
	Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Sekcja
2019/2020	SSI	Języki Asemblerowe	3-4	2
Imię:	Mateusz	Prowadzący:	KH	
Nazwisko:	Chłopek			
<h2><i>Raport końcowy</i></h2>				
<p>Temat projektu:</p> <p style="text-align: center; font-size: 1.2em;">Rozwiązywanie układów równań metodą Seidla.</p>				
Data oddania: dd/mm/rrrr		23/01/2020		

1. Metoda Seidla

a. Algorytm

Użyty przeze mnie algorytm służy do rozwiązywania układów równań liniowych. Polega on na przekształceniu macierzy wejściowych w i następnie wyliczanie w iteracjach kolejnych przybliżeń niewiadomych. Metoda Seidla metodą niedokładną co oznacza, że należy określić warunek stopu obliczeń. Tym warunkiem jest przekroczenie maksymalnej (wcześniej już określonej) maksymalnej liczby iteracji lub osiągnięcie zamierzonej precyzji.

Algorytm przedstawia się w sposób następujący:

Na wejściu otrzymujemy macierz A zawierającą współczynniki niewiadomych w równaniach oraz wektor B wyrazów wolnych. Następnie należy stworzyć macierz alfa, wektor beta oraz wektor X_{akt} gdzie:

$$alfa_{ij} = -\frac{A_{ij}}{A_{ii}}$$

$$alfa_{ii} = 0$$

$$beta_i = \frac{B_i}{A_{ii}}$$

$$X_{akt} = beta$$

Teraz w pętli należy dokonywać przypisania:

$$X_{pop} = X_{akt} \quad \text{oraz} \quad X_{akt} = beta$$

Następnie obliczyć nowe przybliżenia X_{akt} :

$$X_{akt_i} += alfa_{ij} * X_{pop_j}$$

$$\text{dla } j = i + 1 \text{ oraz } j < \text{liczba_niewiadomych}$$

oraz

$$X_{akt_i} += alfa_{ij} * X_{akt_j}$$

$$\text{dla } j = 0 \text{ oraz } j < i$$

Jak już to było wspomniane, należy zapewnić stop obliczeń iteracyjnych. Z tego powodu, w pętli musi znaleźć się jeszcze inkrementowanie licznika iteracji oraz wyliczanie aktualnie osiągniętej precyzji, którą liczy się w następujący sposób:

$$\text{aktualna precyzja} = \sum_{i=0}^n |X_{akt_i} - X_{pop_i}|$$

b. Problemy z metodą Seidla

Metoda Seidla niestety nie jest w stanie rozwiązać każdego układu równań. Wymagane jest, aby elementy na głównej przekątnej [macierzy A] były silnie dominujące. Z tego względu, generator, który napisałem

w celu tworzenia dowolnie dużego układu równań, należało zmienić, tak aby układ był rozwiązywalny metodą Seidla.

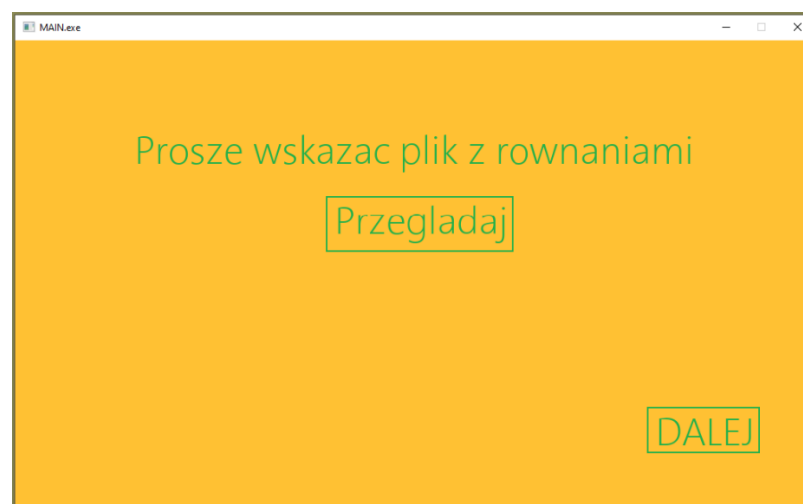
Kolejnym problemem była wielowątkowość. Metoda Seidla nie jest najlepszym wyborem jeśli program ma zostać zrealizowany z wykorzystaniem więcej niż jednego wątku. Związane jest to m. in. ze sposobem w jaki wyliczane są niewiadome. W danej iteracji, n -ta niewiadoma potrzebuje mieć już wyliczone wszystkie „poprzedzające” je niewiadome ($x_1, x_2, x_3, \dots, x_{n-1}$) z aktualnej iteracji co nie jest możliwe, nawet przy założeniu idealnie równoległej, równie szybkiej pracy wątków. Z tego powodu należało wprowadzić synchronizację pracy wątków.

2. Wykorzystanie bibliotek – realizacja w C++ i ASM.

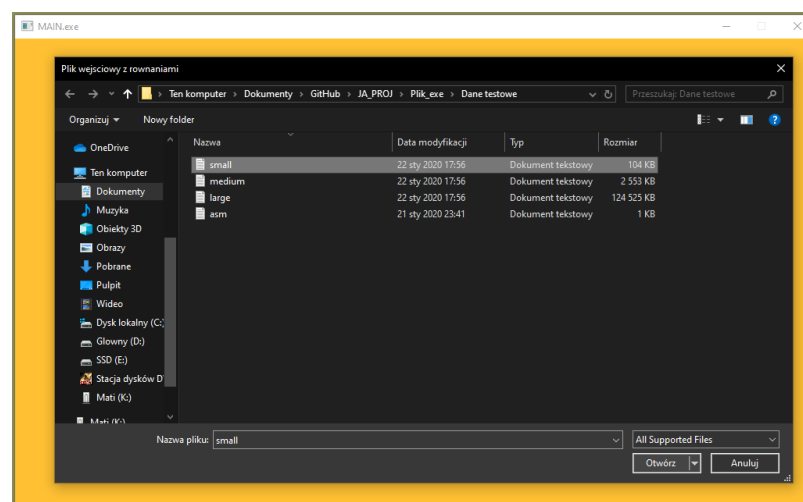
Zastosowany algorytm został całkowicie zrealizowany zarówno w C++ jak i w asemblerze. Program główny wczytuje dane wejściowe (macierz A, wektor B), ma określone parametry (max. liczba iteracji i docelowa precyzja), tworzy wszystkie wymagane zmienne (macierze pośrednie, tablica booli do synchronizacji itp.) i wywołuje funkcję biblioteczną – C++ lub asembler – zależnie od wyboru użytkownika, która to wylicza niewiadome. Ostatecznie program główny zapisuje je do pliku.

3. Działanie programu

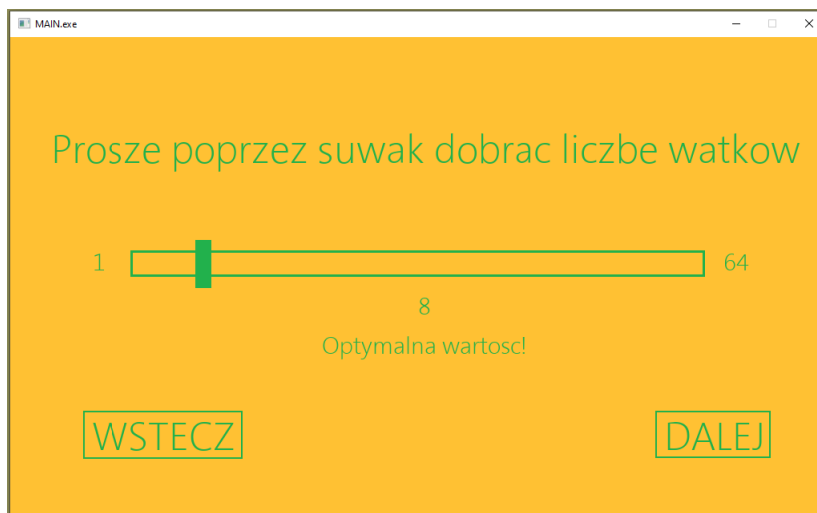
Na poniższych zdjęciach zostało ukazane działanie programu.



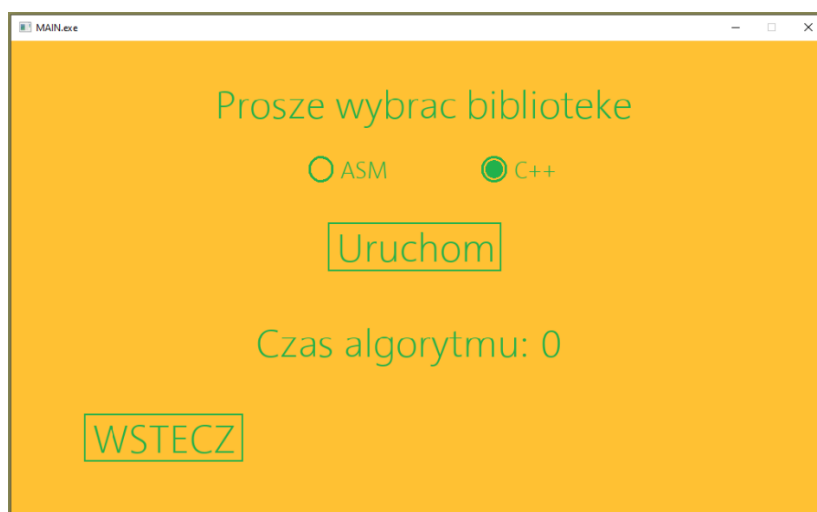
Rys. 1 Uruchomienie programu



Rys. 2 Wskazanie pliku z równaniami po naciśnięciu przycisku „Przeglądaj”



Rys. 3 Wskazanie liczby wątków



Rys. 4 Wybranie biblioteki



Rys. 5 Uruchomienie i wyświetlenie czasu dla biblioteki C++



Rys. 5 Uruchomienie i wyświetlenie czasu dla biblioteki ASM

4. Porównanie czasów C++ i ASM dla trzech zestawów danych.

a. Dla 1 wątku

	C++	ASM
SMALL – 200 równań	2ms	2ms
MEDIUM – 1000 równań	16ms	13ms
LARGE – 7000 równań	850ms	550ms

b. Dla 8 wątków

	C++	ASM
SMALL – 200 równań	27ms	26ms
MEDIUM – 1000 równań	72ms	50ms
LARGE – 7000 równań	1277ms	815ms

c. Dla więcej niż 8 wątków i dla zestawu SMALL

	C++	ASM
SMALL – 200 równań	1.43s	1.21s
MEDIUM – 1000 równań	4,78s	3.77s
LARGE – 7000 równań	21.50s	17.03s

5. Kod w ASM:

```
.data
Aaddr DQ 0
Baddr DQ 0
alfa DQ 0
beta DQ 0
variablesNumber DQ 0
xOld DQ 0
xNew DQ 0
```

```

condition DQ 0
isReady DQ 0
precision REAL4 0.0
maxIterations DQ 0
threadsNumber DQ 0

minus REAL4 -0.0
mutex DWORD 0

.code
SeidelAsm PROC
    ;zmienne lokalne
    LOCAL lowerBound:QWORD
    LOCAL upperBound:QWORD
    LOCAL boolCondition:QWORD
    LOCAL counter:QWORD

    ;POBRANIE ARGUMENTÓW
    MOV Aaddr, RCX
    MOV Baddr, RDX
    MOV alfa, R8
    MOV beta, R9

    MOV EAX, DWORD PTR [RSP + 80]
    MOV variablesNumber, RAX

    MOV RAX, QWORD PTR [RSP + 88]
    MOV xOld, RAX

    MOV RAX, QWORD PTR [RSP + 96]
    MOV xNew, RAX

    MOV EAX, DWORD PTR [RSP + 104]
    MOV lowerBound, RAX

    MOV EAX, DWORD PTR [RSP + 112]
    MOV upperBound, RAX

    MOV RAX, QWORD PTR [RSP + 120]
    MOV condition, RAX

    MOV RAX, QWORD PTR [RSP + 128]
    MOV isReady, RAX

    MOVSS XMM0, DWORD PTR [RSP + 136]
    MOVSS precision, XMM0

    MOV EAX, DWORD PTR [RSP + 144]
    MOV maxIterations, RAX

    MOV counter, 0

    ;zabezpieczenie rejestrów
    PUSH RBX

    ;ROZPOCZĘCIE GŁÓWNEJ CZĘŚCI FUNKCJI

    ;PĘTLA FOR
    ;PRZYPIISANIE i = lowerBound
    MOV RAX, lowerBound
    MOV R8, RAX

LOOP1:
    ;PORÓWNANIE i < upperBound
    MOV RAX, R8
    CMP RAX, upperBound
    JE END1

    ;PRZYPIISANIE divisor = A[i][i]
    MOV RCX, R8
    MOV RDX, QWORD PTR [Aaddr]
    MOV RAX, QWORD PTR [RDX + RCX * 8]
    MOVSS XMM0, DWORD PTR [RAX + RCX * 4]
    MOVSS XMM1, minus
    XORPS XMM0, XMM1
    MOVSS XMM2, XMM0

    ;ZAGNIEŻDŻONA PĘTLA FOR
    ;PRZYPIISANIE j = 0
    MOV R9, 0

LOOP2:
    ;PORÓWNANIE j < variablesNumber
;dolna granica przedziału dla wątku
;górna granica przedziału dla wątku
;zmienna wykorzystywana przy synchronizacji
;licznik pętli

;macierz A
;wektor B
;macierz alfa
;wektor beta

;liczba niewiadomych

;X poprzedniej iteracji

;X nowej iteracji

;dolna granica przedziału dla wątku

;górna granica przedziału dla wątku

;suma wyliczana do precyzji

;tablica booli wykorzystywana do synchronizacji

;ustalony poziom precyzji

;maksymalna liczba iteracji

;wyzerowanie lokalnego licznika pętli

```

```

MOV RAX, R9
CMP RAX, variablesNumber
JE END2

;A[i][j] -> XMM0
MOV RAX, R8
MOV RCX, R9
MOV RDX, QWORD PTR [Aaddr]
MOV RAX, QWORD PTR [RDX + RAX * 8]
MOVSS XMM0, DWORD PTR [RAX + RCX * 4]
; XMM0 / XMM2
DIVSS XMM0, XMM2
; alfa[i][j] = XMM0 (= -A[i][j] / XMM2)
MOV RAX, R8
MOV RCX, R9
MOV RDX, QWORD PTR [alfa]
MOV RAX, QWORD PTR [RDX + RAX * 8]
MOVSS DWORD PTR [RAX + RCX * 4], XMM0
INC R9
JMP LOOP2

END2:
;beta[i] = B[i] / XMM2
;xNew[i] = beta[i]
MOVSS XMM1, minus
XORPS XMM2, XMM1
MOV RAX, R8
MOV RCX, QWORD PTR [beta]
MOV RDX, QWORD PTR [Baddr]
MOVSS XMM0, DWORD PTR [RDX + RAX * 4]
DIVSS XMM0, XMM2
MOVSS DWORD PTR [RCX + RAX * 4], XMM0
MOV RCX, QWORD PTR [xNew]
MOVSS DWORD PTR [RCX + RAX * 4], XMM0

;alfa[i][i] = 0
MOV RAX, R8
MOV RCX, R8
MOV RDX, QWORD PTR [alfa]
MOV RAX, QWORD PTR [RDX + RAX * 8]
PXOR XMM0, XMM0
MOVSS DWORD PTR [RAX + RCX * 4], XMM0

;isReady[0][i] = true
MOV RAX, R8
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 0]
MOV BYTE PTR [RDX + RAX], 1
INC R8
JMP LOOP1

END1:

;synchronizacja wątków
SynchLoop0:
MOV RAX, boolCondition
CMP RAX, 1
JE SynchLoop0End
MOV boolCondition, 1
MOV R8, 0
ForLoop0:
MOV RAX, r8
CMP RAX, variablesNumber
JE ForLoop0End
INC R8
;sprawdzenie czy obliczenia dla danej niewiadomej zostały zakończone
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 0]
MOV AL, BYTE PTR [RDX + RAX]
CMP AL, 1
JE ForLoop0
MOV boolCondition, 0
ForLoop0End:
JMP SynchLoop0
SynchLoop0End:
MOV boolCondition, 0

;rozpoczęcie pętli DO WHILE
DOLOOP:
INC counter
;pierwsza pętla FOR
MOV RAX, lowerBound
MOV R8, RAX

```

LOOP3:

```
MOV RAX, R8
CMP RAX, upperBound
JE ENDLOOP3
;sprawdzanie czy i + 4 < upperBound
ADD RAX, 4
CMP RAX, upperBound
JA Rest
;instrukcje wektorowe, jeśli wątek może przetworzyć jeszcze min. 4 niewiadome

;isReady[1][i] = false
;isReady[2][i] = false
MOV RAX, R8
MOV RDX, QWORD PTR [isReady]

MOV RCX, QWORD PTR [RDX + 8]
MOV BYTE PTR [RCX + RAX], 0
MOV BYTE PTR [RCX + RAX + 1], 0
MOV BYTE PTR [RCX + RAX + 2], 0
MOV BYTE PTR [RCX + RAX + 3], 0

MOV RCX, QWORD PTR [RDX + 16]
MOV BYTE PTR [RCX + RAX], 0
MOV BYTE PTR [RCX + RAX + 1], 0
MOV BYTE PTR [RCX + RAX + 2], 0
MOV BYTE PTR [RCX + RAX + 3], 0

;xOld[i] = xNew[i]
;przesłanie 4 floatów
MOV RDX, QWORD PTR [xNew]
MOV RCX, QWORD PTR [xOld]
MOVUPS XMM0, [RDX + RAX * 4]
MOVUPD [RCX + RAX * 4], XMM0

;isReady[1][i] = true
MOV RDX, QWORD PTR [isReady]
MOV RCX, QWORD PTR [RDX + 8]
MOV BYTE PTR [RCX + RAX], 1
MOV BYTE PTR [RCX + RAX + 1], 1
MOV BYTE PTR [RCX + RAX + 2], 1
MOV BYTE PTR [RCX + RAX + 3], 1

ADD R8, 4
JMP LOOP3
```

;jeśli wątek przetworzy jeszcze mniej niż 4 niewiadome to bez instrukcji wektorowych

Rest:

```
MOV RAX, R8

;xOld[i] = xNew[i]
MOV RDX, QWORD PTR [xNew]
MOV RCX, QWORD PTR [xOld]
MOVSS XMM0, DWORD PTR [RDX + RAX * 4]
MOVSS DWORD PTR [RCX + RAX * 4], XMM0

;isReady[1][i] = true
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 8]
MOV BYTE PTR [RDX + RAX], 1

INC R8
JMP LOOP3
```

ENDLOOP3:

;synchronizacja wątków

SynchLoop1:

```
MOV RAX, boolCondition
CMP RAX, 1
JE SynchLoop1End
MOV boolCondition, 1
MOV R8, 0
```

ForLoop1:

```
MOV RAX, r8
CMP RAX, variablesNumber
JE ForLoop1End
INC R8
;sprawdzenie czy obliczenia dla danej niewiadomej zostały zakończone
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 8]
MOV AL, BYTE PTR [RDX + RAX]
CMP AL, 1
JE ForLoop1
MOV boolCondition, 0
```



```

ForLoop1End:
    JMP SynchLoop1
SynchLoop1End:
    MOV boolCondition, 0

;druga pętla FOR
MOV RAX, lowerBound
MOV R8, RAX

LOOP4:
    MOV RAX, R8
    CMP RAX, upperBound
    JE ENDLOOP4

    ;isReady[1][i] = false
    ;isReady[2][i] = false
    MOV RAX, R8
    MOV RDX, QWORD PTR [isReady]
    MOV RCX, QWORD PTR [RDX + 16]
    MOV BYTE PTR [RCX + RAX], 0
    MOV RCX, QWORD PTR [RDX + 24]
    MOV BYTE PTR [RCX + RAX], 0

    ;xNew[i] = beta[i]
    MOV RAX, R8
    MOV RDX, QWORD PTR [beta]
    MOVSS XMM0, DWORD PTR [RDX + RAX * 4]
    MOV RDX, QWORD PTR [xNew]
    MOVSS DWORD PTR [RDX + RAX * 4], XMM0

    ;pierwsza zagnieżdżona pętla FOR
    ;j = i + 1
    MOV RAX, R8
    INC RAX
    MOV R9, RAX

LOOP5:
    MOV RAX, R9
    CMP RAX, variablesNumber
    JE ENDLOOP5
    MOV RAX, R8
    MOV RBX, R9

    ;RCX = alfa[i]
    MOV RCX, QWORD PTR [alfa]
    MOV RCX, QWORD PTR [RCX + RAX * 8]

    ;RDX = xOld
    MOV RDX, QWORD PTR [xOld]
    MOVSS XMM0, DWORD PTR [RCX + RBX * 4]
    MOVSS XMM1, DWORD PTR [RDX + RBX * 4]
    MULSS XMM0, XMM1

    ;xNew[i] += XMM0 (=alfa[i][j] * xOld[j])
    MOV RBX, R8
    MOV RCX, QWORD PTR [xNew]
    ADDSS XMM0, DWORD PTR [RCX + RBX * 4]
    MOVSS DWORD PTR [RCX + RBX * 4], XMM0
    INC R9
    JMP LOOP5

ENDLOOP5:

;druga zagnieżdżona pętla FOR
MOV R9, 0

LOOP6:
    MOV RAX, R9
    CMP RAX, R8
    JE ENDLOOP6

    ;przygotowanie tablicy isReady[2]
    MOV RCX, QWORD PTR [isReady]
    MOV RCX, QWORD PTR [RCX + 16]
    MOV RBX, R9

    ;pętla WHILE
    XOR RDX, RDX

LOOP7:
    ;oczekiwanie na zgłoszenie gotowości obliczeń dla danej niewiadomej
    MOV DL, BYTE PTR [RCX + RBX]
    CMP DL, 0
    JE LOOP7

```

ENDLOOP7:

```
;xNew[i] += alfa[i][j] * xNew[j]
MOV RAX, R8
MOV RBX, R9

;RCX = alfa[i]
MOV RCX, QWORD PTR [alfa]
MOV RCX, QWORD PTR [RCX + RAX * 8]

;RDX = xNew
MOV RDX, QWORD PTR [xNew]
MOVSS XMM0, DWORD PTR [RCX + RBX * 4]
MOVSS XMM1, DWORD PTR [RDX + RBX * 4]
MULSS XMM0, XMM1

;xNew[i] += XMM0 (=alfa[i][j] * xNew[j])
MOV RBX, R8
MOV RCX, QWORD PTR [xNew]
ADDSS XMM0, DWORD PTR [RCX + RBX * 4]
MOVSS DWORD PTR[RCX + RBX * 4], XMM0
INC R9
JMP LOOP6
```

ENDLOOP6:

```
;isReady[2][i] = true
MOV RAX, R8
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 16]
MOV BYTE PTR [RDX + RAX], 1
INC R8
JMP LOOP4
```

ENDLOOP4:

```
;synchronizacja wątków
```

SynchLoop2:

```
MOV RAX, boolCondition
CMP RAX, 1
JE SynchLoop2End
MOV boolCondition, 1
MOV R8, 0
```

ForLoop2:

```
MOV RAX, r8
CMP RAX, variablesNumber
JE ForLoop2End
INC R8
;sprawdzenie czy obliczenia dla danej niewiadomej zostały zakończone
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 16]
MOV AL, BYTE PTR [RDX + RAX]
CMP AL, 1
JE ForLoop2
MOV boolCondition, 0
```

ForLoop2End:

```
JMP SynchLoop2
```

SynchLoop2End:

```
MOV boolCondition, 0
```

```
;trzecia pętla FOR
```

```
MOV RAX, lowerBound
MOV R8, RAX
```

LOOP8:

```
MOV RAX, R8
CMP RAX, upperBound
JE ENDLOOP8
```

```
;isReady[1][i] = false
```

```
;isReady[3][i] = false
```

```
MOV RAX, R8
MOV RDX, QWORD PTR [isReady]
MOV RCX, QWORD PTR [RDX + 8]
MOV BYTE PTR [RCX + RAX], 0
MOV RCX, QWORD PTR [RDX + 24]
MOV BYTE PTR [RCX + RAX], 0
```

```
;obliczanie sumy do sprawdzenia precyzji
```

```
;condition[i] = 0
```

```
MOV RAX, R8
MOV RDX, QWORD PTR [condition]
MOV DWORD PTR [RDX + RAX * 4], 0
;condition[i] += abs(xNew[i] - xOld[i])
```

```

MOV RCX, QWORD PTR [xNew]
MOVSS XMM0, DWORD PTR [RCX + RAX * 4]
MOV RCX, QWORD PTR [xOld]

MOVSS XMM1, DWORD PTR [RCX + RAX * 4]
SUBSS XMM0, XMM1
XORPS XMM1, XMM1
COMISS XMM0, XMM1
JNB SKIPABS
MOVSS XMM1, minus
XORPS XMM0, XMM1

```

SKIPABS:

```

;condition[i] /= variablesNumber
CVTSI2SS XMM1, variablesNumber
DIVSS XMM0, XMM1
MOVSS DWORD PTR [RDX + RAX * 4], XMM0

;isReady[3][i] = true
MOV RAX, R8
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 24]
MOV BYTE PTR [RDX + RAX], 1
INC R8
JMP LOOP8

```

ENDLOOP8:

```

;synchronizacja wątków

```

SynchLoop3:

```

MOV RAX, boolCondition
CMP RAX, 1
JE SynchLoop3End
MOV boolCondition, 1
MOV R8, 0

```

ForLoop3:

```

MOV RAX, r8
CMP RAX, variablesNumber
JE ForLoop3End
INC R8
;sprawdzenie czy obliczenia dla danej niewiadomej zostały zakończone
MOV RDX, QWORD PTR [isReady]
MOV RDX, QWORD PTR [RDX + 24]
MOV AL, BYTE PTR [RDX + RAX]
CMP AL, 1
JE ForLoop3
MOV boolCondition, 0

```

ForLoop3End:

```

JMP SynchLoop3

```

SynchLoop3End:

```

MOV boolCondition, 0

```

```

;czwarta pętla FOR - sumowanie składników sumy precyzji
PXOR XMM3, XMM3
MOV R8, 0

```

LOOP9:

```

MOV RAX, R8
CMP RAX, variablesNumber
JE ENDLOOP9
;XMM3 += condition[i]
MOV RAX, R8
MOV RDX, QWORD PTR [condition]
MOVSS XMM1, DWORD PTR [RDX + RAX * 4]
ADDSS XMM1, XMM3
MOVSS XMM3, XMM1
INC R8
JMP LOOP9

```

ENDLOOP9:

```

;warunek pętli DO WHILE

```

```

;precyzja
MOVSS XMM0, XMM3
MOVSS XMM1, precision
CMPLSS XMM0, XMM1
CVTSS2SI RAX, XMM0
CMP RAX, 0
JNZ ENDDOLOOP

```

```

;liczba iteracji
MOV RAX, counter
CMP RAX, maxIterations

```

```

JAE ENDDOLOOP
JMP DOLOOP

;koniec pętli DO-WHILE -> koniec obliczeń
ENDDOLOOP:

;przywrócenie stanu rejestrów
POP RBX

RET
SeidelAsm ENDP
END

```

6. Instrukcje wektorowe

W programie instrukcje wektorowe są wykorzystywane w dla: $xOld[i] = xNew[i]$

Zrealizowane jest to w sposób następujący:

```

MOV RDX, QWORD PTR [xNew]
MOV RCX, QWORD PTR [xOld]
MOVUPS XMM0, [RDX + RAX * 4]
MOVUPD [RCX + RAX * 4], XMM0

```

Polega to na:

- Przesłaniu do RDX adresu tablicy xNew, a do RCX adresu tablicy xOld.
- Przesłaniu do XMM0 4 floatów z tablicy xNew
- Przesłaniu do xOld 4 floatów z XMM0

Wcześniej jest jeszcze sprawdzenie czy wątek ma do przetworzenia jeszcze przynajmniej 4 zmienne. W przeciwnym wypadku instrukcje wektorowe nie są wykonywane.

7. Wnioski

Napisanie programu pozwoliło mi zrozumieć pewne rzeczy. Po pierwsze to, że użycie wielu wątków nie zawsze jest korzystne dla czasu działania programu. Niektóre algorytmy – metoda Seidla – nie jest łatwo zamienić tak, aby obliczenia równoległe je przyspieszyły. Po drugie, napisanie funkcji w assemblerze nie jest wcale takie trudne, większość mojej procedury opierała się na zaledwie kilku rozkazach: MOV, ADD, INC, CMP, JMP (i ich wariacjach np. MOVSS, ADDSS, JNE itp.). Bardzo ciekawym i efektywnym rozwiązaniem jest stosowanie instrukcji wektorowych. W moim programie użyłem je zaledwie w jednym miejscu, lecz dopiero po ich zastosowaniu udało się osiągnąć lepszy czas, niż dla funkcji w C++.