# Table of Contents

**1. Laboratorium Języków Assemblerowych** ..... 2
***1.1. Ćwiczenie 1*** ..... 2
1.1.1. Przygotowanie: Środowisko sprzętowe i programistyczne ..... 2
1.1.1.1. Instalacja Visual Studio (Community Edition 2019) ..... 3
1.1.2. Wymagania ..... 4
1.1.3. Wstęp ..... 4
1.1.4. Powtórka z teorii: Funkcje i ich wywoływanie z poziomu assemblera ..... 5
1.1.4.1. Standardy wywoływania funkcji ..... 5
1.1.5. Praktyka: Tworzenie projektu C++ i konsersja do ASM ..... 7
1.1.5.1. Tworzenie pliku z kodem źródłowym ..... 7
1.1.5.2. Konwersja projektu z C++ na ASM ..... 8
1.1.5.3. Utworzenie punktu startowego oraz uzupełnienie bibliotek ..... 10
1.1.5.4. Oznaczenie pliku ASM do kompilacji ..... 12
1.1.5.5. Instalacja dodatku kolorującego kod źródłowy w assemblerze ..... 12
1.1.6. Praktyka: Zadanie do wykonania ..... 13
1.1.6.1. Fragmenty kodu do wykorzystania ..... 13
1.1.7. Debuggowanie i uruchamianie ..... 17
1.1.8. Metoda weryfikacji wyników ..... 19
1.1.9. Pomocne zasoby ..... 19

# 1. Laboratorium Języków Assemblerowych

## 1.1. Ćwiczenie 1

Laboratorium języków assemblerowych, ćwiczenie 1

### 1.1.1. Przygotowanie: Środowisko sprzętowe i programistyczne

Do wykonania ćwiczenia niezbędny jest komputer PC w postaci maszyny fizycznej lub wirtualnej z systemem Windows 7, 8, 8.1 lub 10, z zainstalowanym i skonfigurowanym środowiskiem programistycznym Microsoft Visual Studio 2019 w wrsji co najmniej Community Edition [1)]. Do pracy w laboratorium nada się również wersja VS Professional lub Enterprise. Proszę natomiast nie pomylić z Visual Studio Code, to zupełnie inny produkt, który nie nadaje się wprost do wykorzystania w naszym laboratorium do realizacji zadań, w szczególności kolejnych ćwiczeń.



Jednym z najprostszych rozwiązań do przygotowania środowiska laboratoryjnego jest wykorzystanie gotowej maszyny wirtualnej, która jest dostępna na stronach Microsoft. Można ją wykorzystać w celach testowych, w okresie nie dłuższym niż 30-90dni (zależnie od daty pobrania i wersji). Dostępne są maszyny wirtualne dla różnych platform wirtualizacyjnych.

W przypadku chęci skorzystania z gotowej maszyny, obraz można pobrać m.in. z poniższego linku:

- Opcja 1: Windows 10 + Visual Studio 2019 Enterprise [2)];
- Opcja 2: Windows 10, a następnie własnoręczna instalacja Visual Studio 2019 Community[3)];

Jako środowisko witualizacyjne można wykorzystać Oracle Virtual Box[4)], dostępne praktycznie na każdą platformę desktopową. W przypadku instalacji Linuxowych typu headless, można wykorzystać KVM[5)].



Maszyny wirtualne z pewnością będą wymagały zainstalowania aktualizacji Windows i aplikacji, w tym kilkukrotnych restartów, zatem instalację należy wykonać zawczasu i doprowadzić ją do stanu stabilnego PRZED zajęciami.

Proszę rownież pamiętać o zainstalowaniu dodatków specyficznych dla platformy wirutalizacyjnej (np, Oracle Virtual Box Add-ons), co może pociągać za sobą konieczności w zmianie konfiguracji maszyny wirtualnej (np. konieczność dołożenia napędu wirtualnego w celu zainstalowania ww dodatków). Sugeruję rownież zwiększenie pamięci RAM do co najmniej 8GB (o ile to możliwe), liczby dostępnych wątków procesora (przynajmniej 2) oraz zmianę wirtualnej karty graficznej na zgodną z akceleracją sprzętową DirectX/Direct 3D (np. w Oracle Vitual Box jest to karta VBoxSVGA).



Niezależnie od przyjętego modelu należy pamiętać, że zajęcia wymagają 64-bitowego

systemu operacynego.

### 1.1.1.1. Instalacja Visual Studio (Community Edition 2019)

Instalator Visual Studio Communicy Edition 2019 pozwala na zainstalowanie wielu różnych komponentów. Minimalny zestaw komponentów został przedstawiony na poniższych rysunkach 1, 2, 3:

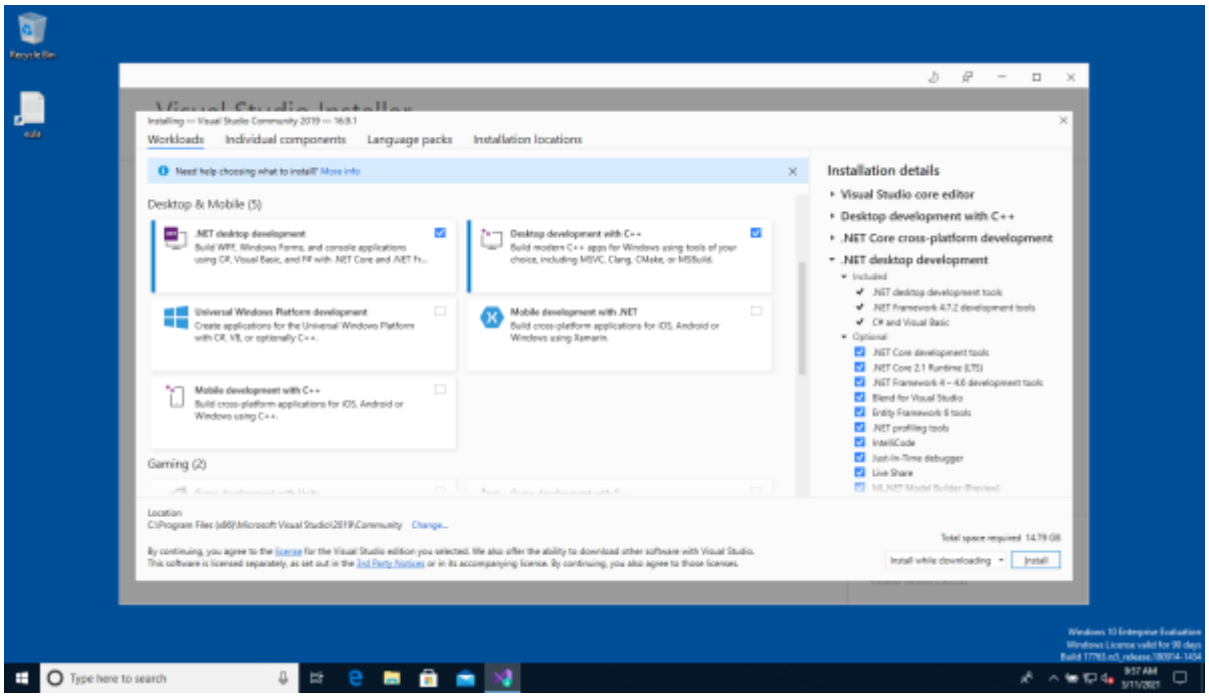



Fig. 1: Opcje instalacyjne część 1

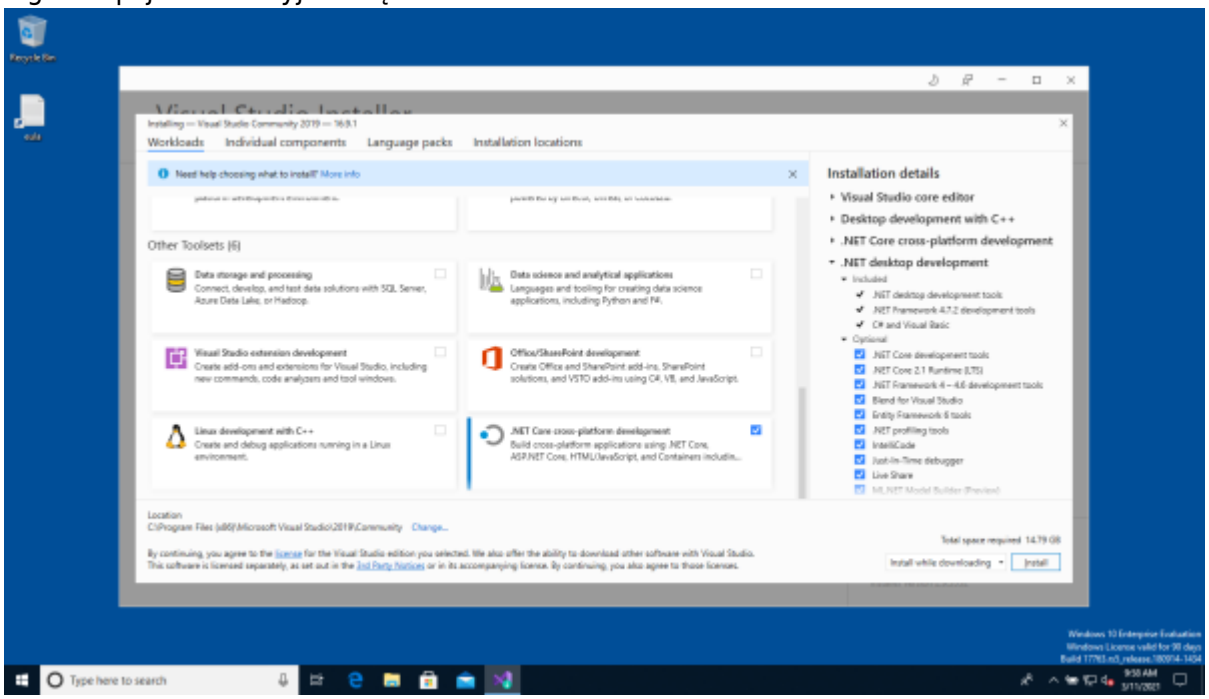



Fig. 2: Opcje instalacyjne część 2

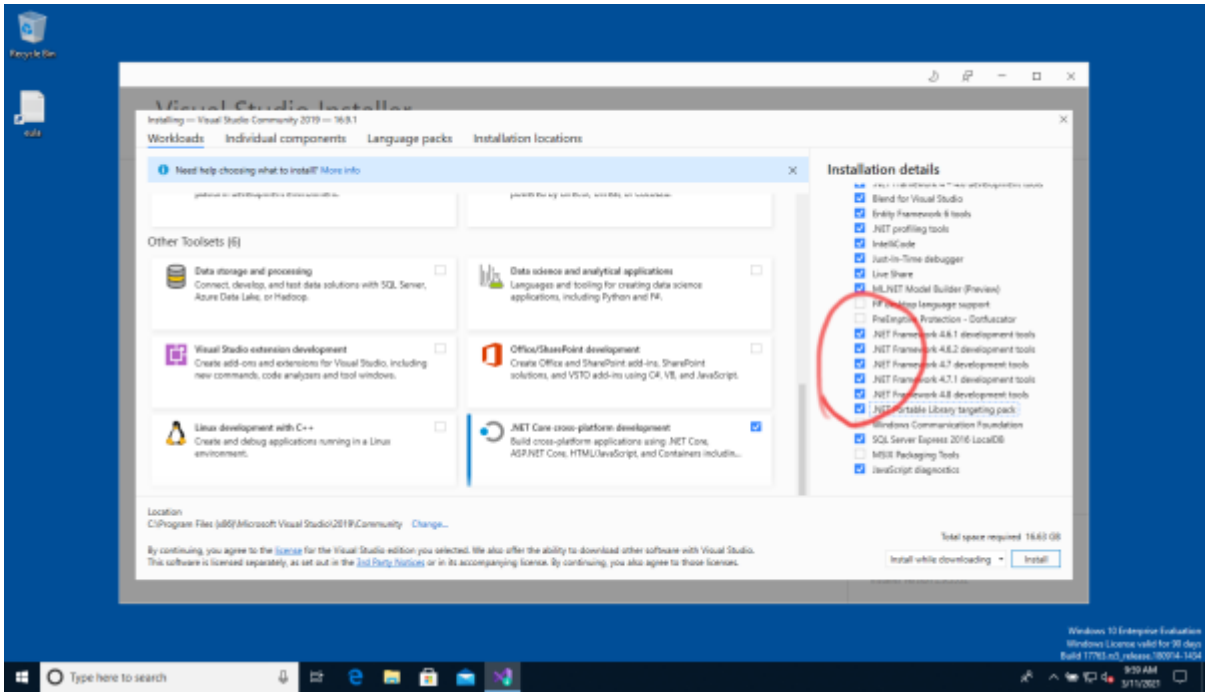


Fig. 3: Opcje instalacyjne część 3 - uwaga - frameworki .NET po prawej stronie trzeba zaznaczyć ręcznie!

### 1.1.2. Wymagania

Studenci muszą rozumieć sposób tworzenia i działania aplikacji typu “Command Line”, podstawy języka assembler, C++ i C#

### 1.1.3. Wstęp

W ramach tego ćwiczenia stworzysz w czystym assemblerze dla x64 (procesory Intel/AMD, 64 bity) aplikację konsolową, która będzie prowadziła interakcję z systemem operacyjnym Windows 10 (7, 8, 8.1) wywołując jego funkcje poprzez interfejs Windows API.

Jest to przykład technologiczny, ponieważ tworzenie w ten sposób skomplikowanych aplikacji i złożonych interfejsów użytkownika jest ... bezsensowne 😬, choć możliwe. Natomiast zdedycowanie przyda się przy robieniu prostych usług obliczeniowych, narzędzi, etc., które wymagają zwartości kodu i wykonywania zadań z maksymalną prędkością.
Zwróć uwagę, że podane przykłady nie ograniczają się do funkcji interfejsu użytkownika, a obejmują całe Windows API. **A ze względu na rozmiar aplikacji, całość niejednokronie mieści się w pamięci L1 procesora, zatem wykonuje się niespotykanie szybko.**

Ponieważ w Visual Studio nie ma gotowego modelu projektu na potrzeby aplikacji w czystym assemblerze, wykorzystamy model aplikacji konsolowej w języku C++ i przezbroimy go na potrzeby kompilacji zamiast kodu w języku C++, kodu w czystym assemblerze.

Takie podejście wymaga ręcznego przekonfigurowania kilku pramatetrów we właściwościach projekt, a w szczególności:

- zmiany kompilatora na `masm`;
- zmiany docelowej platformy na x64;
- podania nazwy głównej funkcji jako punktu wejścia ([Entry Point]) tak, aby sytem operacyjny

wiedział, którą funkcję ma zawołać po uruchomieniu aplikacji - w wielu językach i środowiskach programowania jest to domyślnie funkcja `main()`, natomiast w przypadku assemblera, nie ma takiej reguły i trzeba podać samemu, o którą funkcję chodzi;
- uzupełnienia brakujących bibliotek: od wersji 2019 środowiska (w szczególności pod Windows 10) cześć standardowych funkcji dostępnych uprzednio domyślnie w bibliotekach linkowanych do każdego projektu, aktualnie zostało wyłączonych i trzeba je jawnie wskazać; uwaga - dokładny skład bibliotek jest zależny oczywiście od tego, co jest wykorzystywane (wywoływane) w kodzie źródłowym - w środowisku produkcyjnym należy zoptymalizować kod poprzez usunięcie zbędnych bibliotek;



Uwaga! Ponieważ wykorzystujemy pliki `asm`, które natywnie nie są rozpoznawane przez Visual Studio jako pliki z kodem źródłowym, nie są one automatycznie zapisywane w momencie wywołania kompilacji, jak ma to miejsce w przypadku plików `.cpp`, `.cs` itp. Co prawda istnieje możliwość nauczenia VS 2019, aby plik był uznawany za źródłowy, niemniej jednak wykracza to poza niniejszą instrukcję. Proszę zatem pamiętać za każdym razem kliknąć zapis do pliku (Ctrl+s) przed kompilacją.

## 1.1.4. Powtórka z teorii: Funkcje i ich wywoływanie z poziomu assemblera

W przypadku pisania kodu w języku assembler, każdy bajt musi być przydzielony, zadeklarowany i użytkowany w odpowiedni sposób, szczególnie w kontekście wywoływania funkcji Windows API. Ponieważ pracujemy na systemie 64.-bitowym, konieczne jest stosowane reguł wywołania funkcji zgodnie z kanonem.

### 1.1.4.1. Standardy wywoływania funkcji

Parametry wywołania funkcji przekazujemy poprzez rejestry, o ile ich liczba nie przekracza czterech. W przeciwnym wypadu przekazujemy je przez stos (pozostałe). Natomiast obowiązkowo należy zaalokować na stosie przestrzeń na argumenty nawet ... jeśli przekazujemy je przez rejestry 🙂[6].

#### 1.1.4.1.1. Przekazanie argumentów całkowitych

W przypadku, gdy do funkcji przekazujemy argumenty numeryczne całkowite, w szczególności np. adresy i wskaźniki, należy użyć w kolejności następujących rejestrów:

1. argument 1 → RCX,
2. argument 2 → RDX,
3. argument 3 → R8,
4. argument 4 → R9,
5. każdy kolejny argument umieszczamy na stosie z zachowaniem szesnastobitowego wyrównania danych (z przyczyn wydajnościowych).

Przykładowy kod w C++:

```
func1(int a, int b, int c, int d, int e, int f);
```

```
// a w RCX, b w RDX, c w R8, d w R9, f, a następnie e, są odkładane na
stosie
```

#### 1.1.4.1.2. Przekazywanie argumentów zmiennoprzecinkowych

W przypadku przekazywania argumentów zmiennoprzecinkowych, kolejne wartości ładujemy do rejestrów XMM:

1. argument 1 → XMM0,
2. argument 2 → XMM1,
3. argument 3 → XMM2,
4. argument 4 → XMM3,
5. analogicznie jak dla stałoprzecinkowych.

Przykładowy kod w C++:

```
func2(float a, double b, float c, double d, float e, float f);
// a jest XMM0, b w XMM1, c w XMM2, d w XMM3, f a następnie e są odkładane
na stosie
```

#### 1.1.4.1.3. Przekazywanie argumentów mieszanych

W przypadku mieszania argumentów zmiennoprzecinkowych z dziesiętnymi, stosuje się system pozycyjny w zależności od typu, a zatem jeśli przekazujemy np, int1, float1, float2, int2, to wartości należy umieścić w rejestrach (w kolejności):

1. int1 → RCX,
2. float1 → XMM1,
3. float2 → XMM2,
4. int2 → R9,
5. piąty i następny argument poprzez stos.

Przykładowy kod w C++:

```
func3(int a, float b, float c, int d, int e, float f);
// a w RCX, b w XMM1, c w XMM2, d w R9, f, a następnie e, poprzez stos
```



Rejestry po wywołaniu funkcji ulegają zniszczeniu (zmianie wartości), zatem nie należy oczekiwać, że po wywołaniu funkcji, rejestry RCX, RDX, R8, R9, XMM0-XMM3 będą miały wartości takie, jak przed wywołaniem funkcji!

#### 1.1.4.1.4. Zwrot wartości z funkcji

Wartość całkowita (w tym wskaźnik) zwracana jest w rejestrze RAX.

Wartości zmiennoprzecinkowa zwracana jest w rejestrze XMM0.

## 1.1.5. Praktyka: Tworzenie projektu C++ i konsersja do ASM

Stwórz nowy projekt: aplikacja typu command-line (konsolowa) dla języka C++, natywnego (nie C++ .NET) rysunek 4

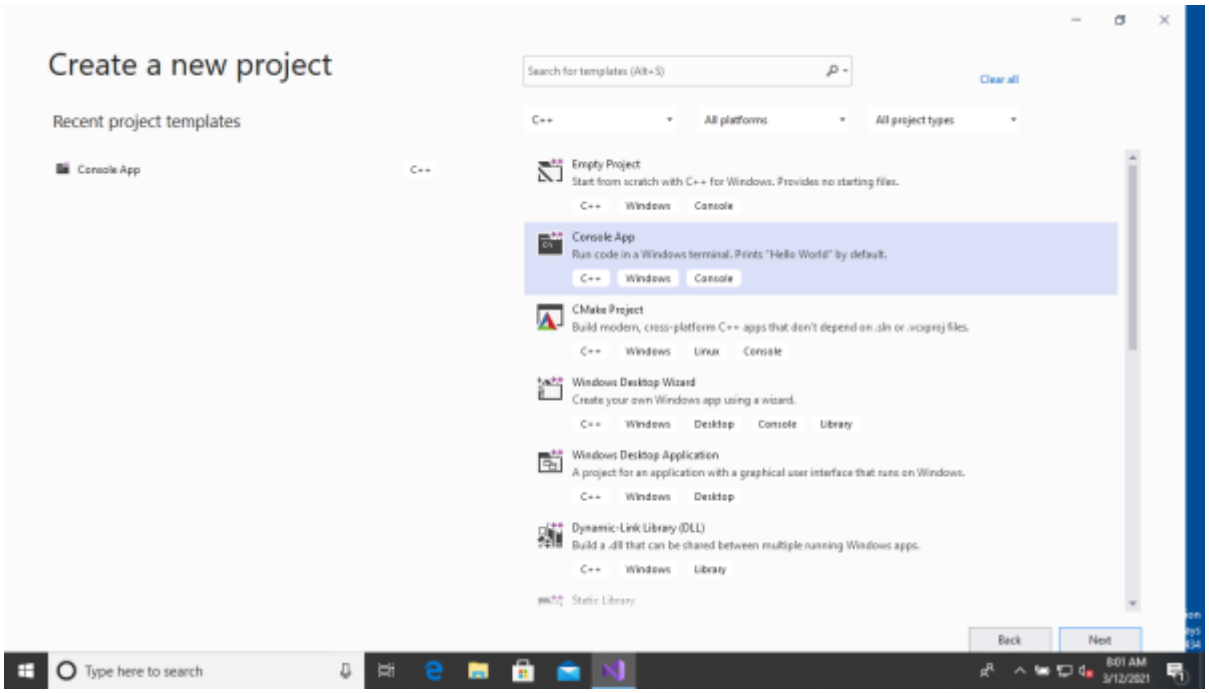


Fig. 4: Tworzenie projektu typu C++, kod natywny

### 1.1.5.1. Tworzenie pliku z kodem źródłowym

Usuń wygenerowany plik .cpp (z poziomu VS) a następnie dodaj nowy plik .asm (można to zrobić klikając w projekt (nie w solucję!) i wybierając dodanie nowego elementu rysunek 5, a następnie wybrać plik typu tekstowego i zmienić jego nazwę oraz rozszerzenie rysunek 6:

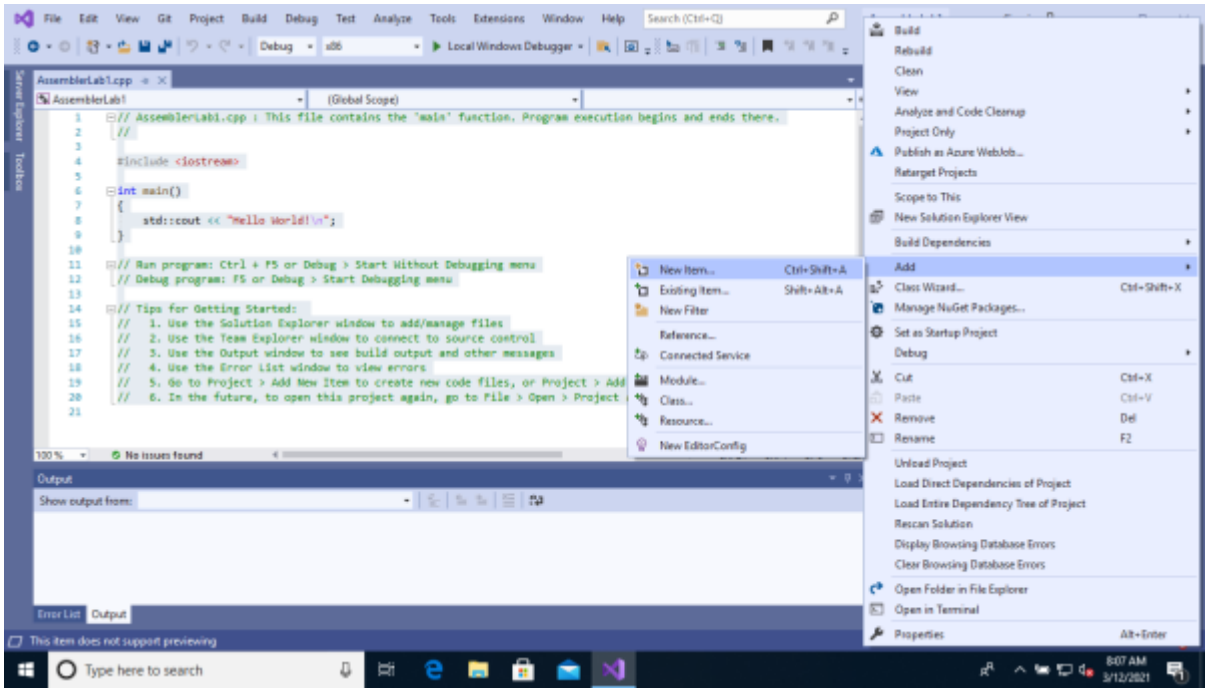


Fig. 5: Dodanie nowego elementu

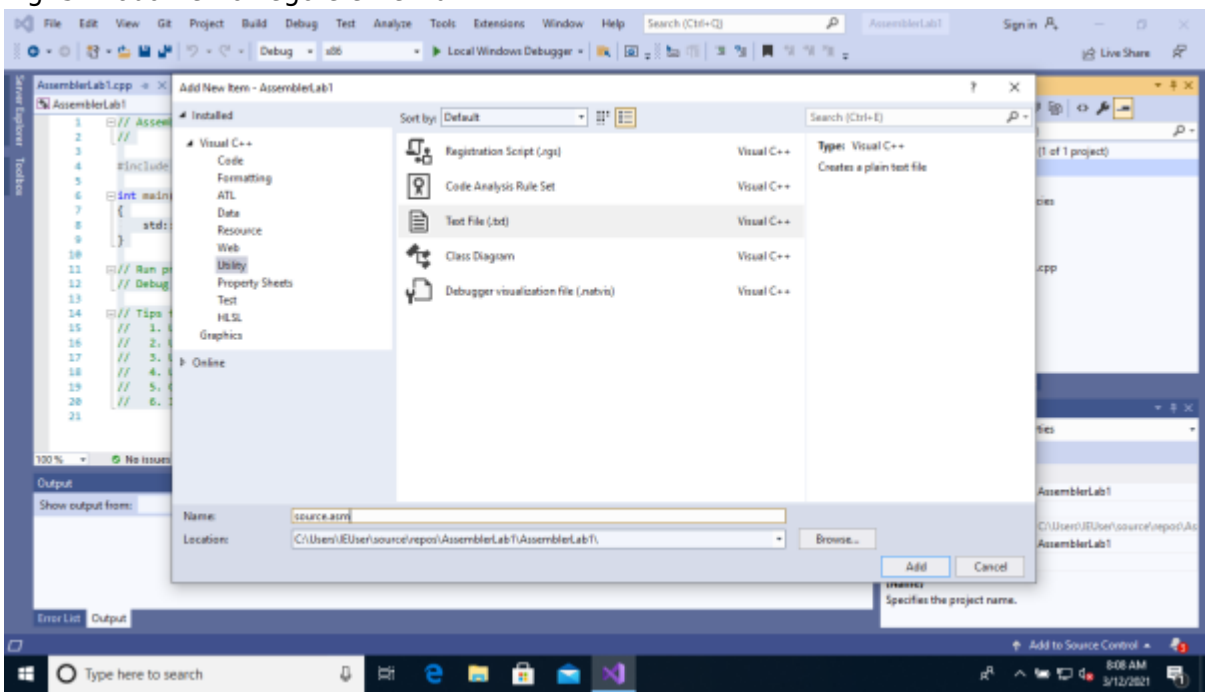


Fig. 6: Dodanie nowego elementu - plik asm

### 1.1.5.2. Konwersja projektu z C++ na ASM

W celu konwersji należy poinformować VS, że pliki źródłowe będą kompilowane za pomocą MASM a nie kompilatora C/C++. W tym celu klikamy prawym przyciskiem na projekt i wybieramy opcję dostosowania budowania projektu rysunek 7. W ustawieniach należy wybrać `masm(.targets, .props)` rysunek 8.

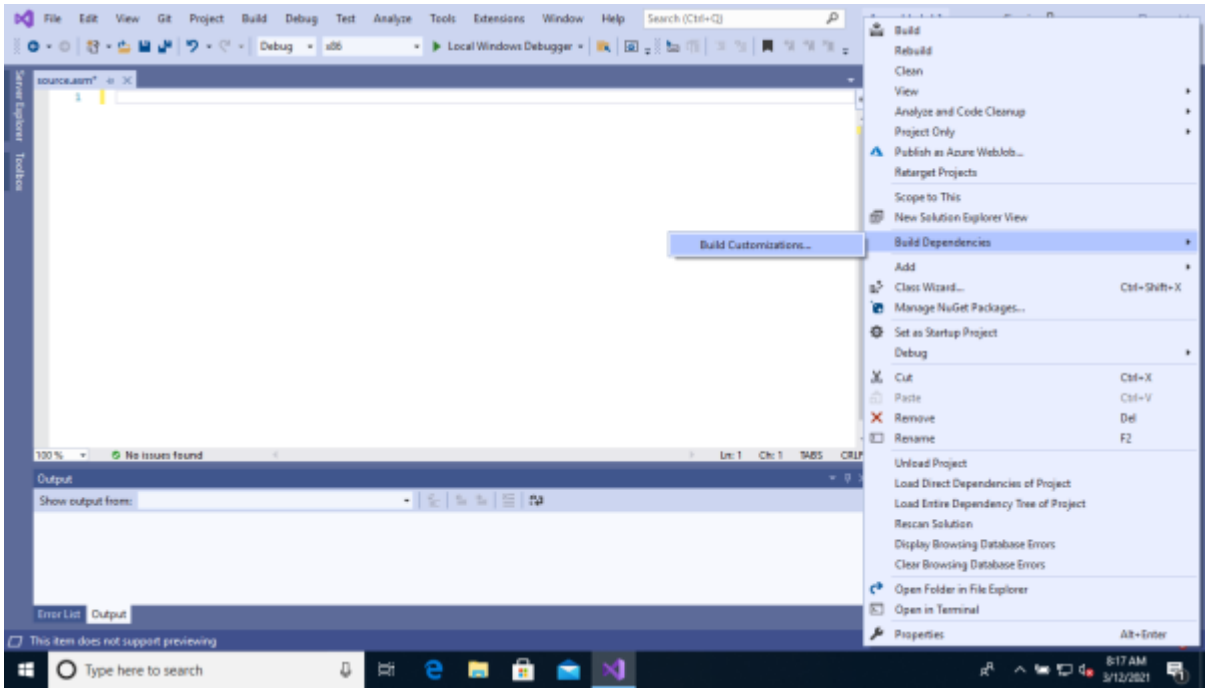


Fig. 7: Wybór menu dostosowanie budowania projektu

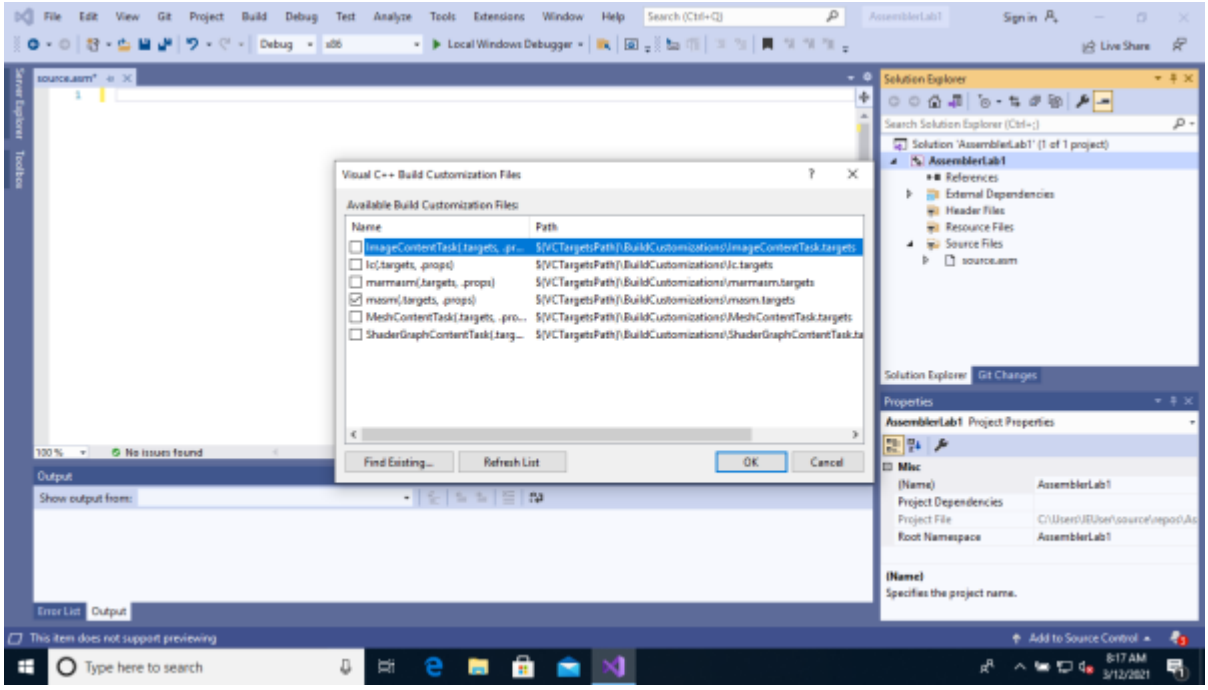


Fig. 8: Wybór kompilatora MASM

Następnie należy zmienić rodzaj budowanej aplikacji z x86 (aplikacja 32-bitowa) na x64 (aplikacja 64-bitowa): menu Build→Configuration Manager rysunek 9.

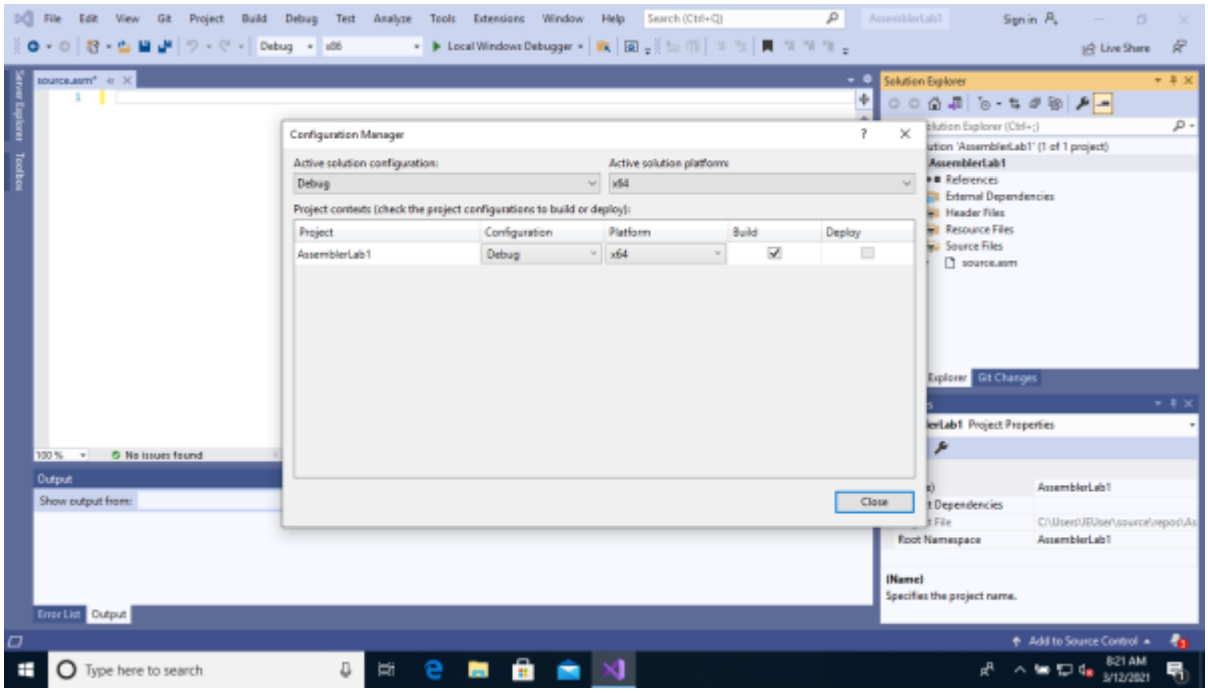


Fig. 9: Konfiguracja środowiska - budowa wersji 64-bitowej

### 1.1.5.3. Utworzenie punktu startowego oraz uzupełnienie bibliotek

W tym kroku poinformujemy linker, jaki punkt startowy ma wpisać do pliku wykonywanego. W językach wysokiego poziomu jest to domyślnie funkcja `main()`. W przypadku assemblera nie ma takiego założenia, a zatem kod źródłowy musi być zgodny z ustawioną tutaj wartością.



Podana do linkera nazwa funkcji punktu wejścia do kodu musi zgadzać się z kodem źródłowym. W przypadku rozbieżności, program będzie działał niepoprawnie, nie uruchomi się lub rzuci wyjątkiem przy starcie!

W podanym przykładzie, nazwa funkcji to `mojaFunkcjaAsm`. Aby dostać się do okna konfiguracyjnego należy w drzewie projektu wskazać projekt (nie solucję!), a następnie prawym przyciskiem myszy kilknąć w projekt, wybrać pozycję `Properties`, lub na skróty [Alt+Enter]. Pojawi się okienko jak na rysunku 10. Następnie w sekcji `Linker` należy wybrać opcję `Advanced` i w polu `Entry Point` wpisać nazwę funkcji.

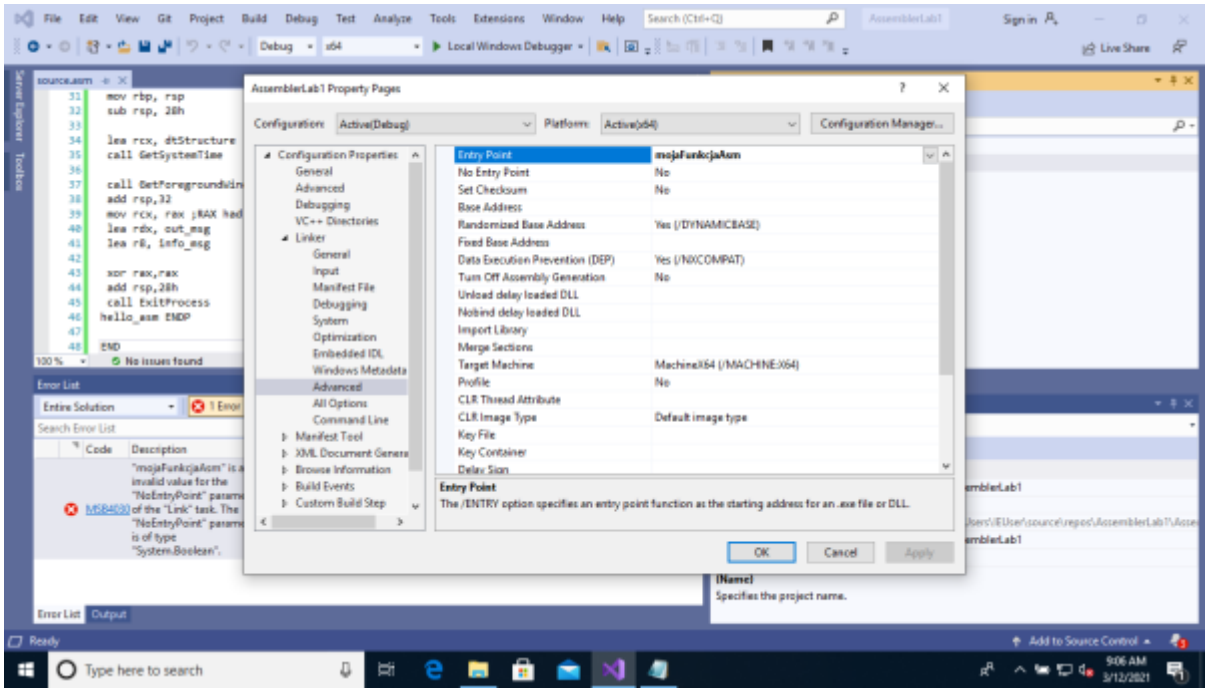


Fig. 10: Konfiguracja punktu wejścia do programu

W tej sekcji należy również dołożyć brakujące biblioteki, z których będziemy korzystać. Uwaga - w zależności od wersji Windows i bieżącego poziomu aktualizacji, ten krok może nie być potrzebny, ale nie zaszkodzi. W szczególności w tych bibliotekach znajdują się finkcje typu `sprintf`.
Uwaga - podane biblioteki należy dołożyć do tych, które standardowo już tam się znajdują. Usunięcie znajdujących się tam bibliotek (m.in. `kernel32.lib`,`user32.lib`) spowoduje brak możliwości użycia standardowych okienek dialogowych oraz interakcji z Windows API. Aby dołożyć biblioteki należy wskazać w menu `Linker` pozyję `Input`, a następnie wybrać `Additional Dependencies` i opcję `<Edit..>` po rozwinięciu listy wyboru. Następnie w górnym polu wpisać nazwy plików bibliotecznych, które mają być dostępne dodatkowo podczas kompilacji (jedna pozycja w linii). Biblioteki do uzupełnienia to:

- `legacy_stdio_definitions.lib`
- `legacy_stdio_wide_specifiers.lib`

Szczegóły na rysunku 11.

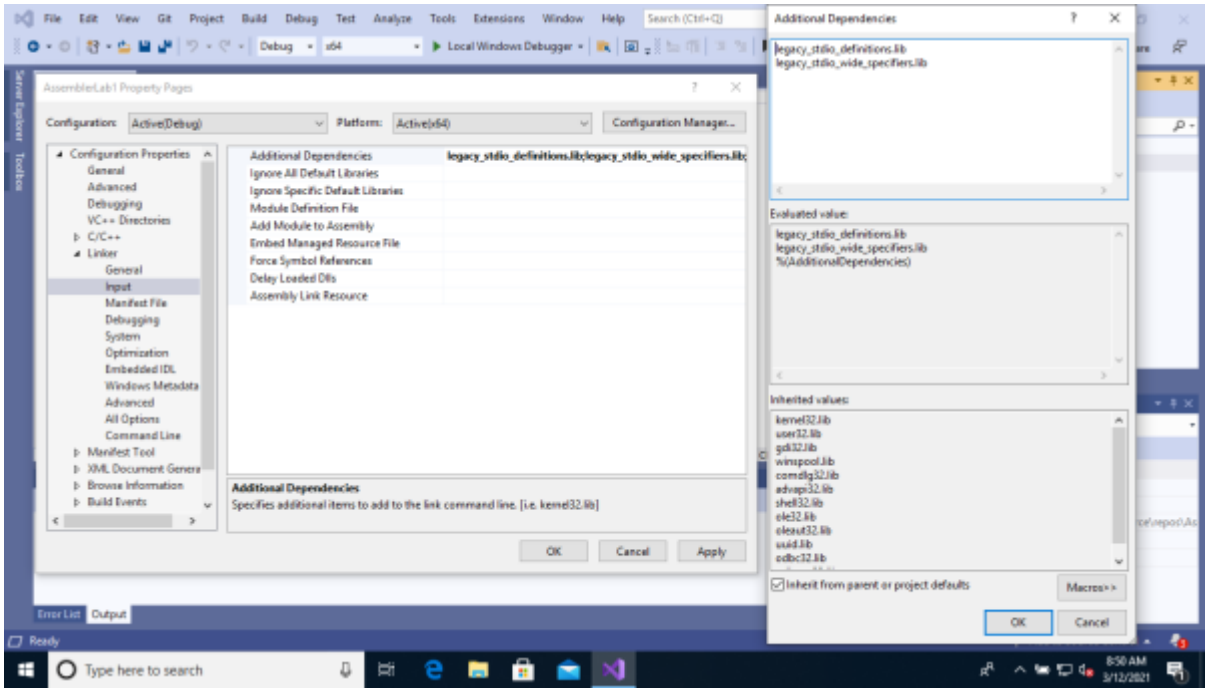


Fig. 11: Konfiguracja bibliotek dodatkowych

### 1.1.5.4. Oznaczenie pliku ASM do kompilacji

W niektórych przypadkach zdarza się, że plik `.asm` nie jest automatycznie rozpoznawany jako plik źródłowy z kodem w assemblerze. W związku z tym należy jawnie poinformować o tym środowiko. W tym celu należy kliknąć nazwę pliku w drzewie projektu prawym przyciskiem i wybrać opcję `Properites`, a następnie w sekcji `General` wskazać `Item Type` i ustawić jego wartośc na `Microsoft Macro Assembler`. Szczegóły na rysunku 12.

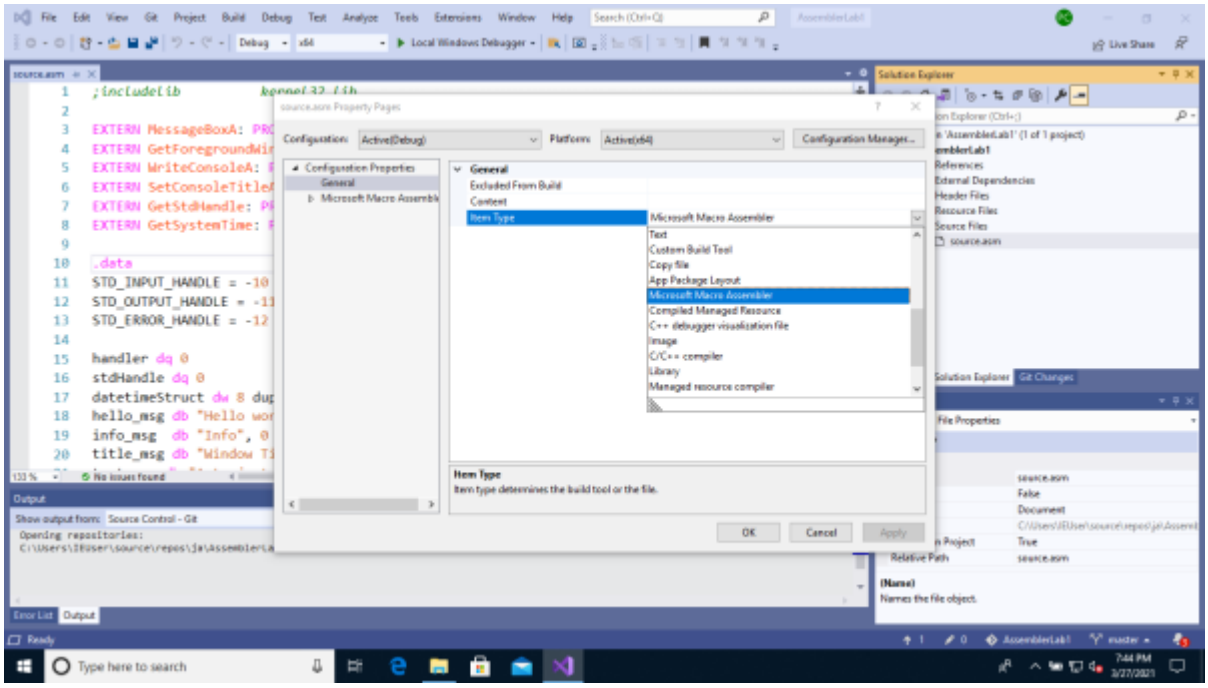


Fig. 12: Oznaczenie pliku ASM jako pliku z kodem źródłowym w assemblerze

### 1.1.5.5. Instalacja dodatku kolorującego kod źródłowy w assemblerze

Na sieci jest dostępne rozszerzenie do Visual Studio o nazwie **AsmDude**, które (m.in.) automatycznie

koloruje kod źródłowy w assemblerze, pomaga w pisaniu kodu (podaje nazwy etykiet, zmiennych, itp) oraz podczas debbugowania wyświetla okienka z wartościami rejestrów. Plugin ten może zostać zainstalowany poprzez managera rozszerzeń Visual Studio[7].

## 1.1.6. Praktyka: Zadanie do wykonania

Zaimplementuj odczyt daty (użyj funkcji `GetSystemTime`(sysinfoapi.h)[8] odczytującej strukturę `_SYSTEMTIME`(minwinbase.h)[9]. Struktura składa się z 8 wartości szesnastobitowych.



Uwaga - nie deklaruj struktury, jako takiej, tylko dla uproszczenia zadeklaruj docelową, statyczną tablicę 8 wartości typu Word (16bit) w segmencie danych.

Skonwertuj rok, miesiąc i dzień do postaci tekstowej ASCII i wyświetl na ekranie za pomocą:

- w linii poleceń (cmd) za pomocą funkcji `WriteConsoleA`;
- w okienku dialogowym, za pomocą funkcji `MessageBoxA`;



Zwróć uwagę na położenie poszczególnych słów w strukturze i odpowiadające im wartości.

Możesz dodatkowo wyświetlić bieżącą godzinę, minuty i sekundy.

### 1.1.6.1. Fragmenty kodu do wykorzystania

Pamiętaj o deklaracji segmentu kodu i danych za pomocą operatorów `.data` i `.code`. Na końcu kodu obowiązkowo umieść klauzulę `END`.

#### 1.1.6.1.1. Segment danych

Każdy łańcuch tekstowy musi kończyć się znacznikem o binarnej wartości 0. Nie używamy polskich znaków, ponieważ korzystamy z zestawu znaków ASCII (8-bit), a nie Unicode.
Przykładowy segment danych, który był używany w prezentacji wygląda tak:

```
...

.data
STD_INPUT_HANDLE = -10
STD_OUTPUT_HANDLE = -11
STD_ERROR_HANDLE = -12

handler dq 0
stdHandle dq 0
datetimeStruct dw 16 dup (0)
hello_msg db "Hello world", 0
```

```
info_msg  db "Info", 0
title_msg db "Window Title JA",0
text_msg db "A to jest bardzo dlugi tekst do wyswietlenia w konsoli, ale bez
polskich znakow!",13,10
dummy dq 0

...
```

Aby móc skorzystać z zewnętrznych funkcji Windows API, trzeba poinformować assembler, jak one się nazywają, aby linker mógł potem dopasować nazwy z Twojego kodu do tych, które znajdują się w bibliotece.
W tym celu trzeba użyć dyrektywy EXTERN oraz PROC:

```
...

includelib          kernel32.lib

EXTERN MessageBoxA: PROC
EXTERN GetForegroundWindow: PROC
EXTERN WriteConsoleA: PROC
EXTERN SetConsoleTitleA: PROC
EXTERN GetStdHandle: PROC
EXTERN GetSystemTime: PROC

...
```

Zwyczajowo umieszczamy również informację o bibliotece.

#### 1.1.6.1.2. Segment kodu

Po uruchomieniu programu musimy zachować wartości pewnych rejestrów, w celu ich późniejszego przywrócenia. Na sam koniec należy również wywołać funkcję kończącą program i zwracającą tzw kod powrotu do systemu operacyjnego.

Minimalny zestaw instrukcji wygląda następująco:

```
...
.code
hello_world_asm PROC
push rbp ; save frame pointer
mov rbp, rsp ; fix stack pointer
sub rsp, 8 * (4 + 2)

.... ; tutaj kod Twojego programu


mov rsp, rbp
pop rbp
ret
```

```
hello_world_asm ENDP
END
```



Uwaga, w powyższym przykładzie, główną procedurą jest `hello_world_asm` i to ona została wskazana jako [Entry Point] (punkt wejścia) do programu, w konfiguracji opisanej powyżej. Dostosuj do Twojego nazewnictwa.

Jak widać, całość sprowadza się do zarezerwowania i odłożenia na stosie przestrzeni na tzw. shadow registry, czyli na zachowanie wartości rejestrów i przywrócenie ich po zakończeniu wykonywania programu.

W programie wykorzystujemy interakcję z elementami graficznymi takimi jak okienko główne (Foreground Window) oraz strumień danych (STD_OUT). Podczas wywoływania funkcji, należy jednoznacznie wskazać, względem którego obiektu graficznego chcemy wykonać interakcję lub stworzyć obiekt potomny (np. MessageBox).
W tym celu musimy pobrać dwa uchwyty (handlery):

- uchwyt do bieżącego głównego okienka aplikacji na poziomie systemu operacyjnego (tzw. Foreground Window)
- uchwyt do strumienia danych typu terminalowego, do którego można wpisywać informacje.



Uwaga! Aby wyświetlić coś na konsoli, należy zawsze nacisnąć Enter, nawet ... wirtualnie. A zatem dowolny ciąg znaków przekierowany na STD_OUT pojawi się na konsoli o ile ... na końcu znajdzie się specjalny kod oznaczający znak ENTER. Są to dwie wartości numeryczne, w kolejności 13 i 10. Stąd właśnie w przykładzie w segmencie danych, po treści znajduje się `,13,10`. Jeśli tego nie zrobisz, konieczne będzie przejście do okienka i ręczne naciśnięcie ENTER w celu wyświetlenia przekierowanego tam ciągu znaków z Twojego programu. Inaczej okienko pozostanie puste i tekst nie pojawi się!

W celu odczytania uchwytów przydatne będą dwie funkcje z Windows API, które zwrócą nam wskaźniki (uchwyty) do odpowiednich obiektów. Wartości te należy zachować i można je wykorzystywać wielokrotnie.

Odczyt uchwytu do okienka głównego, ten uchwyt należy zastosować przy wywoływaniu okna dialogowego typu Message Box (i każdego innego):

```
...
call GetForegroundWindow
mov handler, rax
...
```

W powyższym przykładzie zwracana przez funkcję wartość (przez rejestr RAX) zapisywana jest w zmiennej `handler`, typu DQ (64 bity, czyli po prostu adres w pamięci).

Aby odczytać uchwyt do wnętrza terminala, należy skorzystać z innej funkcji:

```
...
```

```
mov rcx, STD_OUTPUT_HANDLE
call GetStdHandle
mov stdHandle, rax
...
```

Przy czym ta funkcja wymaga jednego argumentu, o który strumień danych nam chodzi. Są trzy możliwości:

- STD_INPUT_HANDLE = -10 - standardowe wejście - można go użyć do odczytu danych z konsoli
- STD_OUTPUT_HANDLE = -11 - standardowe wyjście - jego używamy aby coś napisać w terminalu; pamiętaj o kończącym ENTER, czyli `13,10`!;
- STD_ERROR_HANDLE = -12 - strumień dla błędów;

Przykładowy kod wyswietlający na konsoli tekst znajdujący się w zmiennej (pod adresem) `text_msg` znajduje się poniżej. Pierwszy parametr to odczytany wyżej uchwyt do konsoli. W trzecim parametrze należy podać, ile znaków ma zostać wyświetlonych. Czwarty parametr to wskaźnik do miejsca, gdzie funkcja zwróci informacje, ile tak na prawdę wyświetliła znaków (nie do końca wiadomo, czemu to służy).

```
...
; WINAPI WriteConsoleA(
;  RCX => _In_ HANDLE  hConsoleOutput,
;  RDX => _In_ const VOID    *lpBuffer,
;  R8 => _In_ DWORD   nNumberOfCharsToWrite,
;  R9 => _Out_opt_ LPDWORD lpNumberOfCharsWritten,
mov rcx, stdHandle
lea rdx, text_msg
mov r8, sizeof text_msg
mov r9, dummy
call WriteConsoleA
...
```

W celu wyświetlenia okienka dialogowego, należy użyć wskaźnika do okna głównego aplikacji. Jako parametry przekazywane są w kolejności: uchwyt, dwa wskaźniki, do tekstów (muszą kończyć się zerem, RDX to wskaźnik do treści, którą wyświetli dialog, a R8 wskaźnik do nagłówka okienka, który zotanie nadany okienku), które mają pojawić się w nagłówku oraz treści dialogu oraz jako ostatni, maska binarna określająca, jakie mają być zastosowane przyciski i ikony w okienku dialogowym[10)].

```
...
; WINUSERAPI int WINAPI MessageBoxA(
;  RCX =>  _In_opt_ HWND hWnd,
;  RDX =>  _In_opt_ LPCSTR lpText,
;  R8  =>  _In_opt_ LPCSTR lpCaption,
;  R9  =>  _In_ UINT uType);
mov rcx, handler
mov rdx, offset hello_msg
mov r8, offset info_msg
mov r9, 0 ; MB_OK
and rsp, not 8
call MessageBoxA
...
```



Uwaga - w powyższym przypadku wykonano jawnie wyrównanie stosu do wartości podzielnej przez 16 (`and rsp, not 8`). W większości przypadków nie jest to konieczne, ponieważ organizacja danych najczęściej jest 64-bitowa. Gdyby jednak okazało się, że gdzieś został odłożony na stosie pojedynczy bajt lub słowo, ulegnie to zmianie i program może rzucić wyjątkiem. W związku z tym w takim przypadku należy zadbać o wyrównanie wskaźnika stosu tak, aby był podzielny przez 16.

### 1.1.7. Debuggowanie i uruchamianie

Uwagi praktycznie odnośnie debbugowania i podglądania pamięci:

- Pamiętaj, aby na poziomie wywołań funkcji Windows API nigdy nie wykonywać operacji wejścia do wewnątrz ich definicji (Step Into, F11), o ile wcześniej nie zainstalowałeś Windows SDK wraz z definicjami i kodem źródłowym. Zamiast tego należy używać przejścieа do następnej instrukcji (Step Over, F10)
- Okno rejestrów domyślnie pokazuje tylko podstawowy zestaw: więcej po kliknięciu prawym przyciskiem myszy i włączeniu innych zestawów rejestrów
- Okno podglądu pamięci pozwala na zmianę formatowania danych z domyślnego, bajtowego, na wiele innych. Np w kontekście podglądu ww. struktury składającej się z danych 16-bitowych zdecydowanie warto zmienić podgląd na 2-bajtowy. Natomiast do podglądania ciągów znaków ASCII należy wrócić do jednobajtowego.



Aby podejrzeć zawartość pamięci, najprościej jest odnaleźć adres zmiennej w rejestrze, po wykonaniu instrukcji LEA(Load Effective Address). Wartość z okienka podglądu stanu rejestrów należy przekopiować do okienka podglądu stanu pamięci.



Uwaga - powyższą operację należy wykonać pod debuggerem zaraz po wykonaniu instrukcji `LEA` ponieważ następujący po niej `CALL` powoduje zmiany zawartości rejestrów i adres zostanie zamazany.

Na rysunku rysunek 13 przedstawiono kolejność działań w celu odczytania (odnalezienia) miejsca w pamięci, gdzie znajduje się zmienna. Breakpoint (pułapkę) należy ustawić bezpośrednio na następnej instrukcji znajdującej sie po LEA, chyba że jest gwarancja, iż instrukcje pomiędzy LEA i breakpointem nie modyfikują danego rejestru. Odczytaną wartość należy przenieść z okienka rejestrów do okienka pamięci. Po wywołaniu funkcji, zawartość rejestru RCX zmienia się, natomiast dane zwracane są do pamięci, jak na rysunku rysunek 14. Zadanie dodatkowe: zamień widoczną na rysunku zawartość pamięci, a dowiesz się, kiedy powstał screenshot 😉.

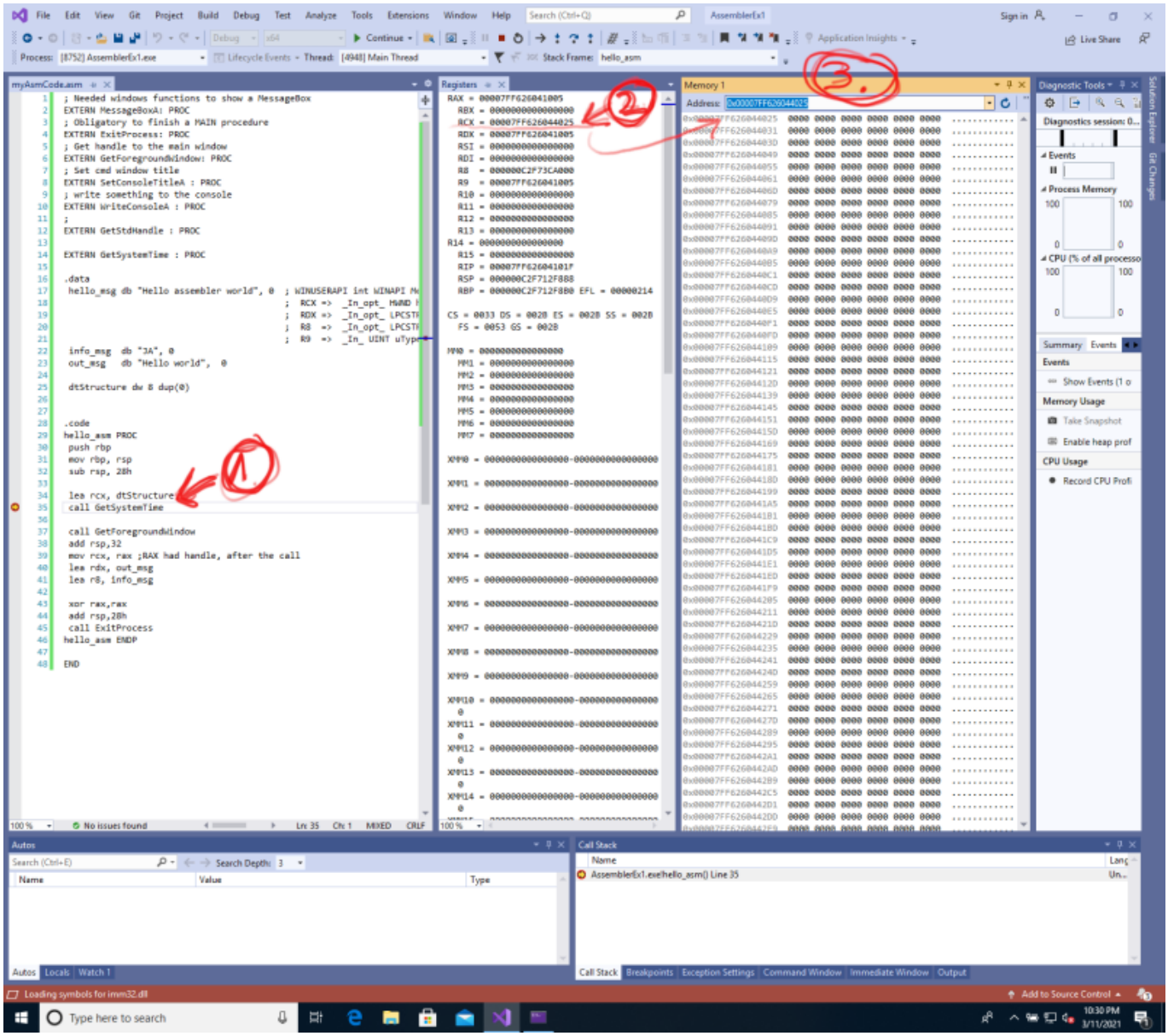


Fig. 13: Wyciąganie adresu dla zmiennej (stan bezpośrednio po wykonaniu LEA)

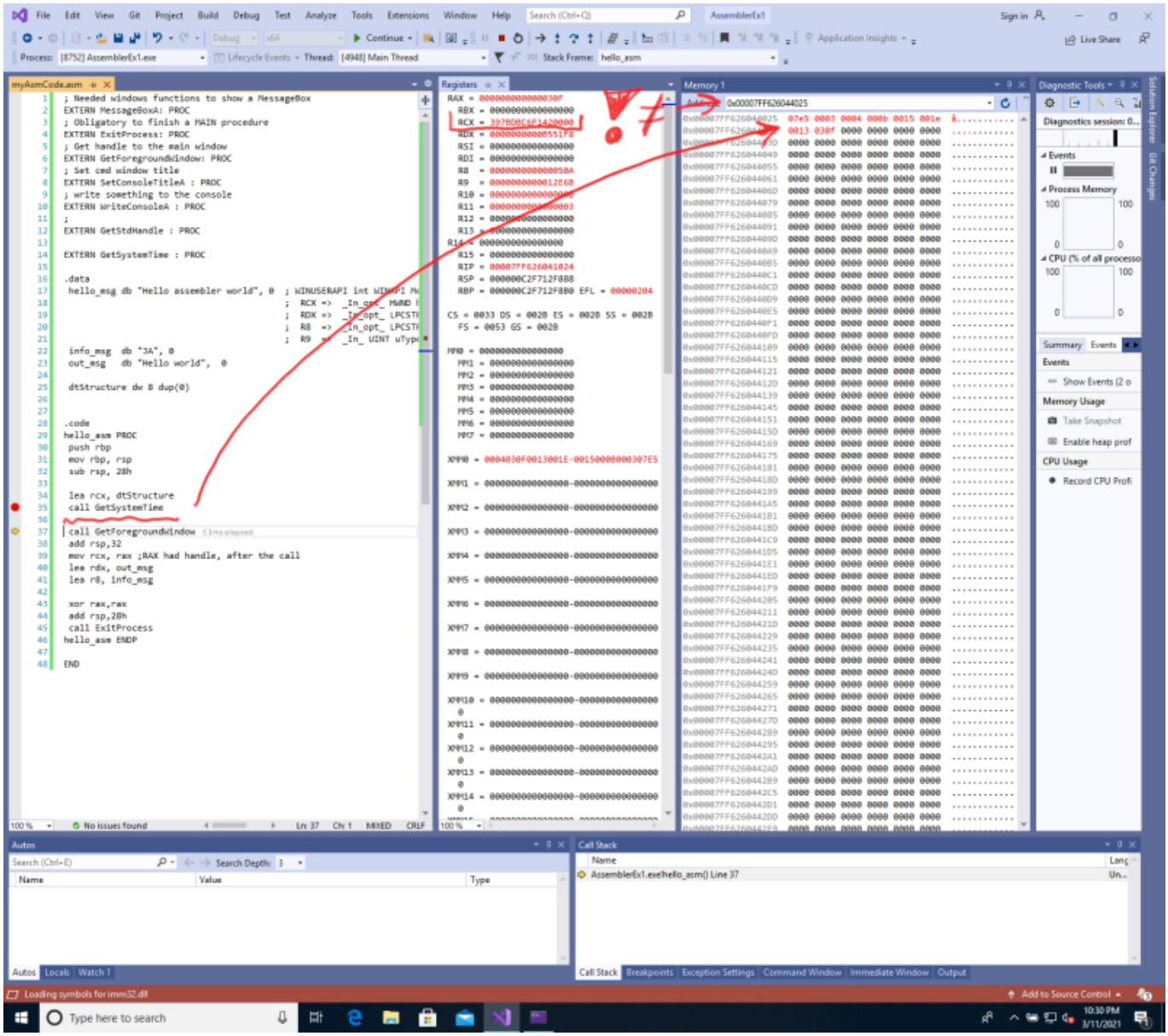


Fig. 14: Obserwacja wyniku w pamięc, po wywołaniu funkcji

## 1.1.8. Metoda weryfikacji wyników

Teksty oraz okienka pojawiające się na ekranie zgadzają się z algorytmem. Zwróć szczególną uwagę na to, czy nie są generowane losowe ciągi znaków, co może sugerować, że zapomniałeś o kończącej łańcuch wartości 0.

## 1.1.9. Pomocne zasoby

- Visual Studio 2019: https://visualstudio.microsoft.com/vs/
- Przydatne funkcje Windows API: https://docs.microsoft.com/en-us/windows/win32/api/_winmsg/

1)

https://visualstudio.microsoft.com/vs/

2)

https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/

3)

https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/

4)
https://www.virtualbox.org
5)
https://www.linux-kvm.org/page/Main_Page
6)
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160
7)
https://marketplace.visualstudio.com/items?itemName=Henk-JanLebbink.AsmDude
8)
https://docs.microsoft.com/en-us/windows/win32/api/sysinfoapi/nf-sysinfoapi-getsystemtime
9)
https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime
10)
https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxa

From:
https://dokuwiki.zmitac.aei.polsl.pl/ - **RAU6 Dokuwiki**

Permanent link:
**https://dokuwiki.zmitac.aei.polsl.pl/pl:edu:zmitac:n1z:ja:lab1?rev=1616871215**

Last update: **2021/03/27 19:53**