Interpreter Pseudoasemblera Specyfikacja

Bartosz Błachut, gr. H1

Spis treści

[Wymagania systemowe 3](#_Toc26794254)

[Działanie programu 3](#_Toc26794255)

[Struktura programu 5](#_Toc26794256)

[Testy 6](#_Toc26794257)

[Przyjęta składnia pseudoasemblera 6](#_Toc26794258)

# 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.

#### 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 Your code:

1.TAB DS 10\*INTEGER

The following command has been debugged: 2.CZTERY DC INTEGER(4)

1 TAB DS 10\*INTEGER 3.DZIESIEC DC INTEGER(10)

4.JEDEN DC INTEGER(1)

Declared variables: 5.ZERO DC INTEGER(0)

label: TAB values: [ 0 0 0 0 0 0 0 0 0 0 ] 6. LA 0,TAB

7. L 1,CZTERY

program state registry: 0 8.TAB\_FILL C 1,DZIESIEC

9. JZ DALEJ

registry 0: 0 10. A 1,JEDEN

registry 1: 0 11. J TAB\_FILL

registry 2: 0 12.DALEJ C 1,JEDEN

registry 3: 0 13. JN SORTOWANIE

registry 4: 0 14. ST 1,0(0)

registry 5: 0 15. A 0,CZTERY

registry 6: 0 16. S 1,JEDEN

registry 7: 0 17. J DALEJ

registry 8: 0 18.SORTOWANIE L 3,JEDEN

registry 9: 0 19. LA 0,TAB

registry 10: 0 20. L 1,ZERO

registry 11: 0 21. L 2,ZERO

registry 12: 0 22.PETLA\_Z L 3,ZERO

registry 13: 0 23. L 2,ZERO

registry 14: 0 24. LA 0,TAB

registry 15: 0 25.PETLA\_W L 4,DZIESIEC

26. SR 4,1

Press Enter to go to the next step... 27. S 4,JEDEN

28. L 6,0(0)

. . .

#### 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

#### Forma komend:

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

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

#### 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*

#### 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

#### 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

#### 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