Table of Contents

1. Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z aplikacją typu WPF w .NET C#, instrukcje SIMD 2
1.1. Środowisko sprzętowe i programistyczne 2
1.2. Wymagania 2
1.3. Wstęp 2
1.4. Konfiguracja projektu 3
1.5. WPF i assembler 3
1.5.1. Tworzenie projektu WPF 3
1.5.2. Dozostałe ustawienia solucji 7
1.5.3. Assembler 7
1.6. Zadanie do wykonania 7
1.7. Metoda weryfikacji wyników 8
1.8. Pomocne zasoby 9

1. Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z aplikacją typu WPF w .NET C#, instrukcje SIMD

1.1. 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 Community Edition[1)].



Proszę nie pomylić z Visual Studio Code, to zupełnie inny produkt, który nie nadaje się wprost do wykorzystania w naszym laboratorium do realizacji zadań.



Szczegóły instalacji i konfiguracji środowiska przedstawiono w ramach Laboratorium 1 i 2.

1.2. Wymagania

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

1.3. Wstęp

W czasie zajęć laboratoryjnych skorzystamy z modelu WPF, przy czym będziemy wykorzystywać bezpośrednie akcje (Eventy), a nie bindowanie zmiennych (Bindings) w celu ułatwienia analizy kodu. GUI będzie prostym oknem dialogowym, które będzie zawierało zestaw kontrolek służących do pobierania i wyświetlania wartości (pola typu `TextBox`) oraz przycisków wywołujących akcje (`Button`).



W ramach zajęć nie będziemy tworzyli żadnych elementów walidujących wprowadzane przez użytkownika dane, w związku z tym w przypadku wpisania łańcucha znaków, którego np. nie można poprawnie skonwertować do wartości numerycznej, program po prostu zgłosi wyjątek. Należy o tym pamiętać podczas testowania, w szczególności istotnym elementem jest znak reprezentujący separator wartości dziesiętnej dla liczb zmiennoprzecinkowych, który jest zależny od języka interfejsu użytkownika w systemie Windows.

1.4. Konfiguracja projektu

W ramach niniejszego laboratorium użyjemy projektu analogicznego do projektu w laboratorium nr 3, przy czym zamiast aplikacji konsolowej w języku C# zostanie wykorzystany frontend w modelu WPF (aplikacja okienkowa).

1.5. WPF i assembler

Windows Presentation Fundation (WPF) to w skrócie sposób tworzenia interfejsów użytkownika dla systemu Windows z wykorzystaniem środowiska .NET (C#, F#, VB.NET). WPF zastępuje poprzednie Windows Forms, związane z kodem niezarządzalnym i językiem C++. WPF cechuje się tym, że widok (layout, okienka i układ kontrolek) jest odseparowany od kodu zarządzającego zawartością i interakcją. Wygląd jest zapisany w postaci osobnych plików z rozszerzeniem `.xaml`, a kod reprezentujący okienko jest w dużej mierze generowany “w locie” podczas edycji wyglądu w edytorze Visual Studio (tzw “Code Behind”). Odseparowanie wyglądu od kodu pozwala na tworzenie modyfikacji wyglądu przez grafików, bez udziału programistów (tworzenie tzw. skórek). Dla każdego widoku (okienka) są zatem tworzone co najmniej dwa pliki: jeden z kodem, drugi z plikiem XML (`.xaml`) reprezentującym położenie i cechy graficzne. WPF jest skalowalny i posiada możliwość tworzenia inerfejsów użytkownika, które automatycznie dostosowują się do proporcji i rozmiarów ekranu oraz ich rozdzielczości i wymiarów (DPI). Więcej informacji można znaleźć w oficjalnym portalu Microsoft[2)] lub na wielu innych stronach. WPF na dzień dzisiejszy jest wiodącym środowiskiem tworzenia aplikacji okienkowych dla Windows w środowisku .NET.

1.5.1. Tworzenie projektu WPF

W Visual Studio należy utworzyć projekt typu WPF Desktop w języku C# (rysnuek 1):

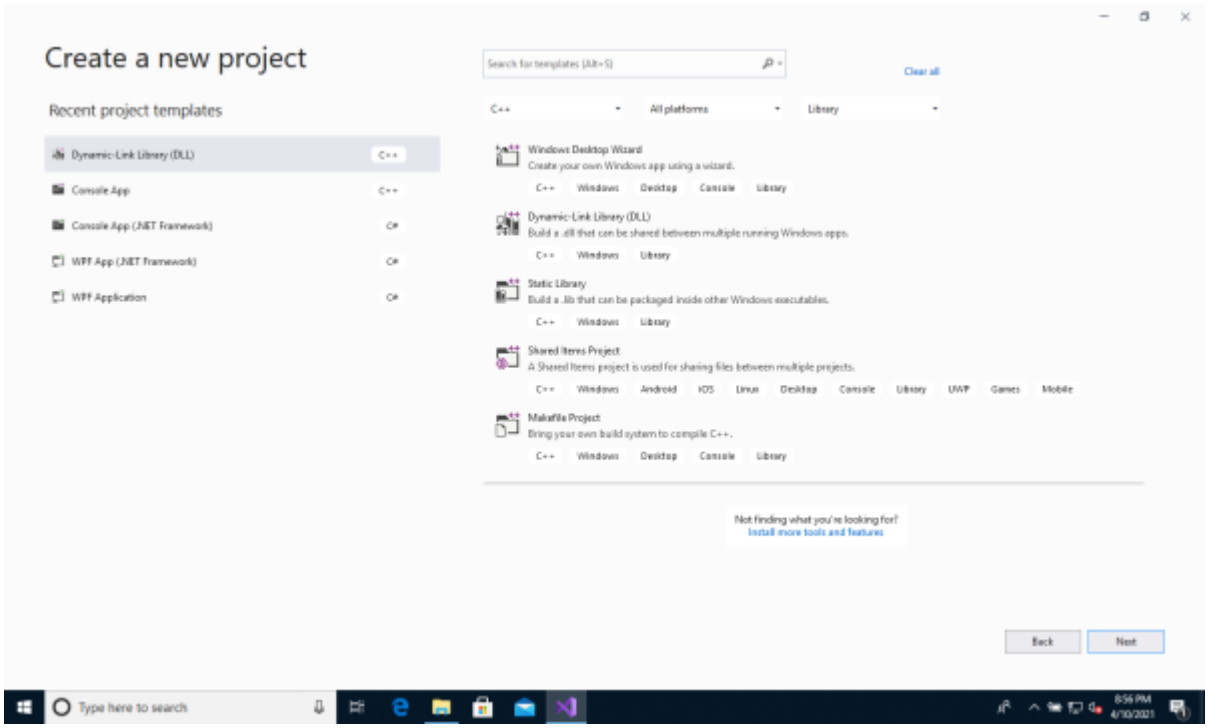


Fig. 1: Tworzenie projektu WPF

Na etapie pytania o wersję frameworku można wybrać dowolny powyżej 3.0, może to być np. 5.0 lub 4.x (o ile został zainstalowany).
Po stworzeniu projektu pojawi się okienko graficznego projektowania interfejsu użytkownika wraz ze skojarzonym plikiem `.xaml`. Dane w pliku opisują wygląd okienka, a zatem interakcja po stronie okienka (np. dołożenie kontrolek) powoduje automatyczną modyfikację treści pliku `.xaml`. Zależność ta jest prawdziwa również w drugim kierunku, czyli ręczne wpisanie tekstu do pliku `.xmal` spowoduje zmianę wyglądu interfejsu użytkownika. \\Główne okienko aplikacji automatycznie otrzymuje nazwę `MainWindow`, a zatem związany z nim plik opisujący będzie miał oznaczenie `MainWindow.xaml`, a kod w języku C# (tzw. Code Behind) odpowiednio `MainWindow.xaml.cs`.

Podczas projektowania, nowe elementy najłatwiej dodać, przeciągając je z menu [Toolbox] (na rysunku 2 po lewej stronie).
Na potrzeby naszej aplikacji wystarczą typy `TextBox` (nie pomylić z `TextBlock`!) oraz `Button`.

Przykładowy interface użytkownika, który będzie odpowiedni na potrzeby niniejszego projektu zaprezentowano na rysunku 2. Każdej kontrolce, do której będzie bezpośrednio odwoływać się kod (tutaj pola typu `TextBox`) należy nadać unikalną nazwę. Nadawanie nazw odbywa się w okienku właściwości ([Properties]) obiektu, które na rysunku 2 znajduje się w prawym dolnym rogu interfejsu użytkownika Visual Studio. Wprowadzenie nazwy ma automatycznie swoje odzwierciedlenie w pliku opisowym `.xaml`.

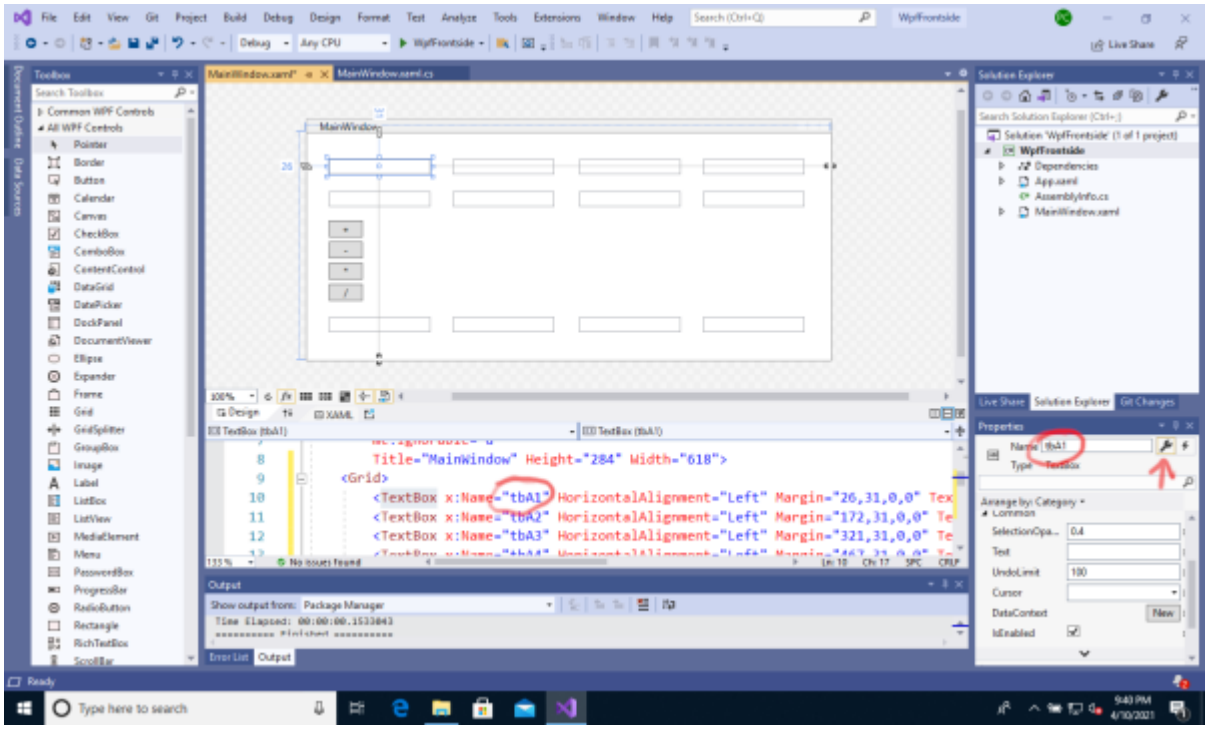


Fig. 2: Przykładowy interface użytkownika



Uwaga! Okienko właściwości obiektu graficznego działa w dwóch trybach, przełączanych za pomocą przycisków znajdujących się na górze, po prawej stronie okienka: klucz oznacza tryb zmiany właściwości (Properties), a błyskawica tryb zmiany akcji (Events). Na rysunku 2 miejsce to zaznaczono czerwoną strzałką.

W przypadku przycisków, nie jest konieczne nadawanie im nazwy, natomiast konieczne jest utworzenie akcji związanej z kliknięciem przycisku. Akcja ta (Event) zostanie wywołana po kliknięciu danego przycisku. Funkcje przypisane do kolejnych przycisków muszą mieć unikalne nazwy, ponieważ

wszystkie są umieszczone w klasie widoku okienka głównego.
Akcję można stworzyć na kilka sposobów:

- Jednym z nich jest dwukrotne kliknięcie przycisku w edytorze graficznym. Metoda ta nie pozwala jednak na wybranie nazwy funkcji.
- Drugim jest przełączenie okienka dialogowego właściwości przycisku w tryb edycji akcji (znaczek błyskawicy, jak na rysunku 3) i wpisanie nazwy funkcji.
- Jest też metoda ręcznej edycja pliku `.xaml` oraz `.xaml.cs` - metoda ta nie jest polecana bo łatwo tutaj popełnić błąd.

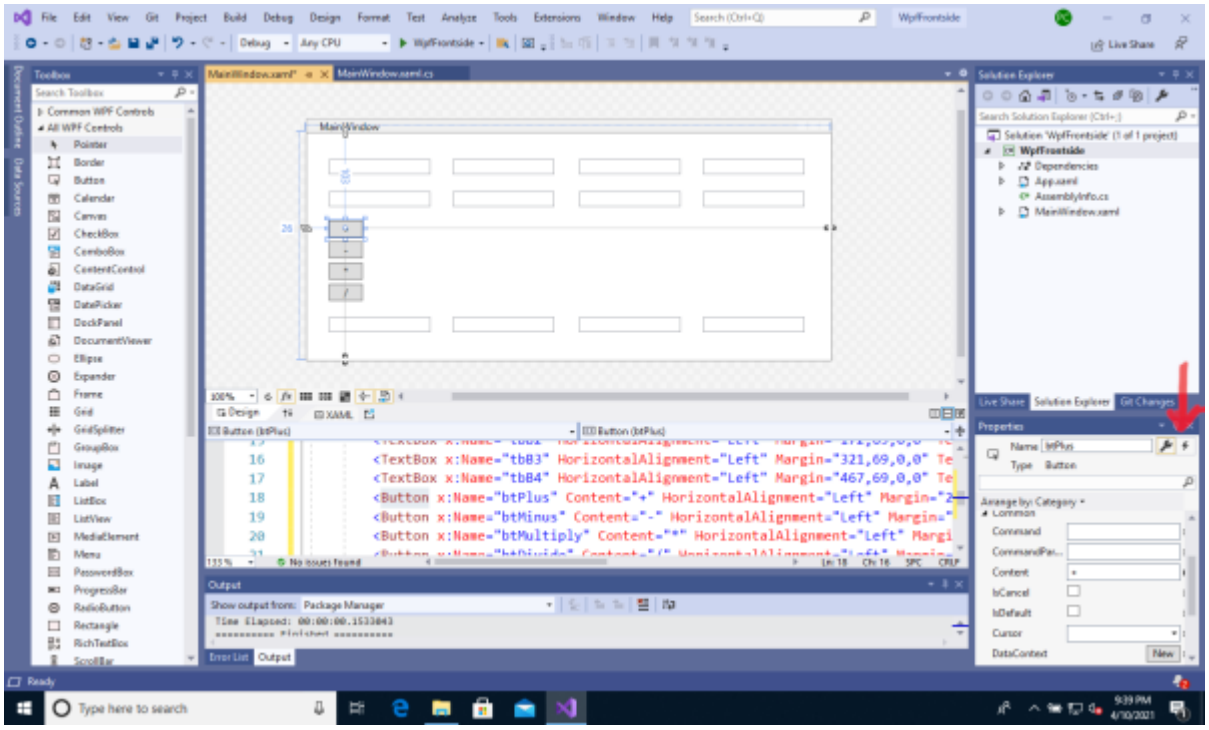


Fig. 3: Przełączenie okienka właściwości w tryb edycji akcji

Dla akcji wykonywanej po kliknięciu przycisku, należy wypełnić pole “Click” (rysunek 4):

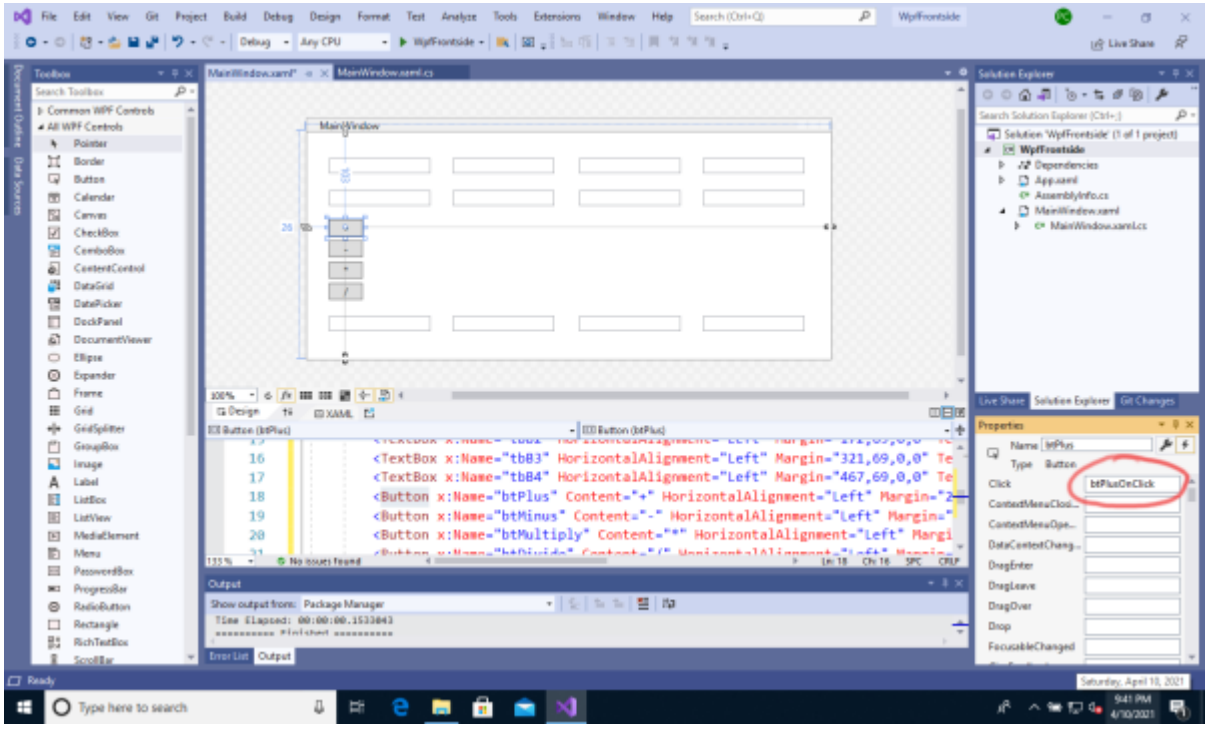


Fig. 4: Akcja `btPlusOnClick`



Wskazane jest, aby nazwa akcji (nazwa funkcji wywoływanej po pojawieniu się interakcji) odzwierciedlała naturę tej akcji oraz odnosiła się do obiektu, którego dotyczy, bowiem w kodzie zarządzającym okienkiem (Code Behind) nie ma bezpośredniego kontekstu i odwołania do obiektu graficznego. używanie nazw typu `funkcja1` nie niesie ze sobą informacji ani w jakich okolicznościach ta funkcja jest wywoływana, ani dla jakiego obiektu.



Przełączenie z widoku projektowania interfejsu do kodu C# (Code Behind) najprościej wykonać używając klawisza [F7]. Ewentualnie mozna go wybrać z drzewa projektu (rysunek 5).

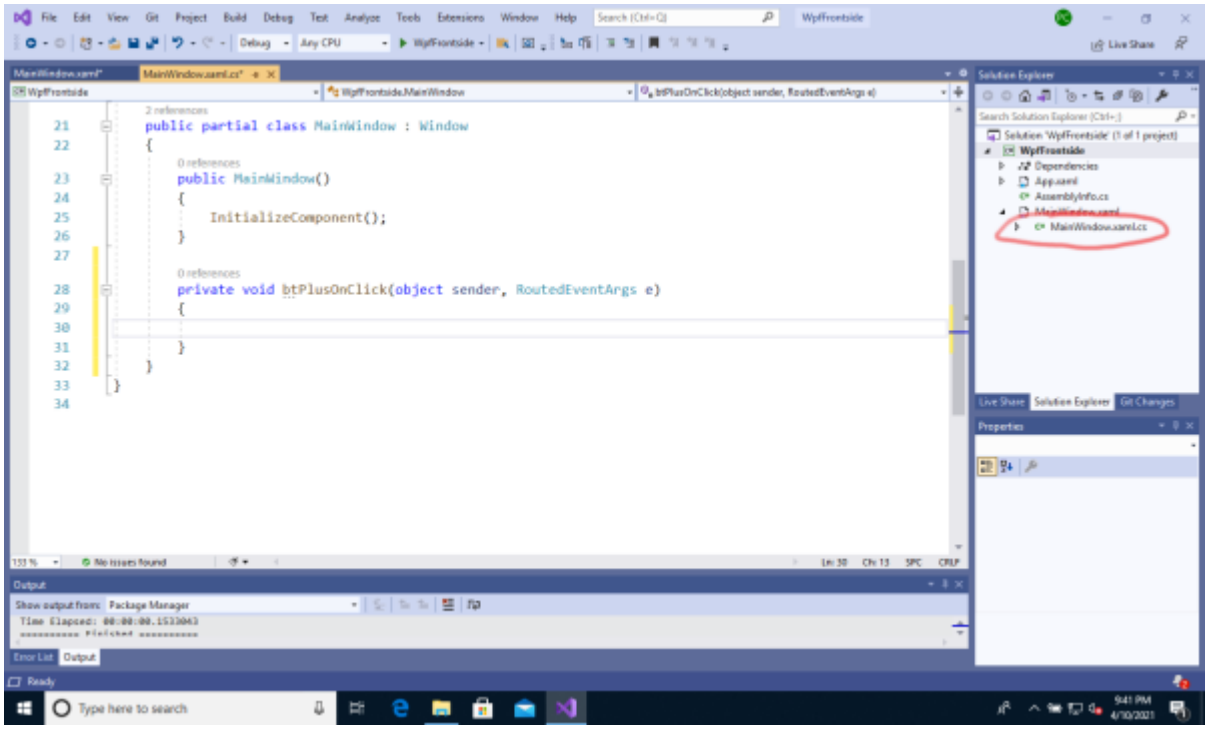


Fig. 5: Code Behind - kod zarządzający akcjami okienka

Odwołanie z poziomu kodu do obiektów graficznych jest realizowane poprzez nazwę instancji obiektu graficznego. Wszystkie kontrolki typu `TextBox` dostarczają zawartość wpisaną w kontrolkę w postaci tekstu (`String`), w polu `Text`. W naszym przykładie interesują nas jednak dane numeryczne. W związku z tym istnieje konieczność konwersji typów z typu `String` to typu numerycznego (tutaj `Float`, analogicznie dla innych typów). Służy do tego funkcja `<typ>.TryParse(…)`. Nietypowo, funkcja ta nie zwraca wartości bezpośrednio, lecz poprzez parametr typu `out` co jest wymagane jawnie podczas wywołania:

```
   ...
   float.TryParse(tbA1.Text, out aArg1[0]); //tbA1 - pole tekstowe, aArg1 -
tablica typu float
   ...
```

Jeśli konwersja nie może zostać dokonana, ponieważ nie da się “przetłumaczyć” łańcucha na zmienną

określowego typu, zostanie zgłoszony wyjątek i wykonywanie programu zostanie przerwane. Oczywiście istnieją mechanizmy walidacji, które pozwalają na wykrycie tego typu sytuacji oraz ich obsługę, niemniej jednak wykraczają one poza ramy niniejszej instrukcji.

1.5.2. Dozostałe ustawienia solucji

W projekcie WPF należy włączyć możliwość kompilacji kodu typu `unsafe` oraz debbugowanie kodu natywnego, analogicznie jak to opisano w instrukcji laboratoryjnej Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z kodem zarządzalnym .NET C#.

Należy też pamiętać o wybraniu wspólnego folderu do generowania wynikowego kodu z projektu w assemblerze i projektu w języku C#, analogicznie jak to opisano w instrukcji laboratoryjnej Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z kodem zarządzalnym .NET C#.



Uwaga! W przypadku projektu WPF w folderze wynikowym może pojawić się podfolder z wersją frameworku .NET, co w praktyce spowoduje, że plik wykonywalny z frontside (GUI) zostanie uruchomiony w innym folderze (w podfolderze) w stosunku do wybranego w ustawieniach, skutkiem czego biblioteka DLL nie zostanie załadowana i zostanie zgłoszony wyjątek. Podfoldery te są dodawane automatycznie. Aby usunąć tę niedogodność, w pliku projektu WPF należy mieścić dodatkową sekcję[3]:

```
<PropertyGroup>
<AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>
<AppendRuntimeIdentifierToOutputPath>false</AppendRuntimeIdentifierToOutputPath>
</PropertyGroup>
```

1.5.3. Assembler

Do solucji należy dodać projekt w assemblerze. Kod w assemblerze należy skompilować do biblioteki DLL, a następnie dołączyć ją dynamicznie do rozwiązania w C#. Jest to dokładnie ten sam proces, który został opisany w Ćwiczenie 3: Integracja kodu niezarządzalnego, tworzonego w języku Assembler wraz z kodem zarządzalnym .NET C# w zakresie tworzenia i konfiguracji części assemblerowej oraz jej udostępniania po stronie C#, natomiast zamiast aplikacji konsolowej występuje aplikacja WPF.

1.6. Zadanie do wykonania

Należy utworzyć prosty kalkulator, który przeliczy wektorowo dla 4 zmiennych wyniki dla operacji dodawania, odejmowania, mnożenia i dzielenia (akcje na osobnych przyciskach). Wymagane jest przeliczenie więcej niż jednego zestawu danych w ramach jednej instrukcji dodawania (trzeba skorzystać z SIMD). Zadanie należy wykonać na typach zmiennoprzecinkowych, pojedynczej precyzji

(32 bity, po stronie C# typ `float`).

Należy zadeklarować po jednej funkcji assemblerowej dla każdej operacji.

Wartości najwygodniej będzie przekazać do funkcji assemblerowych wpisując je po stronie kodu w C# do dwóch tablic źródłowych, a na wyniki zaalokować tablicę trzecią. Liczba zmiennych jest znana (4) i stała, zatem długości tablic nie trzeba przekazywać do funkcji assemblerowej. Wskaźniki na tablice można przekazać poprzez rejestry lub poprzez wskaźnik na strkturę zawierającą wskaźniki do ww. tablic (do wyboru).

Tablice po stronie C# można zadeklarować np następująco:

```
...
public partial class MainWindow : Window
    {
        public float[] aArg1 = { 0, 0, 0, 0 }; //argument 1
        public float[] aArg2 = { 0, 0, 0, 0 }; //argument 2
        public float[] aResults = { 0, 0, 0, 0 }; //wynik
...
```



Powyżej dołączono kontekst: zmienne zostały umieszczone w klasie okienka. Słowo kluczowe `partial` w deklaracji typu klasy `MainWindow` oznacza, że w pliku zaimplementowana jest jedynie część klasy. Pozostała część kodu znajduje się w pliku zarządzanym automatycznie przez edytor GUI okienka. Dla dewelopera w praktyce ten mechanizm jest przezroczysty, a cały kod należy umieszczać w tym pliku (`.xaml.cs`).



Część instrukcji SIMD (SSE, AVX) wymaga specyficznego ułożenia danych w pamięci, w szczególności wyrównania (umieszczenia zmiennej pod adresem podzielnym przez 16). C# nie daje gwarancji umieszczenia początkowego elementu w pamięci pod adresem zgodnym z wymaganiami kodu niezarządzalnego (pomimo wyrównywania zmiennych do 64 bitów). W związku z tym, przy dostępie do pamięci, nie należy stosować instrukcji assemblerowych, wymagających wyrównania, w szczególności np. nie należy stosować `MOVAPS`, tylko jej wolniejszą wersję, nie wymagającą specyficznego wyrównania zmiennych w pamięci, czyli `MOVUPS`.



Użycie instrukcji wymagającej wyrównania może powodować, że program będzie losowo działał i rzucał wyjątek (nawet w kolejnych uruchomieniach będzie zachowywał się niepowtarzalnie), ponieważ może się tak zdarzyć, że zmienna akurat będzie poprawnie ułożona pod adresem podzielnym przez 16, przez przypadek.

1.7. Metoda weryfikacji wyników

Sprawdzenie debuggerem wraz z podglądem rejestrów / stanu pamięci oraz obserwacja wyników i sprawdzenie obliczeń.

1.8. Pomocne zasoby

Visual Studio 2019: https://visualstudio.microsoft.com/vs/

1)
https://visualstudio.microsoft.com/vs/
2)
https://docs.microsoft.com/en-us/dotnet/desktop/wpf/getting-started/walkthrough-my-first-wpf-desktop-application?view=netframeworkdesktop-4.8
3)
https://docs.microsoft.com/en-us/visualstudio/ide/how-to-change-the-build-output-directory?view=vs-2019

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

Permanent link:
https://dokuwiki.zmitac.aei.polsl.pl/pl:edu:zmitac:n1z:ja:lab4?rev=1618163639

Last update: **2021/04/11 19:53**