

# Sprawozdanie z projektu Pseudo Assembler Interpreter

Tymoteusz Bartnik

Październik 2019

## **Streszczenie**

Zadaniem projektu było zrobienie interpretera do języka pseudo assembler, który powstał na bazie assemblera. Program został wykonany w języku wysokiego poziomu - C.

Opiekun projektu: prof. dr hab. inż. Władysław Homenda prof. PW  
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej

# Spis treści

<b>1</b>	<b>Wstęp</b>	<b>3</b>
<b>2</b>	<b>Język pseudo assembler</b>	<b>3</b>
2.1	Krótką charakteryzacja . . . . .	3
2.2	Operacje . . . . .	3
2.2.1	Alokacja pamięci . . . . .	3
2.2.2	Arytmetyczne . . . . .	4
2.2.3	Porównanie . . . . .	4
2.2.4	Skoki . . . . .	5
2.2.5	Odwołanie do pamięci . . . . .	5
<b>3</b>	<b>Struktura programu</b>	<b>5</b>
3.1	Main . . . . .	5
3.2	Structures . . . . .	6
3.2.1	Var . . . . .	7
3.2.2	Command . . . . .	7
3.2.3	Operation . . . . .	8
3.2.4	Reg . . . . .	8
3.3	Interface . . . . .	9
3.4	Parser . . . . .	9
3.5	Commands . . . . .	10
<b>4</b>	<b>Kody błędów</b>	<b>10</b>
<b>5</b>	<b>Działanie programu</b>	<b>12</b>
5.1	Tryb DEBUG . . . . .	12
5.2	Tryb PRODUCTION . . . . .	13
<b>6</b>	<b>Zrzuty ekranu</b>	<b>13</b>
<b>7</b>	<b>Zakończenie</b>	<b>14</b>

# 1 Wstęp

Głównym zadaniem projektu było zrobienie prostego interpretera do języka pseudo assembler, który wypisywałby co dzieje się na poszczególnych rejestrach oraz zmiennych w trakcie wykonywania kolejnych operacji. Założeniem było, że kod pseudo assemblera, który program odczytuje z pliku jest w 100% poprawny, jednak postanowio pójść o krok dalej i w programie dodatkowo sprawdza się poprawność składni i jeśli nie jest ona odpowiednia wypisuje się błędy wraz z odpowiednimi i unikatowymi dla nich kodami, aby program ten mógł również służyć do nauki języka pseudo assembler. Cały program jest podzielony na kilka plików źródłowych wraz z plikami nagłówkowymi, aby oddzielić w pewien sposób logikę jego działania. Cały program został napisany w języku angielskim, tzn. nazwy zmiennych, funkcji, komentarze, dane wyjściowe.

## 2 Język pseudo assembler

### 2.1 Krótka charakteryzacja

Język pseudo assembler powstał jako pochodna rodziny języków assemblera. Jest on znacznie bardziej uproszczony niż typowy assembler, jednak idealnie nadaje się do nauki programistycznego myślenia i przedstawienia, jak działają zmienne i adresy w komputerze.

Typowy model komputera posiada pamięć, 16 rejestrów (numerowanych od 0 do 15) oraz rejestr stanu, który przechowuje informację o znaku wyniku ostatnio wykonanej operacji.

### 2.2 Operacje

#### 2.2.1 Alokacja pamięci

- {słowo} DC INTEGER({wartość})
- {słowo} DC {ile}\*INTEGER({wartość})
- {słowo} DS INTEGER
- {słowo} DS {ile}\*INTEGER

### 2.2.2 Arytmetyczne

- {etykieta} A {rejestr}, {słowo} - dodawanie wartości słowa do rejestru
- {etykieta} AR {rejestr1}, {rejestr2} - dodawanie wartości rejestru2 do rejestru1
- {etykieta} S {rejestr}, {słowo} - odejmowanie wartości słowa od rejestru
- {etykieta} SR {rejestr1}, {rejestr2} - odejmowanie wartości rejestru2 od rejestru1
- {etykieta} M {rejestr}, {słowo} - pomnożenie rejestru przez wartość słowa
- {etykieta} MR {rejestr1}, {rejestr2} - pomnożenie rejestru1 przez wartość rejestru2
- {etykieta} D {rejestr}, {słowo} - podzielenie rejestru przez wartość słowa
- {etykieta} DR {rejestr1}, {rejestr2} - podzielenie rejestru1 przez wartość rejestru2

### 2.2.3 Porównanie

Porównanie odbywa się poprzez odjęcie dwóch wartości i znak wyniku działania zapisywany jest do rejestru stanu (0 - zero, 1 - wynik dodatni, 2 - wynik ujemny).

- {etykieta} C {rejestr}, {słowo} - porównanie rejestru z wartością słowa
- {etykieta} CR {rejestr1}, {rejestr2} - porównanie rejestru1 z rejestrem2

#### 2.2.4 Skoki

- {etykieta} J {etykieta2} - skok do operacji z etykietą2
- {etykieta} JZ {etykieta2} - skok do operacji z etykietą2, gdy rejestr stanu ma wartość 0
- {etykieta} JP {etykieta2} - skok do operacji z etykietą2, gdy rejestr stanu ma wartość 1
- {etykieta} JN {etykieta2} - skok do operacji z etykietą2, gdy rejestr stanu ma wartość 2

#### 2.2.5 Odwołanie do pamięci

- {etykieta} L {rejestr}, {słowo} - przypisanie do rejestru wartość słowa
- {etykieta} LA {rejestr}, {słowo} - przypisanie do rejestru adres słowa
- {etykieta} LR {rejestr1}, {rejestr2} - przypisanie do rejestru1 wartość rejestru2
- {etykieta} ST {rejestr}, {słowo} - przypisanie do słowa wartość rejestru

### 3 Struktura programu

#### 3.1 Main

Główny plik programu - main, zawiera w sobie operacje preprocesora podłączające do niego inne pliki programu. W głównej funkcji występuje deklaracja zmiennych takkich jak:

- vars - (typ Var) lista przechowująca informacje o zmiennych deklarowanych w pseudo assemblerze,
- reg - (typ Reg) tablica , która zawiera informacje o rejestrach z pseudo asseblera,
- statusReg - (typ int) rejestr stanu,
- varsTab - (typ int) dynamiczna tablica przechowująca wartości zmiennych z pseudo assemblera,

- `commands` - (typ `Command`) tablica komend, zawierająca informacje o poszczególnych komendach: ich nazwach, liczbie parametrów i wskaźnikach do funkcji,
- `operations` - (typ `Operation`) tablica operacji, w której zapisywane są poszczególne linie kodu pseudo assemblera jako operacje.

W funkcji `main` następuje wywoływanie po sobie innych funkcji z poszczególnych plików.

## 3.2 Structures

W pliku źródłowym `structures.c` znajduje się implementacja funkcji odpowiadających za działania na listach: `vars` oraz `operations`. Plik nagłówkowy `structures.h` posiada prototypy tych funkcji oraz min. kilka stałych zdefiniowanych za pomocą operacji preprocesora np.

- `MAX_WORD_SIZE` - maksymalna długość poszczególnego słowa (domyślnie 100),
- `COMMANDS_NUM` - liczba komend pseudo assemblera (domyślnie 21).

Dodatkowo znajdują się tu również zmienne globalne: `DEBUG`, `PRODUCTION`, `FILE_NAME` (muszą być to zmienne, ponieważ jej wartość jest pobierana od użytkownika na początku działania programu), a także zmienna `regChanged`, która zapisuje ostatnio zmieniony rejestr. Program poza nimi jest pozbawiony zmiennych globalnych i wszystkie potrzebne zmienne są przekazywane do funkcji jako parametry lub wskaźniki.

W pliku tym znajdują się również deklarację typów złożonych - struktur takich jak:

### 3.2.1 Var

Struktura zapisująca informacje o zmiennych pseudo assemblera.

- label[ ] - (typ char) nazwa zmiennej,
- index - (typ int) index w tablicy wszystkich zmiennych, symulacja adresu (adresy są numerowane od 0 i kolejna zmienna ma adres +1, np. druga zmienna ma adres 1),
- elements - (typ int) ile elementów posiada,
- prev, next - (typ Var\*) kolejny i poprzedni element w liście.

### 3.2.2 Command

Struktura zapisująca wszystkie komendy pseudo assemblera (np. A, AR).

- name - (typ char\*) nazwa komendy, wprowadzana na "sztywno" w funkcji initCommands,
- paramNum - (typ int) ilość przyjmowanych parametrów (wszystkie komendy przyjmują 0, 1 lub 2 parametry),
- dataSection - (typ bool) czy komenda występuje w sekcji inicjalizacji zmiennych,
- regVarType - (typ int) typ parametrów to znaczy: zmienna przyjmuje wartość 0 - jeśli komenda nie przyjmuje parametrów, które są rejestrem, 1 - jeśli tylko pierwszy parametr jest rejestrem, 2 - gdy oba parametry są rejestrami,
- func - (typ function) wskaźnik do funkcji danej komendy.

### 3.2.3 Operation

Struktura zapisująca poszczególne operacje pseudo assemblera (linie kodu).

- fullLine[ ] - (typ char) cała linijka kodu pseudo assemblera,
- label[ ] - (typ char) etykieta operacji (jeśli występuje),
- command[ ] - (typ char) nazwa komendy pseudo assemblera (np. A, C),
- paramsString[ ] - (typ char) parametry komendy jako ciąg znaków,
- param1[ ] - (typ char) pierwszy parametr komendy,
- param2[ ] - (typ char) drugi parametr komendy (jeśli istnieje),
- paramsNum - (typ int) liczba parametrów,
- isLabel - (typ bool) flaga mówiąca czy etykieta operacji występuje,
- isCommand - (typ bool) flaga mówiąca czy komenda w operacji występuje,
- areParams - (typ bool) flaga mówiąca czy występują parametry,
- lineNum - (typ int) numer linii operacji,
- commandWsk - (typ Command\*) wskaźnik na komendę operacji,
- prev, next - (typ Operation\*) wskaźniki na poprzedni i następny element z listy.

### 3.2.4 Reg

Struktura symulująca rejestry.

- value - (typ int) wartość jaką posiada dany rejestr,
- isEmpty - (typ bool) flaga mówiąca czy podany rejestr jest pusty.



### 3.3 Interface

Pliki `interface.h` oraz `interface.c` odpowiadają za wypisywanie informacji do konsoli w ładnym stylu za pomocą specjalnych kodów ANSI pozwalających min. na zmianę koloru czcionki w konsoli czy też pogrubienie jej. Znajdują się w nich funkcje min. do wypisywania wszystkich elementów z listy `vars` wraz z odpowiednimi wartościami czy też rejestrów, a także informacje rozpoczynające działanie programu, wypisywanie błędów itp.

W funkcji `initInterface()`, znajduje się specjalna instrukcja, która po włączeniu konsoli rozszerza ją na cały ekran. Jest to potrzebne, ponieważ podczas wykonywania poszczególnych operacji pseudo assemblera, program tworzy 3 kolumny:

- I - Kolumna pokazująca cały kod pseudo assemblera oraz podkreślająca obecnie wykonywaną instrukcję.
- II - Kolumna z rejestrami, obecnie zmieniany rejestr jest podkreślany.
- III - Kolumna ze zmiennymi oraz kilkoma informacjami o nich takimi jak: adres, ilość elementów, poszczególne elementy czy też nazwa.

Jeśli okno konsoli będzie zbyt małe, program nie będzie działał prawidłowo.

### 3.4 Parser

Plik `parses.c` jest jednym z bardziej rozbudowanych w całym programie. Jego zadaniem jest przeanalizowanie pliku, w którym znajduje się kod pseudo assemblera (domyślnie `file.txt`), sprawdzenie jego poprawności oraz zapisanie poszczególnych linii kodu do listy operacji.

### 3.5 Commands

Plik `commands.c` w funkcji `initCommands()` posiada zapisanie poszczególnych komend do tablicy. Dzięki takiemu rozwiązaniu np. w parserze nie trzeba wstawiać instrukcji warunkowych (`if`) ze wszystkimi komendami, aby sprawdzić czy ciąg znaków jest komendą. Struktura komend posiada także informacje o liczbie parametrów jakie przyjmuje, informacje czy znajdują się w części deklaracji zmiennych pseudo assemblera. Dzięki temu w łatwy sposób można zwalidować kod pseudo assemblera. Struktura `Command` posiada także wskaźnik do funkcji, która operuje na rejestrach i zmiennych. W pliku `commands.c` znajduje się również implementacja tych funkcji wraz z paroma funkcjami pomocniczymi jak `toInt()`, która konwertuje ciąg znaków na zmienną typu `integer`.

## 4 Kody błędów

Funkcja `logError()`, znajdująca się w pliku `interface.c`, przyjmuje wskaźnik do zmiennej typu `char` (`errorInfo`), numer błędu (`errorNum`) oraz numer linii, w której wystąpił błąd (`lineNum`). Wypisuje błąd w formacie: `[ERROR X] Line: NrLinii — Info: InfoBłędu`. Każdy błąd przy wypisywaniu posiada informację o jego kodzie. Część błędów jest 'wypluwana' podczas procesu parsowania, natomiast inna podczas wykonywania poszczególnych operacji.

Kod błędu	Informacja
1	Cannot open the file
2	Command in the wrong place
3	No command in right place
4	Initialization of the variable in the program
5	The name of the label cannot start with a number
6	The label contains forbidden characters
7	No parameters
8	Wrong number of parameters
9	Wrong register
10	The provided variable is not initialized
11	Wrong name of variable
12	Incorrect syntax
13	Invalid value of shift
14	Invalid value of address
15	Bad label on jump
16	No number of items
17	The number of items cannot be zero
18	Invalid variable type
19	The variable value is missing

Tabela 1: Kody błędów

## 5 Działanie programu

Uogólnione działanie programu:

1. Dołączenie potrzebnych plików.
2. Inicjalizacja zmiennych.
3. Inicjalizacja rejestrów (ustawienie flagi, że wszystkie są puste).
4. Inicjalizacja interfacu (umożliwienie używania kodów ASCII w konsoli oraz domyślnie powiększenie okna konsoli).
5. Pobranie danych startowych od użytkownika (nazwa pliku, status trybu DEBUG oraz PRODUCTION).
6. Ekran powitania ze wstępnymi informacjami.
7. Inicjalizacja komend (zapisanie do tablicy wszystkich komend).
8. Parsowanie.
9. Wykonywanie poszczególnych operacji.
10. Wypisanie końcowej zawartości rejestrów oraz zmiennych.
11. Podsumowanie działania programu.
12. Zwolnienie zaalokowanej pamięci w zmiennych dynamicznych.

Program posiada dwa tryby działania, które są ustalane przez użytkownika przy starcie: DEBUG oraz PRODUCTION.

### 5.1 Tryb DEBUG

Dzięki włączeniu trybu debugowania, program wypisuje dodatkowe informacje w konsoli, takie jak informacje odnośnie parsowania czy strukturę wszystkich operacji.

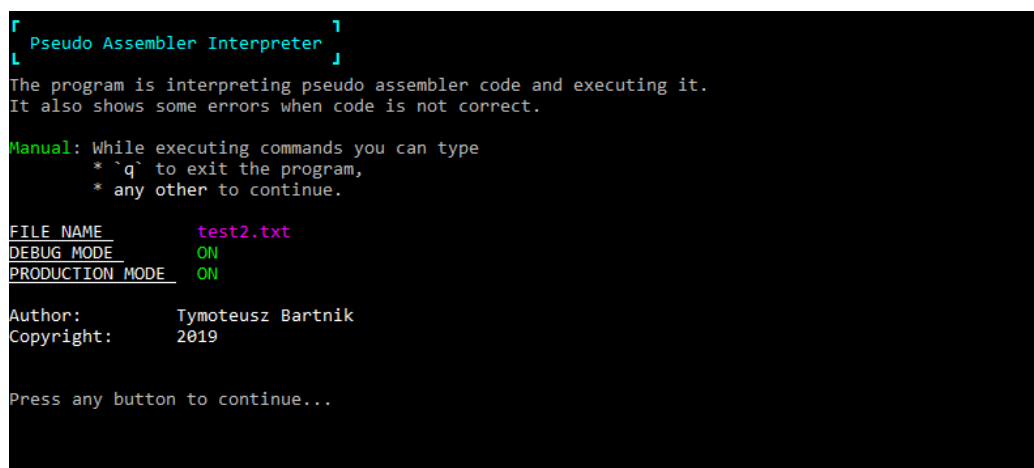
## 5.2 Tryb PRODUCTION

Kiedy ten tryb jest włączony, program po wykonaniu poszczególnych operacji czeka na rozkaz użytkownika, który może wpisać następujące komendy:

- **q** - aby wyjść z programu,
- **inny przycisk** - aby przejść do następnej operacji.

Dodatkowo program wypisuje wszystkie operacje (obecnie wykonywana jest zaznaczona), rejestry, a także zmienne z pseudo assemblera. Dzięki zastosowaniu specjalnych kodów ASCII wypisywanie poszczególnych sekcji można było umieścić w 3 osobnych kolumnach. Trzeba było jednak zrobić parę założeń, aby np. przy długiej etykiecie lub dużej liczbie elementów w tablicy, zawartość konsoli "nie rozjechała się". Zrobiono to za pomocą specjalnej funkcji shortenToSize(), która skraca podaną jako parametr zmienną typu char do określonej długości (jeśli jest za długa) i dodaje na jej końcu napis (...). Domyślnie wszystkie etykiety mają ustawiony maksymalny limit na 20, a parametry operacji na 23. Zawartość tablicy zmiennej pseudo assemblera, gdy jest odpowiednio duża, jest przesuwana do nowej linii.

## 6 Zrzuty ekranu



```
[ Pseudo Assembler Interpreter ]
The program is interpreting pseudo assembler code and executing it.
It also shows some errors when code is not correct.

Manual: While executing commands you can type
        * 'q' to exit the program,
        * any other to continue.

FILE NAME      test2.txt
DEBUG MODE     ON
PRODUCTION MODE ON

Author:        Tymoteusz Bartnik
Copyright:     2019

Press any button to continue...
```

Rysunek 1: Wstępne informacje

```
┌ All operations ┐
└────────────────┘
1.  A,2      DC 5*INTEGER(5)
2.  A,B      DS 3*INTEGER
3.  A,C      DC INTEGER(69)
4.  CZTERYDZIESI(...) DC INTEGER(4)
5.  TD      DC INTEGER(1)
6.  TD      DC INTEGER(4)
10.  LA      4,CZTERYDZIESI(...)
12.  L      6,32(14)
14.  A      4,TD
16.  L      5,0(4)
18.  A      3,4(4)
20.  ST      3,4(4)

┌ All Registers ┐
└────────────────┘
reg[0] = ??
reg[1] = ??
reg[2] = ??
reg[3] = 6
reg[4] = ??
reg[5] = ??
reg[6] = ??
reg[7] = ??
reg[8] = ??
reg[9] = ??
reg[10] = ??
reg[11] = ??
reg[12] = ??
reg[13] = ??
reg[14] = Address of data section
reg[15] = Address of program section
Status register = 002210

┌ All vars with values ┐
└────────────────┘
A,2 | Address: 0 | Elements: 5 | Value(s): [ 5 5 5 5 5 ]
A,B | Address: 20 | Elements: 3 | Value(s): [ 0 0 0 ]
A,C | Address: 32 | Elements: 1 | Value: 69
CZTERYDZIESI(...) | Address: 36 | Elements: 1 | Value: 4
TD | Address: 40 | Elements: 1 | Value: 1
TD | Address: 44 | Elements: 1 | Value: 4

] Current executed operation: L 3, CZTERYDZIESIATRZYDZIESI
] Operation number: 7
] Press: q to exit or any other key to continue
```

Rysunek 2: Wykonywanie po linijce

## 7 Zakończenie

Wykonanie całego projektu zajęło około 15 godzin. Podczas pracy często występowały problemy związane z łańcuchami znaków czy też wskaźnikami i wskaźnikami do wskaźników, jednak dość szybko były one rozwiązywane. Przez wiele operacji porównania potrzebnych do sprawdzenia poprawności napisanego kodu w pseudo assemblerze, program nie jest zbyt wydajny przy dużej ilości operacji i deklarowanych zmiennych.