Instytut Informatyki Politechniki Śląskiej

Zespół Mikroinformatyki i Teorii Automatów Cyfrowych

Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Sekcja
2017/2018	**SSI**	**Języki Asemblerowe**	**5**	**9**

Imię:	**Dominik**	**Prowadzący**:	**AO**
Nazwisko:	**Dziembała**		

Raport końcowy

Temat projektu:

Szyfrowanie ROT13

Data oddania: dd/mm/rrrr	**21/01/2018**

1. Temat projektu

Tematem projektu jest aplikacja służąca do zaszyfrowania tekstu podanego przez użytkownika za pomocą szyfru ROT13.

Aplikacja została stworzona w środowisku Visual Studio 2015, użytym językiem programowania był język C#, a sam program jest w wersji 64bit. Aplikacja może zostać uruchomiona przez konsolę z parametrami startowymi w formacie „–i [ścieżka do pliku wejściowego] –o [ścieżka do pliku wyjściowego] –c [ilość wątków] lub przez podwójne kliknięcie. Po uruchomieniu aplikacja wykrywa dostępną ilość wątków procesora z zakresu od 1 do 64. Następnie użytkownik wybiera plik tekstowy, który ma zostać zaszyfrowany, oraz plik tekstowy w którym ma zostać zapisany zaszyfrowany tekst. Użytkownik ma również możliwość wybrania samemu ilości wątków, które zostaną użyte do zaszyfrowania tekstu wejściowego. Po wybraniu pliku wejściowego i wyjściowego zostają odblokowane 3 przyciski realizujące szyfrowanie ROT13 używając odpowiednich funkcji z bibliotek DLL. Zaszyfrowany tekst zostaje natychmiast po zakończeniu pracy algorytmu zapisany do pliku wejściowego, a w programie zostaje wyświetlony jego czas pracy.

Główna część aplikacji odpowiada za sprawdzenie poprawności plików wejściowych i wyjściowych, a także za wczytanie do pamięci zawartości szyfrowanego pliku tekstowego *.txt wskazanego przez użytkownika. Następnym etapem jest utworzenie tablicy wątków, podział danych oraz wywołanie odpowiednich funkcji bibliotecznej. Po wykonaniu szyfrowania tekst zostaje zapisany do pliku wyjściowego. Z racji tego, że wykorzystanym językiem programowania jest C#, pamięć zostaje zwolniona w momencie wywołania przez program Garbage Collector’a.

W celu zaszyfrowania tekstu użytkownik ma do dyspozycji 3 funkcje, 2 z biblioteki napisanej w języku asemblerowym lub 1 w języku C. Funkcje napisane w języku asemblerowym przyjmują jako argumenty szyfrowany tekst oraz jego długość, a funkcja napisana w języku C przyjmuje za argumenty szyfrowany tekst, zmienną w której zostaje zapisany tekst po zaszyfrowaniu, pozycja początkowa oraz pozycja końcowa. Ponadto do realizacji funkcji asemblerowej wykorzystującej wektory zostały użyte rejestry XMM.

2. Analiza zadania

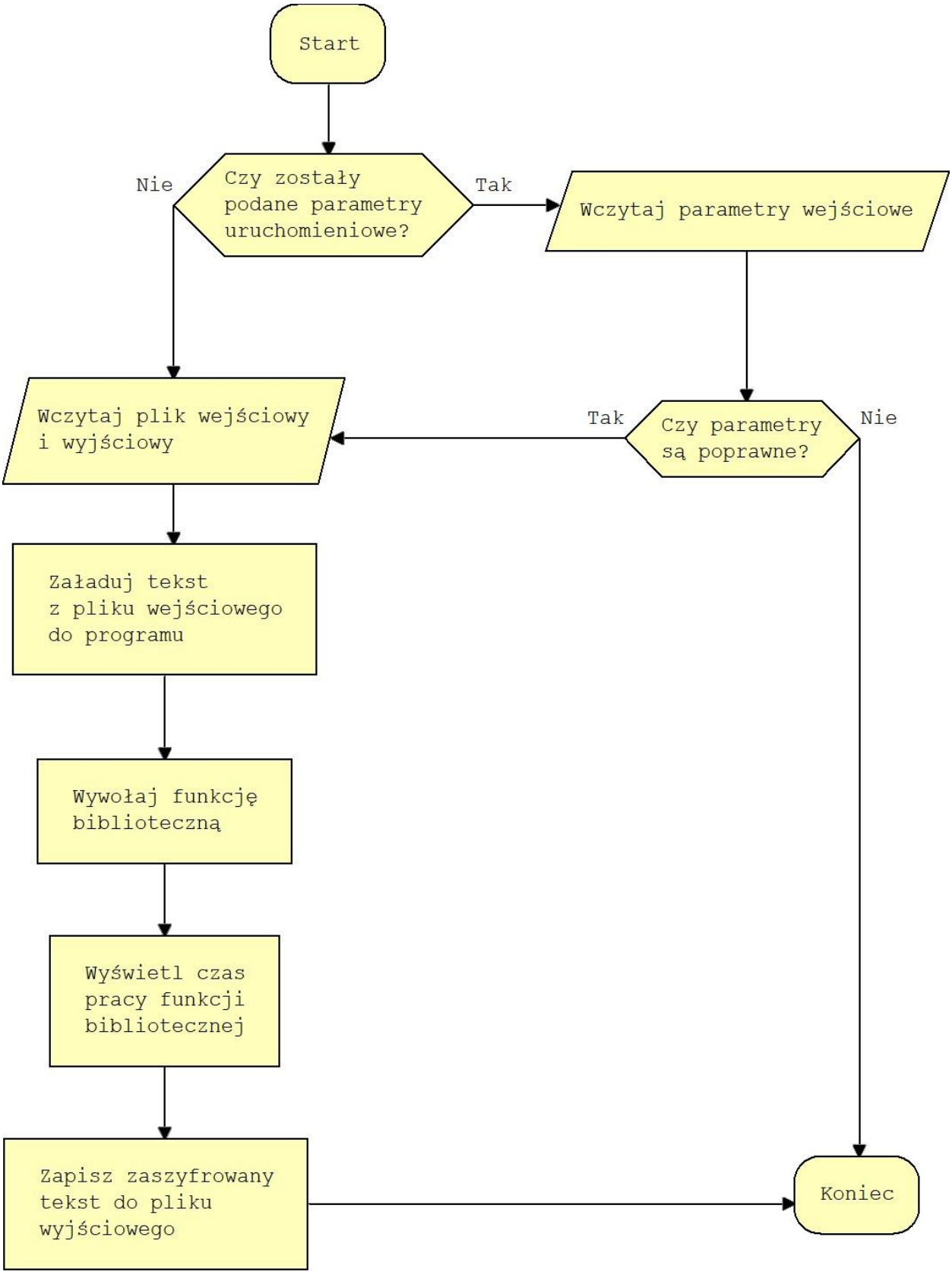
Aplikacja wykorzystuje do szyfrowania alfabet łaciński oraz rozróżnia wielkość liter, czyli wielka litera po zaszyfrowaniu pozostaje wielką literą, mała litera – małą. Szyfr ROT13 jest szyfrem przesuwającym. Jego działanie polega na zamianie każdego znaku alfabetu łacińskiego na znak występujący 13 pozycji po nim. Należy również wspomnieć, że szyfr ROT13 jest swoją własną funkcją odwrotną co oznacza, że ponowne użycie tego szyfrowania na już zaszyfrowanym tekście spowoduje jego ujawnienie. Jeżeli podczas szyfrowania litera przekroczy zakres alfabetu wtedy zostają użyte kolejne litery z jego początku. Można zauważyć, że od litery N/n zamiana litery na występującą 13 pozycji po niej przekracza zakres alfabet, więc zamiast przesuwania się 13 pozycji w przód możemy przesunąć się o tyle samo wstecz.

Sam podział na wątki pozwala również w łatwy sposób podzielić szyfrowany tekst na części odpowiednio od 1 do 64. Jeżeli nie da się podzielić danego tekstu na równe części wówczas wątki które są mniejsze lub równe wynikowi modulo dostają dodatkową literę.

O ile podczas implementacji algorytmu szyfrującego w języku C nie wystąpiły żadne problemy, o tyle podczas implementacji w języku asemblerowym był problem z przesłaniem i odebraniem tekstu. Problem został rozwiązany za pomocą słowa kluczowego *fixed*, które pozwala na wykorzystanie *char** w języku C#.

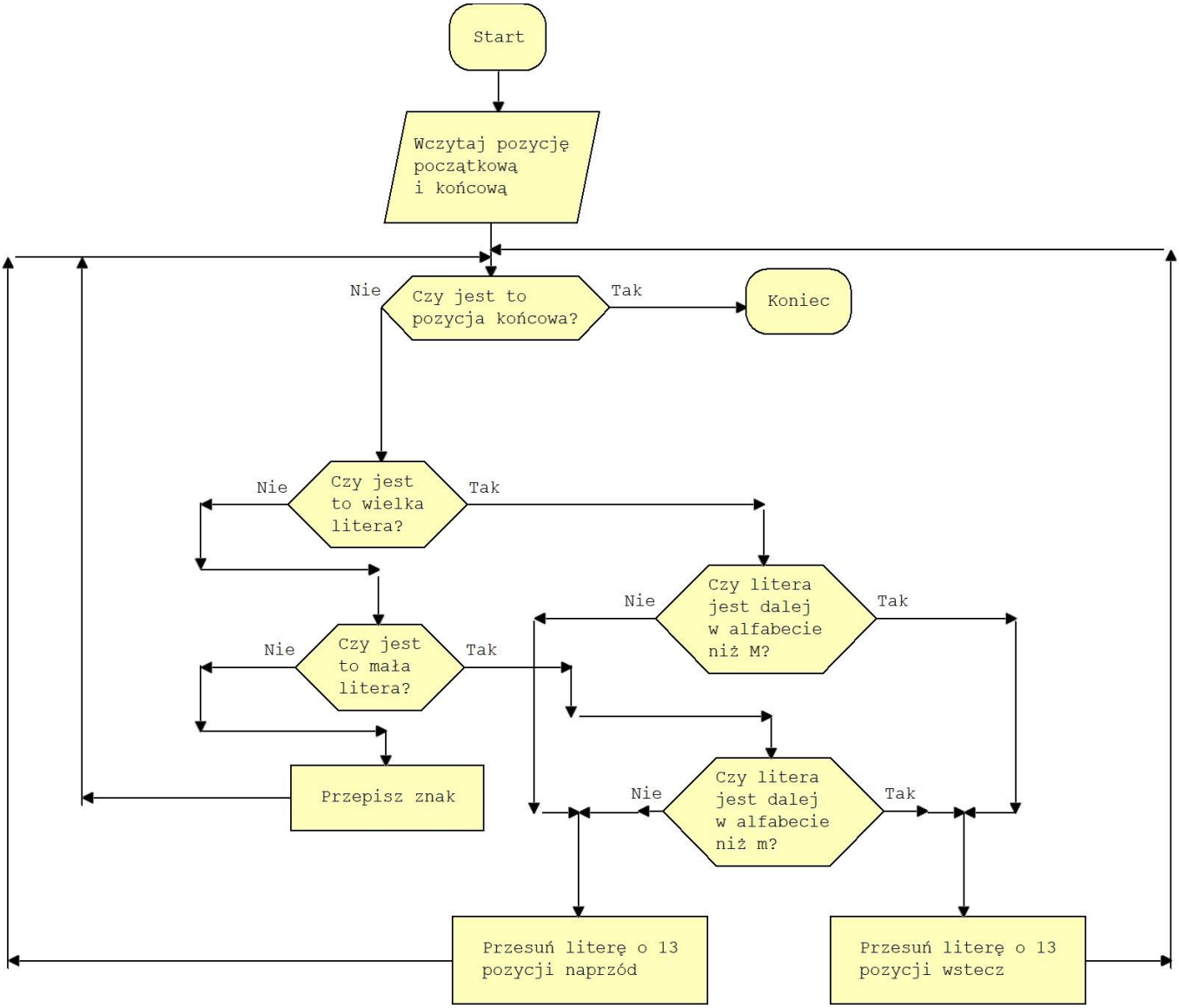
3. Schemat blokowy

3.1. Główny program



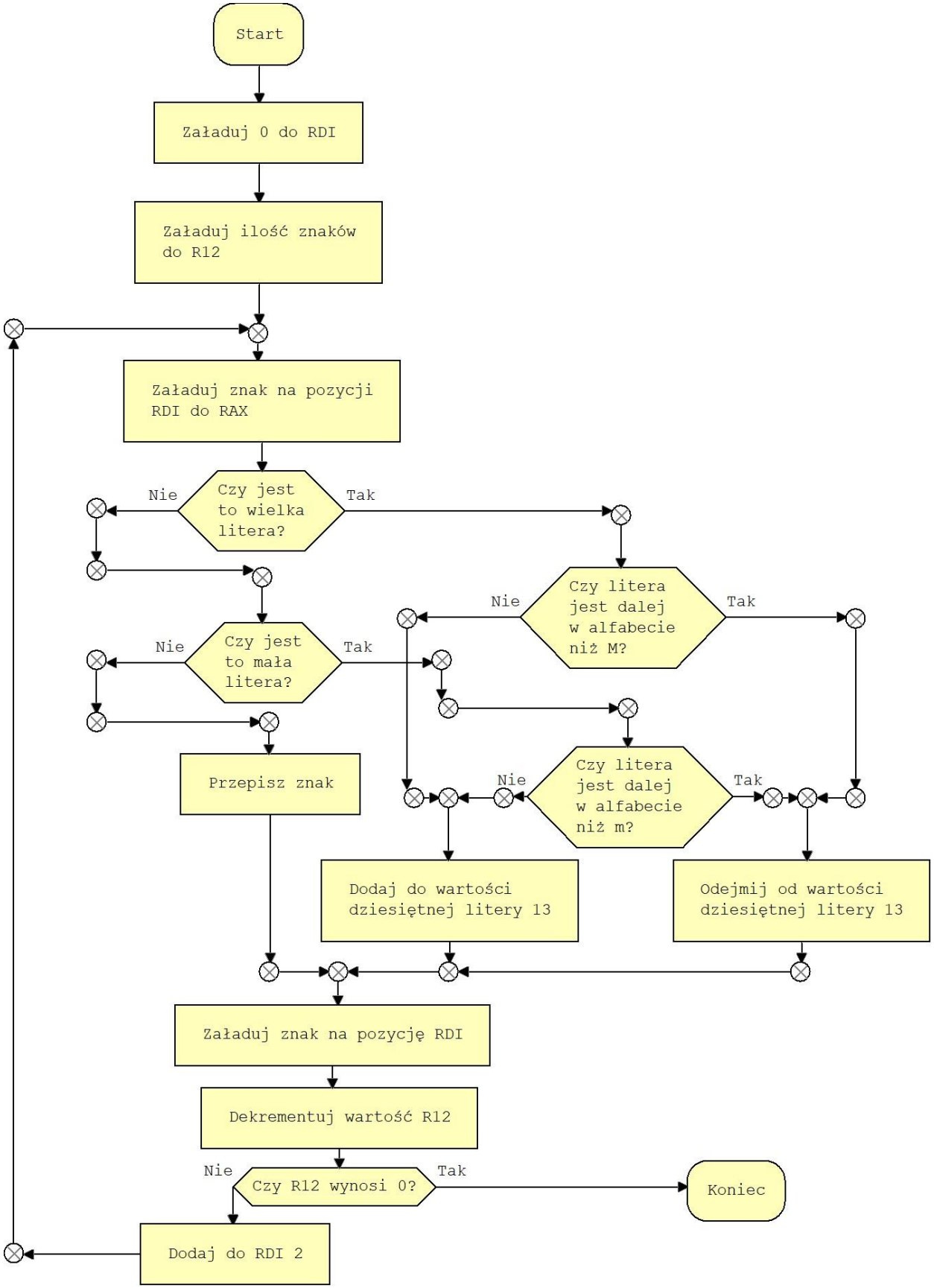
Schemat blokowy 1 Główny program.

3.2. Biblioteka C



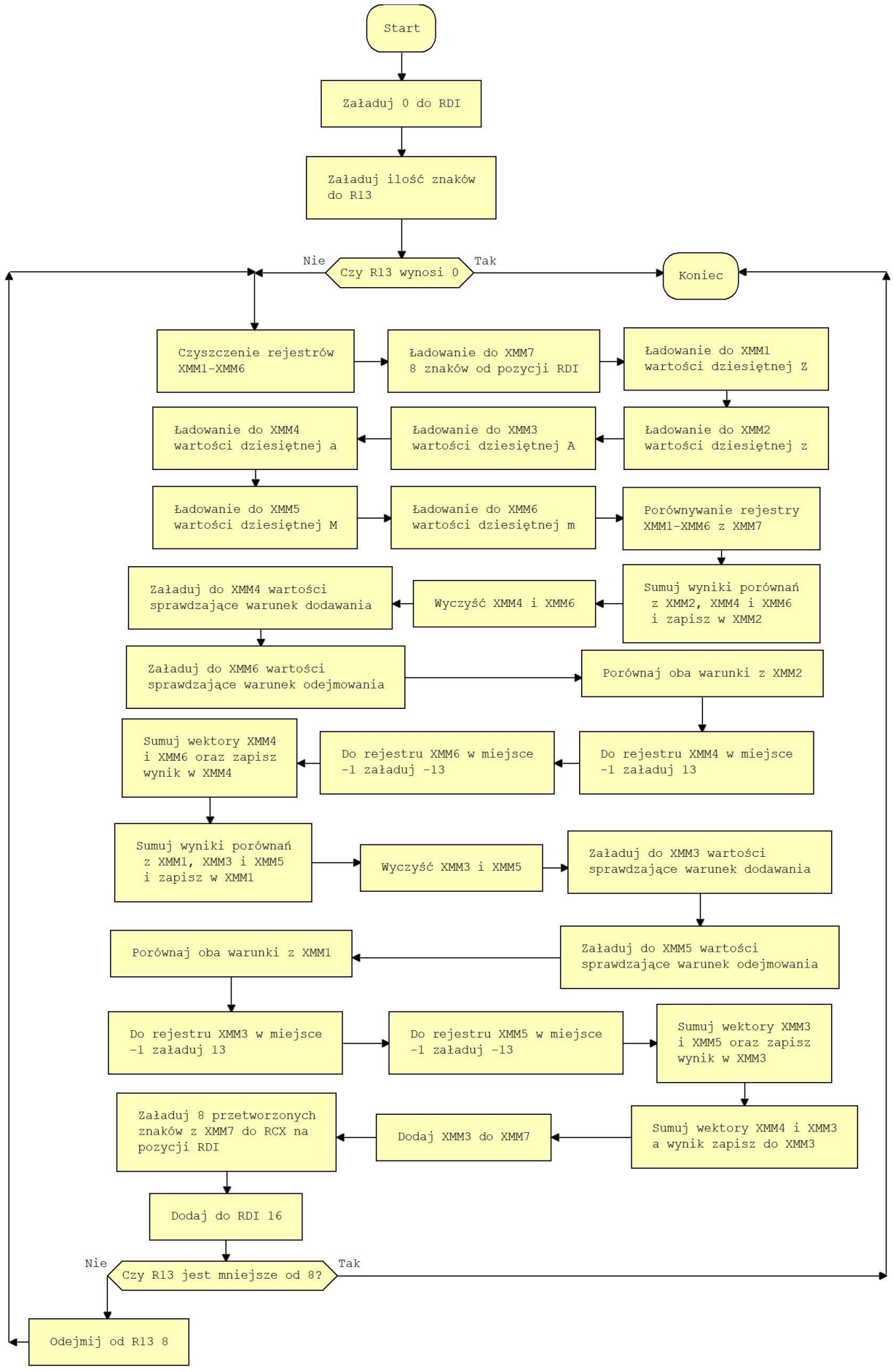
Schemat blokowy 2 Biblioteka C.

3.3. Biblioteka ASM



Schemat blokowy 3 Biblioteka ASM.

3.4. Biblioteka ASM z instrukcjami wektorowymi



Schemat blokowy 4 Biblioteka ASM z użyciem instrukcji wektorowych.

4. Analiza programu głównego

Program główny napisany został w języku C#. Na początku działania program zostaje sprawdzona ilość wątków jakimi dysponuje procesor. Później zostają sprawdzone parametry uruchomieniowe. Jeżeli parametry są poprawne program załaduje je, jeżeli nie są poprawne – program zostanie zakończony. W przypadku braku parametrów użytkownik będzie musiał podać przynajmniej ścieżkę do pliku wejściowego i wyjściowego.

Aplikacja umożliwia użytkownikowi wybranie plików tekstowych z wykorzystaniem graficznego obiektu interfejsu OpenFileDialog który wyświetla tylko pliki z rozszerzeniem txt. Po wybraniu pliku wejściowego oraz wyjściowego zostają odblokowane przyciski z wyborem poszczególnych funkcji bibliotecznych. Użytkownik może również wybrać ilość wątków za pomocą których zostanie zrealizowane szyfrowanie. Zakres wyboru ilości wątków jest od 1 do 64.

W momencie kliknięcia jednego z przycisków wywołujących funkcję biblioteczną zostaje utworzona tablica wątków. Następnie zostają wywołane wszystkie wątki z tablicy wraz z przekazanymi do nich parametrami. W tym momencie następuje podział tekstu szyfrowanego na N części, których ilość zależy od ilości wybranych wątków. Jeżeli ilość liter jest mniejsza od ilości wątków, wówczas niektóre wątki nie wykonają żadnego przekształcania tekstu. W momencie wywołania wątków zostaje uruchomione odliczania czasu pracy konkretnego algorytmu. Czas zostaje zatrzymany w momencie ukończenie szyfrowania przez wszystkie wątki. Po zakończeniu wszystkie N części zostają scalone razem w jeden tekst wynikowy, który następnie zostaje zapisany w pliku wyjściowym. Na sam koniec zostaje pasek postępu przy wybranym przycisku zapełnia się symbolizując wykonanie się szyfrowania, a obok niego wyświetlany zostaje czas pracy w milisekundach.

5. Analiza bibliotek DLL

5.1. Biblioteka DLL C

Jako że algorytm szyfrujący operuje tylko na literach alfabetu łacińskiego, biblioteka w języku C składa się wyłącznie z jednej funkcji cEncodeText. Jako argumenty przyjmuje ona kolejno tekst szyfrowany, zmienną w której zostanie zapisany rezultat szyfrowania, pozycję początkową oraz pozycję końcową.

Początkowo funkcja pobiera pozycję początkową i końcową. Pozycje te zostaną wykorzystane w pętli *for* do przemieszczania się po kolejnych elementach tekstu szyfrowanego. Dodatkowo zostaje wykorzystana zmienna *j*, która odpowiada za aktualny indeks w tekście wynikowym.

W każdym przebiegu pętli analizowany jest kolejny znak z tekstu wejściowego. Zostaje on poddany instrukcjom warunkowym, które sprawdzają czy jest to wielka czy mała litera. Jeżeli znak zostanie rozpoznany jako litera, wówczas przechodzi on do kolejnego testu, tym razem sprawdza się warunek czy litera występuje po literze M/m. Gdy instrukcja warunkowa *if* zwróci *true,* litera zostaje zastąpiona w tekście wynikowym na znak znajdujący się 13 pozycji wcześniej, a w przypadku wartości *false* w tekście wynikowym zostanie zapisana litera występująca 13 pozycji dalej. Jeżeli sprawdzany znak nie został rozpoznany jako litera do tekstu wynikowego zostaje przepisany ów znak bez żadnej zmiany. Na koniec każdej iteracji zmienna *j* zostaje inkrementowana.

5.2. Biblioteka DLL ASM

Biblioteka w języku asemblerowym posiada 2 niezależne procedury. Jedna z nich implementuje algorytm szyfrowania ROT13 z użyciem instrukcji wektorowych, druga bez korzystania z nich.

Procedura nie korzystająca z rejestrów XMM ma nazwę encodeText. Przed wykonaniem jakichkolwiek przekształceń tekstu, na stos wysyłane są zawartości rejestrów rcx, rdi oraz r12. Później do rdi zostaje zapisane 0 – od teraz rdi będzie służyło jako „skoczek” po kolejnych literach w tekście. Do r12 wpisana zostaje ilość liter które mają zostać przeanalizowane.

Następnie procedura wchodzi w pętle, w której na początku ładujemy do rax numer znaku. Jego wartość dziesiętną odczytujemy za pomocą rejestru al. Wartość ta jest sprawdzana za pomocą instrukcji *cmp*, która porównuje dwie wartości ze sobą, oraz odpowiednich instrukcji skoków warunkowych *ja, jae, jb, jbe*. Wykorzystując skoki warunkowe procedura skacze do odpowiednich etykiet, które po rozpoznaniu że dany znak to litera dodają lub odejmują 13. W przeciwnym wypadku następuje znak nie zostaje zmieniony.

Po ukończeniu operacji nad znakiem zostaje sprawdzane warunek końca pętli. Jeżeli pętla zostanie opuszczona, wówczas ładujemy zaszyfrowany fragment tekstu do rax i ładujemy wcześniej wysłane zawartości rejestrów ze stosu ponownie do tych rejestrów.

Przy procedurze korzystającej z rejestrów XMM o nazwie vectorialEncodeText na początku tworzymy sobie tablice z wartościami liter, wartościami testów czy należy odjąć trzynaście czy dodać oraz tablice z wartościami -13 i 13.

Po deklaracji danych wysyłamy zawartość rejestrów rcx, rdi i r13 na stos. Rejestr rdi w tym przypadku również posłuży za „skoczka” po kolejnych znakach w tekście wejściowym, a rejestr r13 będzie przechowywał informację o ilość tych znaków.

Kolejnym krokiem jest sprawdzenie czy ilość znaków jest różna od 0. W zależności od wyniku wchodzimy do pętli lub kończymy działanie procedury. Wchodząc do pętli pierwszym zadaniem procedury jest wyczyszczenie zawartości rejestrów XMM1 – XMM6. Do XMM7

ładowane jest 8 znaków z tekstu szyfrowanego. Później do rejestrów XMM1 – XMM6 ładowane są wartości graniczne z alfabetu łacińskiego. Następnym etapem jest porównanie za pomocą instrukcji *pcmpgtb* tych rejestrów z XMM7. Sumując wartości porównań rejestrów przechowujących wartości graniczne, dla liter małych w XMM2 i dla dużych w XMM1, oraz porównując je z wartościami testów czy należy dodać czy odjąć trzynaście, możemy stwierdzić na których pozycjach znajdują się konkretne litery. Kolejnym krokiem jest zapisanie informacji o działaniach na znakach dla liter małych w XMM4, dla liter wielkich w XMM3. Sumując oba te rejestry dostajemy pełną informacje o działaniach na wszystkich 8 znakach. Suma ta jest zapisywana w rejestrze XMM3. Ten rejestr jest następnym etapie dodawany do XMM7. Zaszyfrowany fragment jest wracany do rcx, „skoczek" jest przesuwany oraz sprawdzany jest warunek końca pętli. Jeżeli ilość znaków jest mniejsza od 8 przed pomniejszeniem tej liczby o już zaszyfrowane znaki to opuszczamy pętle.

Również i tutaj po opuszczeniu pętli ładujemy do rax zaszyfrowany tekst oraz pobieram zawartość wcześniej wysłanych rejestrów ze stosu.

6. Struktura danych wejściowych

Program za dane wejściowe przyjmuje pliki o rozszerzenie txt. Jeżeli użytkownik skorzysta z graficznego wyboru pliku, zostaną mu tylko wyświetlone pliki o tym właśnie rozszerzeniu. W przypadku, gdy aplikacja zostanie uruchomiona z parametrami startowymi, zostaną one sprawdzone pod względem poprawności. Jeśli plik wejściowy nie istnieje lub ma złe rozszerzenie program zakończy działanie.

7. Uruchamianie programu oraz testowanie

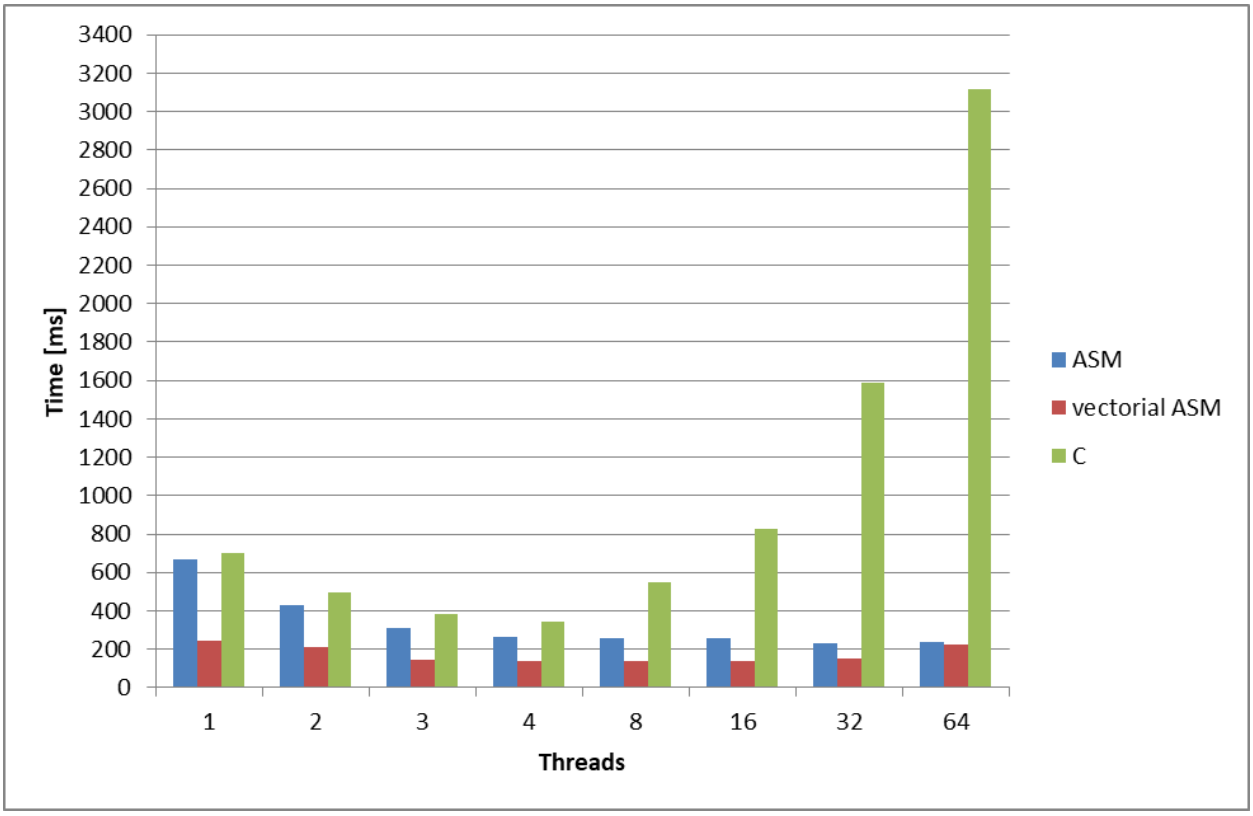
Aplikacja może zostać uruchomiona na dwa sposoby: za pomocą konsoli i dodatkowych parametrów uruchomieniowych lub za pomocą pliku wykonywalnego z rozszerzeniem exe. Jedynym wymogiem jest by biblioteki DLL_ASM.dll oraz DLL_C.dll znajdowały się w tym samym katalogu co plik wykonywalny.

Program był testowany wielokrotnie zarówno pod względem poprawności działania, jak i efektywności algorytmów.

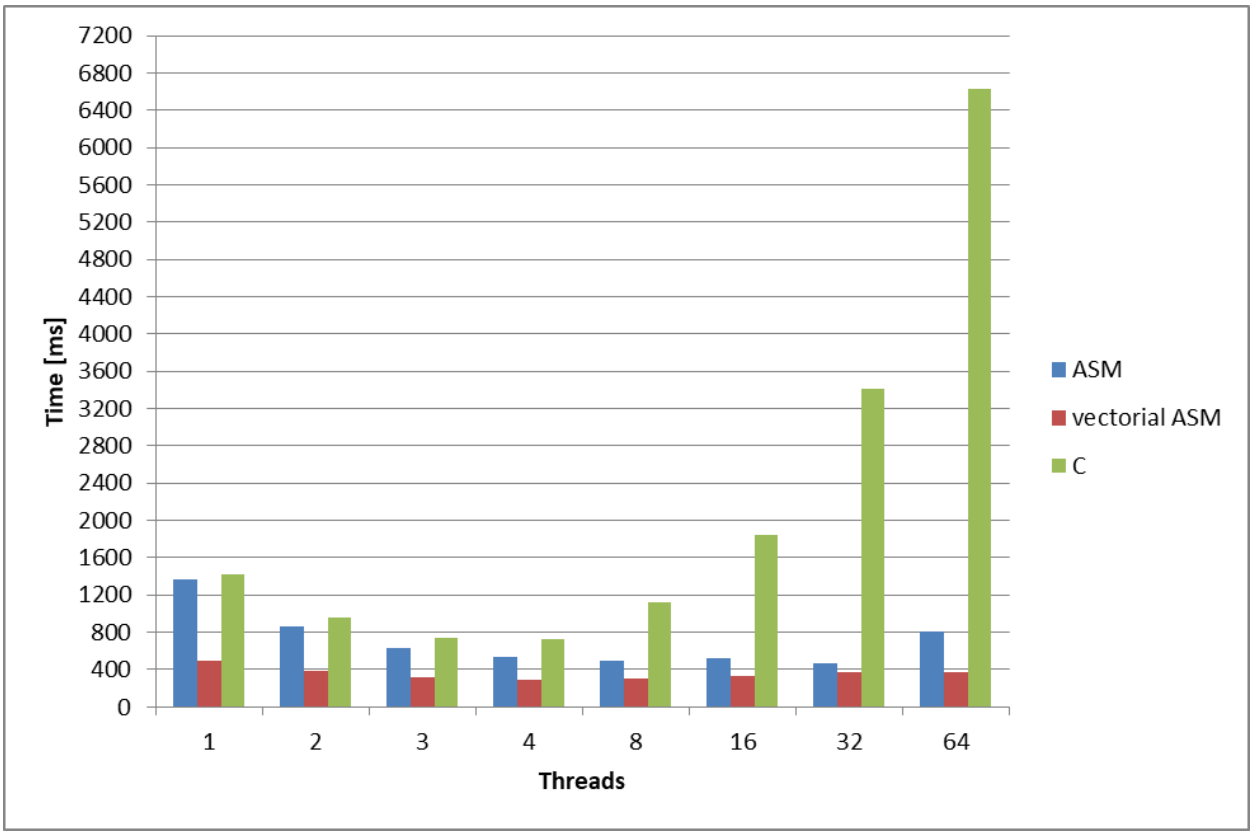
Graficzny wybór plików został zabezpieczony by użytkownik nie mógł wybrać niewłaściwego pliku. Niewłaściwy wybór pliku za pośrednictwem parametrów zawsze skutkuje automatycznym zamknięciem programu.

Również wybór ilości wątków został zabezpieczony przez wykorzystanie suwaka, który umożliwia wybór ilości od 1 do 64 wątków.

8. Wykresy czasowe



Wykres 1 plik 64MB



Wykres 2 plik 128MB

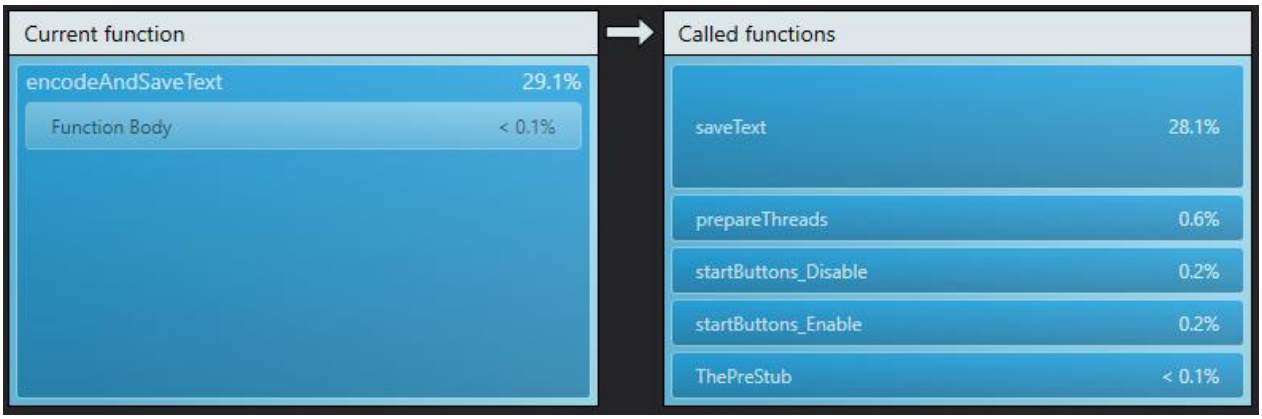
Wyniki czasowe zostały wyznaczone na podstawie średniej arytmetycznej z 10 próbek przeprowadzonych dla każdej z wymienionej ilości wątków. Jak widać na wykresach wraz ze

wzrostem ilości wątków, do ilości jaką posiada dany procesor, czasy wykonywania każdej z funkcji bibliotecznych maleją. Dopiero po przekroczeniu maksymalnej liczby wątków jakie posiada procesor czasy zaczynają rosnąć.

9. Analiza działania programu

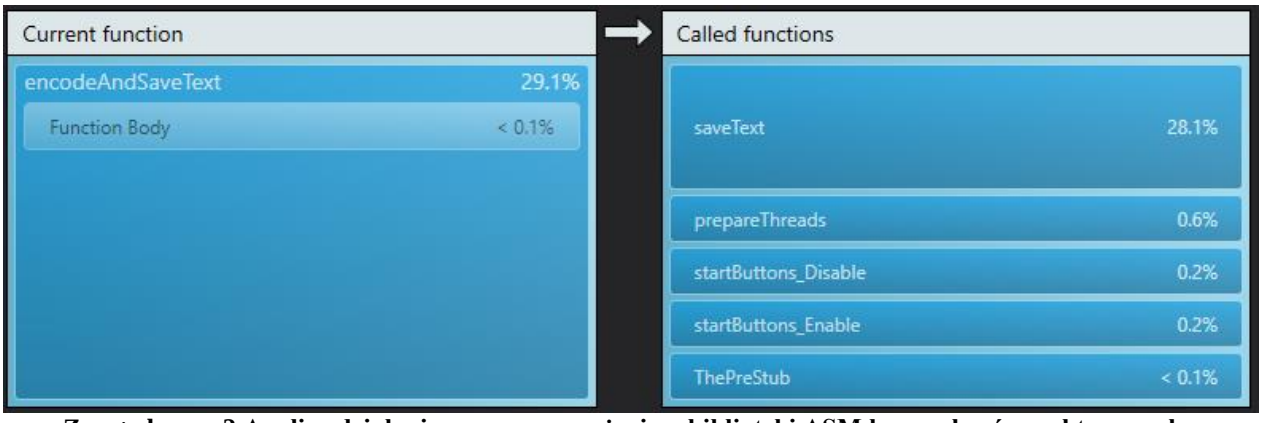
Program został poddany analizie działania pod kątem wydajności.

9.1. Biblioteka C



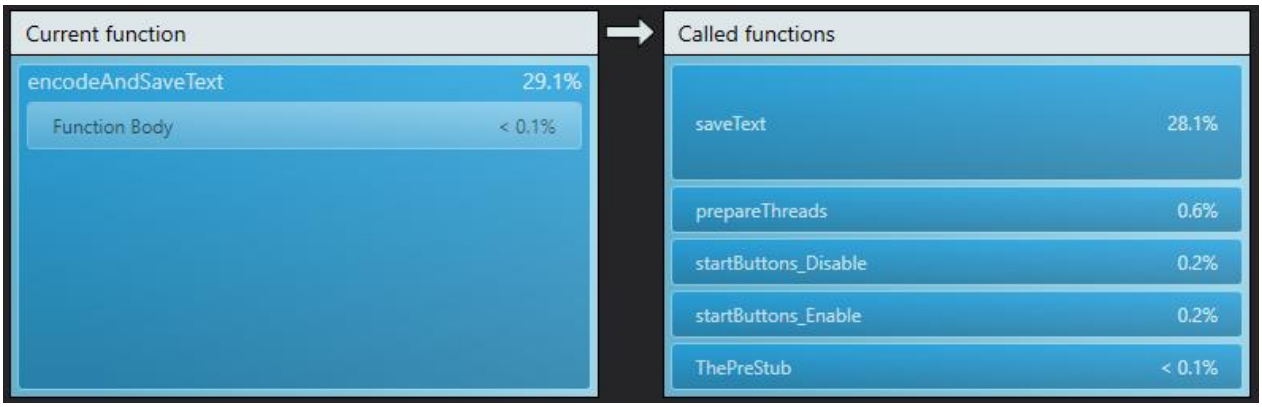
Zrzut ekranu 1 Analiza działania programu z użyciem biblioteki C.

9.2. Biblioteka ASM



Zrzut ekranu 2 Analiza działania programu z użyciem biblioteki ASM bez rozkazów wektorowych.

9.3. Biblioteka ASM z użyciem rozkazów wektorowych



Zrzut ekranu 3 Analiza działania program z użyciem biblioteki ASM oraz z wykorzystaniem instrukcji wektorowych.

9.4. Wnioski z analiz

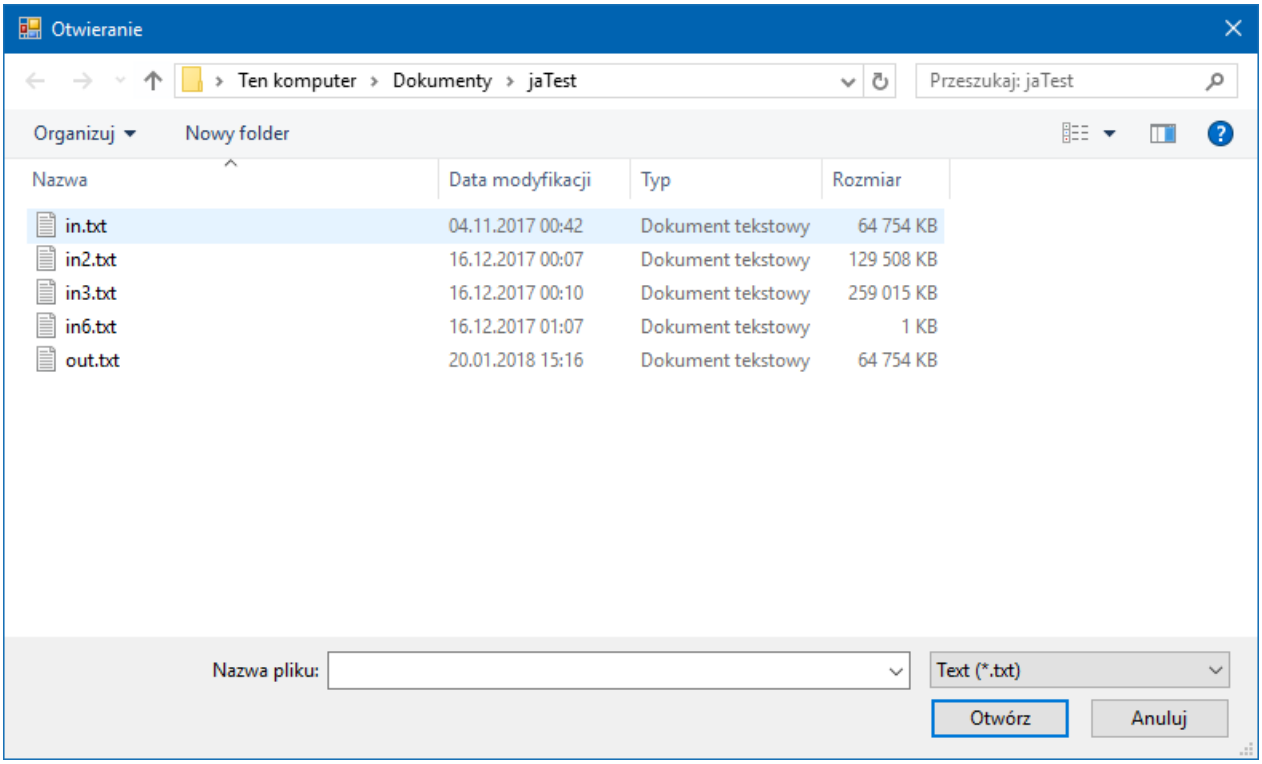
W przypadku każdej z bibliotek najbardziej obciążającym momentem dla procesora jest zapis do pliku. Jest to jednak element, którego nie dało się bardziej zoptymalizować. Do powyższych testów został użyty plik tekstowy o rozmiarze ok. 64MB. Jednak wraz ze wzrostem rozmiaru pliku rośnie czas zapisu zaszyfrowanego tekstu.

10. Instrukcja obsługi

W momencie uruchomienia aplikacji, sprawdzana jest ilość wątków jakie oferuje dany procesor. Jeżeli aplikacja została wywołana z parametrami uruchomieniowymi, wówczas sprawdzana jest ich poprawność, jeżeli zostanie wykryty błąd, aplikacja zostaje zamknięta.

Zrzut ekranu 4 Interfejs graficzny aplikacji.

Następną czynnością jaką należy wykonać jest wybranie pliku wejściowego oraz pliku wyjściowego. Po naciśnięciu przycisku *Select* zostanie otworzone okno wyboru pliku, które wyświetla tylko pliki o rozszerzeniu txt.



Zrzut ekranu 5 Okno wyboru pliku.

Po wybraniu plików i załadowaniu tekstu szyfrowanego do pamięci zostają odblokowane przyciski wywołujące funkcje biblioteczne.

Zrzut ekranu 6 Wygląd interfejsu aplikacji po wybraniu plików.

Ostatnim etapem jest wybór funkcji bibliotecznej jaka zostanie użyta do zaszyfrowania pliku wejściowego. Po jego naciśnięciu pasek postępu przesuwa się aż do 100% i wyświetlany jest czas pracy użytej funkcji bibliotecznej.

Zrzut ekranu 7 Wygląd interfejsu aplikacji po wykonaniu szyfrowania.

11. Wnioski

Projekt realizuje wszystkie założenia z karty projektu, plik wyjściowy po wykonaniu dowolnej funkcji bibliotecznej zawiera poprawnie zaszyfrowany tekst. Analizując wykresy czasowe można zauważyć jak bardzo przydały się instrukcje wektorowe w bibliotece asemblerowej. Z przyczyn technicznych nie można było wykorzystać rejestrów YMM, które z pewnością jeszcze bardziej zwiększyłyby wydajność tej procedury. Pomimo wielkości plików wejściowych samo szyfrowanie trwa wystarczająco szybko, by nie zauważyć jakiejkolwiek przerwie w pracy programu. Jednak sam zapis zaszyfrowanego tekstu do pliku przy większych plikach trwa dość długo, co da się zaobserwować nawet kilkudziesięciu sekundową przerwą w działaniu. Samo ładowanie danych do programu nie trwa zbyt długo. Przy obecnym procesorze oraz dostępnej pamięci RAM maksymalny rozmiar pliku jaki mógł być testowany na moim komputerze wynosił około 128MB, powyżej czas oczekiwania na wyniki był uciążliwie długi.

Uważam, że warto stosować w aplikacjach kod napisany w języku asemblera, ponieważ pozwala to bardzo zoptymalizować cały algorytm. W przyszłych projektach, gdy będzie to możliwe, będę próbował zoptymalizować algorytm za pomocą instrukcji wektorowych z języka asemblera.