructions:
../_tests/bubble_sort.txt
```

Następnie użytkownik może zdecydować czy uruchomić program w trybie debuggera.

```
Run in debug mode? [y/n]: y
```

Później, w zależności od decyzji użytkownika, program wykonuje kod pseudoasemblera z pliku, wskazanego przez użytkownika, na dwa różne sposoby.

## 1. *Tryb debuggera*

Po wczytaniu kodu pseudoasemblera, program wykonuje go linijka po linijce, czekając na naciśnięcie klawisza *Enter* przez użytkownika. W konsoli wyświetlane są stany rejestrów, zmiennych, cały kod pseudoasemblera, obecnie wykonywana linijka kodu oraz log – jakie zmiany nastąpiły po wykonaniu obecnej linii kodu.

```
log_1:   created and array of 10 elements labelled TAB with an assigned value: 0

The following command has been debugged:
1       TAB     DS      10*INTEGER

Declared variables:
label:  TAB   values: [    0    0    0    0    0    0    0    0    0    0   ]

program state registry: 0

registry 0:     0
registry 1:     0
registry 2:     0
registry 3:     0
registry 4:     0
registry 5:     0
registry 6:     0
registry 7:     0
registry 8:     0
registry 9:     0
registry 10:    0
registry 11:    0
registry 12:    0
registry 13:    0
registry 14:    0
registry 15:    0

Press Enter to go to the next step...
```

```
Your code:
1.TAB           DS    10*INTEGER
2.CZTERY        DC    INTEGER(4)
3.DZIESIEC      DC    INTEGER(10)
4.JEDEN         DC    INTEGER(1)
5.ZERO          DC    INTEGER(0)
6.              LA    0,TAB
7.              L     1,CZTERY
8.TAB_FILL      C     1,DZIESIEC
9.              JZ    DALEJ
10.             A     1,JEDEN
11.             J     TAB_FILL
12.DALEJ        C     1,JEDEN
13.             JN    SORTOWANIE
14.             ST    1,0(0)
15.             A     0,CZTERY
16.             S     1,JEDEN
17.             J     DALEJ
18.SORTOWANIE   L     3,JEDEN
19.             LA    0,TAB
20.             L     1,ZERO
21.             L     2,ZERO
22.PETLA_Z      L     3,ZERO
23.             L     2,ZERO
24.             LA    0,TAB
25.PETLA_W      L     4,DZIESIEC
26.             SR    4,1
27.             S     4,JEDEN
28.             L     6,0(0)
. . .
```

## 2. *Tryb normalnej kompilacji*

W razie zrezygnowania przez użytkownika z trybu debuggera, program wypisze wykonany kod pseudoasemblera, wszystkie logi – jakie zmiany nastąpiły po wykonaniu całego kodu pseudoasemblera i końcowy stan zmiennych oraz rejestrów maszyny.

```
log_0:   input parsed

Your code:
1.              TAB     DS      10*INTEGER
2.            CZTERY    DC      INTEGER(4)
3.         DZIESIEC     DC      INTEGER(10)
4.            JEDEN     DC      INTEGER(1)
5.             ZERO     DC      INTEGER(0)
6.                      LA      0,TAB
7.                      L       1,CZTERY
8.         TAB_FILL     C       1,DZIESIEC
9.                      JZ      DALEJ
10.                     A       1,JEDEN
11.                     J       TAB_FILL
12.           DALEJ     C       1,JEDEN
13.                     JN      SORTOWANIE
.
.
.
35.                     ST      7,0(0)
36.         JEST_OK     A       0,CZTERY
37.                     A       2,JEDEN
38.                     CR      2,4
39.                     JN      PETLA_W
40.                     A       1,JEDEN
41.                     C       3,ZERO
42.                     JP      PETLA_Z

log_1:   created and array of 10 elements labeled TAB with an assigned value: 0
log_2:   created a variable labeled CZTERY with an assigned value: 4
log_3:   created a variable labeled DZIESIEC with an assigned value: 10
.
.
.
log_663: registry 1 has been assigned a new value equal to 10; the program state registry is equal to: 1
log_664: registry 3 has been compared with 0; the program state registry is equal to: 0

The machine's state after every line of code in your input file is presented below:

Declared variables:
label:             TAB   values: [    1    2    3    4    5    6    7    8    9   10   ]
label:          CZTERY   value: 4
label:        DZIESIEC   value: 10
label:           JEDEN   value: 1
label:            ZERO   value: 0

program state registry: 0

registry 0:     4
registry 1:     10
registry 2:     1
registry 3:     0
registry 4:     0
registry 5:     0
registry 6:     1
registry 7:     2
registry 8:     0
registry 9:     0
registry 10:    0
registry 11:    0
registry 12:    0
registry 13:    0
registry 14:    0
registry 15:    0

Press Enter to go to the next step...
log_665:        memory freed
Press any key to continue . . ..
```

# Struktura programu

Program korzysta z czterech bibliotek, z których najistotniejsza to *windows.h* – potrzebna do eleganckiego wyświetlania stanu maszyny i zmieniania koloru czcionki w konsoli systemu Windows.

Podstawowym elementem programu są trzy listy dwukierunkowe:

- *memory_block* – lista, będąca pamięcią emulowanej maszyny. Każdy *block*, mający swój adres i wartość, reprezentuje kolejne cztery bajty;
- *label* – lista, zawierająca etykiety. Każdy element tej listy jest połączony z modelem pamięci, poprzez wskaźnik na odpowiedni segment listy *memory_block;*
- *command* – lista, przechowująca kod pseudoasemblera z podanego przez użytkownika pliku, po dokonaniu analizy i podzieleniu go przez funkcję *input_parser*;

Program jest podzielony na odpowiednie funkcje, odpowiedzialne za poszczególne zadania. Całość przebiega w przybliżeniu w poniższy sposób:

1. Po podaniu ścieżki do pliku z kodem pseudoasemblera i wybraniu odpowiedniego trybu przez użytkownika, kod z pliku zostaje analizowany i rozdzielany przez funkcje *input_parser* i zapisywany w liście *command.*
2. Jeśli użytkownik nie zdecyduje się na tryb debuggera, funkcja *main* wypisze w konsoli zawartość listy *command*, czyli w jaki sposób zawartość pliku została podzielona przez funkcję *input_parser*.
3. Następnie w funkcji *main* zostanie wywołana funkcja *interpret*, która będzie wykonywać kod pseudoasemblera linijka po linijce, czytając kolejne elementy listy *command.*
4. Każde polecenie pseudoasemblera (np.: *DC*, *L* itp.) zostaje obsługiwane przez osobną funkcję, wywołaną przez *interpret*. Funkcje: *arithmetic_operations* i *registry_arithmetic_operations* obsługują polecenia *A*, *S*, *M*, *D*, *C* oraz *AR*, *SR*, *MR*, *DR*, *CR*. Pozostałe funkcje: *DC*, *L*, *LA*, *LR* i *ST* obsługują odpowiednio polecenia: *DC*, *L*, *LA*, *LR* i *ST*. Instrukcje skoku zostają obsłużone rekurencyjnie przez funkcję *interpret*. Przy wykonaniu każdego z poleceń arytmetycznych oraz porównania, uaktualniana jest odpowiednio wartość rejestru stanu programu.
5. Podczas wykonywania każdej następnej linijki kodu, na ekran zostają wypisywane logi, które informują użytkownika, o tym co zmieniło się w stanie maszyny podczas wykonywania ostatniego polecenia.

W trybie debuggera funkcja *print_machine_state* wypisuje w konsoli stany rejestrów, utworzone zmienne oraz cały wykonywany kod z podkreśloną linijką, która właśnie została wykonana.

# Testy

Przygotowane przeze mnie testy, zawierające krótkie programy napisane w pseudoasemblerze, znajdują się w katalogu *_tests*.

Dostępne testy:

1. *bubble_sort.txt* – sortowanie algorytmem bąbelkowym dziesięcioelementowego wektora
2. *insertion_sort.txt* – sortowanie algorytmem przez wstawianie dziesięcioelementowego wektora
3. *kwadrat.txt* – obliczanie ilości rozwiązań równania kwadratowego
4. *max.txt* – algorytm, znajdujący maksymalną wartość w dziesięcioelementowym wektorze
5. *max2.txt* – algorytm, znajdujący dwie maksymalne wartości w dziesięcioelementowym wektorze
6. *merging.txt* – algorytm, łączący rosnąco dwa posortowane (rosnąco) wektory
7. *silnia.txt* – algorytm, liczący 5!
8. *ukladanie.txt* – algorytm, układający wektor w sposób taki, że liczby ujemne będą znajdować się na początku, zera w środku, a dodatnie na końcu

# Przyjęta składnia pseudoasemblera

## 1. *Forma komend:*

`[<etykieta>]  <kod rozkazu>  <argument 1>,<argument 2>`

Pomiędzy argumentem 1., a argumentem 2. **nie** występuje separator. Pole etykiety jest nieobowiązujące.

## 2. *Rozkazy arytmetyczne*

- `A <rejestr 1>,<adres komórki pamięci>` - rozkaz dodawania (*ang. add*)
- `AR <rejestr 1>,<rejestr2>` - rozkaz dodawania rejestrów (*ang. add registry*)
- `S <rejestr 1>,<adres komórki pamięci>` - rozkaz odejmowania (*ang. subtract*)
- `SR <rejestr 1>,<rejestr2>` - rozkaz odejmowania rejestrów (*ang. subtract registry*)
- `M <rejestr 1>,<adres komórki pamięci>` - rozkaz mnożenia (*ang. multiply*)
- `MR <rejestr 1>,<rejestr2>` - rozkaz mnożenia rejestrów (*ang. multiply registry*)
- `D <rejestr 1>,<adres komórki pamięci>` - rozkaz dzielenia (*ang. divide*)
- `DR <rejestr 1>,<rejestr2>` - rozkaz dzielenia rejestrów (*ang. divide registry*)
- `C <rejestr 1>,<adres komórki pamięci>` - rozkaz porównania (*ang. compare*)
- `CR <rejestr 1>,<rejestr2>` - rozkaz porównania rejestrów (*ang. compare registry*)

Wszystkie operacje arytmetyczne wpisują wynik swojego działania do rejestru podanego jako `<rejestr 1>`. Po wykonaniu każdej z powyższych rozkazów, aktualizowana jest wartość rejestru stanu programu w sposób następujący:

- dla wyniku, będącego liczbą równą zeru, przypisywana jest wartość: *0*
- dla wyniku, będącego liczbą dodatnią, przypisywana jest wartość: *1*
- dla wyniku, będącego liczbą ujemną, przypisywana jest wartość: *2*

## 3. Rozkazy przesłania

- `L <rejestr 1>,<adres komórki pamięci>` - *ang. load* - przesyła wartość komórki pamięci do rejestru
- `LR <rejestr 1>,<rejestr 2>` - *ang. load registry* - przesyła wartość rejestru drugiego do pierwszego
- `ST <rejestr 1>,<adres komórki pamięci>` - *ang. store* - przesyła wartość rejestru do komórki pamięci o podanym adresie
- `LA <rejestr 1>,<adres komórki pamięci>` - *ang. load address* – przesyła adres komórki pamięci do rejestru

## 4. Rozkazy skoków

Wszystkie te rozkazy powodują wykonanie rozkazu pod wskazaną etykietą, jeśli spełnione są ewentualne warunki:

- `J <etykieta>` - *ang. jump* - skok bezwarunkowy
- `JP <etykieta>` - *ang. jump positive* - skok, jeśli bity znaku w rejestrze stanu wskazują wartość dodatnią
- `JN <etykieta>` - *ang. jump negative* - skok, jeśli bity znaku w rejestrze stanu wskazują wartość ujemną
- `JZ <etykieta>` - *ang. jump zero* - skok, jeśli bity znaku w rejestrze stanu wskazują wartość 0

## 5. Dyrektywy rezerwacji pamięci

- `DC INTEGER (<liczba całkowita>)` - rezerwuje 4B pamięci i zapisuje na nich liczbę
- `DS INTEGER` - rezerwuje 4 B pamięci i zapisuje na nich liczbę 0
- `DC <liczba komórek> * INTEGER (<liczba całkowita>)` - rezerwuje wskazana liczbę komórek pamięci, zapisuje wskazaną wartość
- `DS <liczba komórek> * INTEGER` - rezerwuje wskazaną liczbę komórek pamięci, zapisuje na nich liczbę 0