
	Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki:	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot (Języki Asemblerowe/SMiW):	Grupa	Sekcja
<b>2019/2020</b>	<b>SSI</b>	<b>SMiW</b>	<b>3</b>	<b>5</b>
Imię:	<b>Paweł</b>	<b>Prowadzący:</b> OA/JP/KT/GD/BSz/GB	<b>BZ</b>	
Nazwisko:	<b>Zawierucha</b>			
<h2 style="text-align: center;"><i><b>Raport końcowy</b></i></h2>				
<p><b>Temat projektu:</b></p> <h1 style="text-align: center;">Efekt echa nakładany na pliki .wav</h1>				
<b>Data oddania:</b> dd/mm/rrrr				

# Założenia

Celem projektu było stworzenie dwóch bibliotek dynamicznych: w C++ oraz w Assemblerze, dodających efekt echa do pliku wejściowego .wav i zapisujący wynik do pliku wyjściowego .wav. Projekt ma działać na wielu wątkach w trybie Release x64 oraz korzystać z jak najbardziej optymalnych instrukcji dostępnych dla naszego procesora (w moim przypadku był to instrukcje operujące na rejestrach ymm)

## Analiza zadania

Zadanie polega na dodaniu efektu echa do dźwięku zapisanego w formacie .wav. Na początku należy poprawnie wczytać nagłówek tego pliku oraz dane w nim zawarte, które następnie są dzielone dla poszczególnych wątków (co jest realizowane w głównym programie). Następnie postanowiłem korzystać z następującego algorytmu:

1. Wczytanie danych z pliku wejściowego
2. Wczytanie już przetworzonych danych z pliku wyjściowego
3. Ściszenie już przetworzonych dźwięków z pkt. 2. (aby echo nie trwało w nieskończoność)
4. Połączenie danych z punktu 1. i 2. oraz zapis ich do pliku wyjściowego w dwóch miejscach:
  - a) w miejscu odpowiadającym aktualnym danym z pliku wejściowego w pliku wyjściowym
  - b) przesuniętych o okres trwania echa

Działania te wykonywane do momentu przetworzenia całego pliku wejściowego.

## Specyfikacja wewnętrzna

Główny program napisany w c++ odpowiada za poprawne wczytanie nagłówka tego pliku oraz danych w nim zawartych, które następnie są dzielone dla poszczególnych wątków.

Uruchamiając program można wybrać z której Dll będziemy chcieli skorzystać (cpp/asm) oraz ilości wątków na jakiej ma działać program, domyślnie będzie to ilość optymalna dla komputera.

-i ścieżka do pliku wejściowego .wav  
-d wybrana biblioteka  
-t ilość wątków  
-x ilość prób (dla sprawdzenia prędkości działania programu)

Implementacja funkcji w bibliotece ASM:

```
Delay PROC Source: QWORD, dataSize: QWORD, destination: QWORD, myBegin:
QWORD, myEnd: QWORD, delayStep: QWORD, iterationLength: QWORD
;Source - table address (not your begin)
;dataSize - number of bits (or bytes?) in source table
;destintion - addres of output table
;myBegin - starting point for this thread
;myBegin - ending point for this thread
;delayStep - range of one delay loop (adding one echo)
```

```
;initial for procedure - required
sub    rsp, 8
push   rbx
push   rbp
```

;move arguments from registers to variables

```
mov Source, rcx
mov dataSize, rdx
mov destination, r8
mov myBegin, r9
mov r10, myEnd
mov r12, delayStep
mov r14, delayStep
sub r14, iterationLength
```

```
mov r13,rcx ;move input adres
add r13,rdx ;add input size (to make end of input)
mov iterationLength, r13
```

```
mov r15,r8 ;bufor do zapisu do przodu
add r15,r12
```

```
add r10,rcx ; zeby myend to byl koniec dla rcx
mov myEnd, r10
```

```
add rcx,r9
add r8,r9
add r15,r9
```

MainLoop:

```
vmovdqu ymm0,ymmword ptr [rcx] ;wczytaj dane z inputa
vmovdqu ymm1,ymmword ptr [r8] ;wczytaj dane z outputa
```

VPSRAW YMM1, YMM1, 1 ;przesuniecie bitowe arytmetyczne w prawo o 1 bit  
(sciszenie)

```
vpaddw ymm2,ymm1,ymm0 ;ymm2 - wynik to outputu
vmovdqu ymmword ptr[r8],ymm2 ; zapis do outputu
vmovdqu ymmword ptr[r15],ymm2 ; zapis do outputu do przodu
```

```
add rcx,32 ;przesuniecie o 16 wartosci (rozmiar ymm)
add r8, 32
add r15,32
```

```
mov Source, rcx ;debug
mov destination, r8
mov Source, rcx
```

```
cmp rcx,r10 ; sprawdz czy input nie jest poza swoja granica
jge addingNextDelay
jmp MainLoop
```

addingNextDelay:

```
add rcx, r14 ;przesuwa wskazniki do nastepnej iteracji o delaystep -
length
add r8, r14
add r15, r14
add r10,r12 ; tylko delaystep
```

```

mov Source, rcx
mov destination, r8
mov myEnd, r10

```

```

    cmp rcx,r13      ;sprawdz czy input nie jest poza tablica
    jge finishProcedure
    jmp MainLoop

```

```

finishProcedure:

```

```

    pop rbp
    pop rbx

```

```

    add rsp,28
    ret

```

```

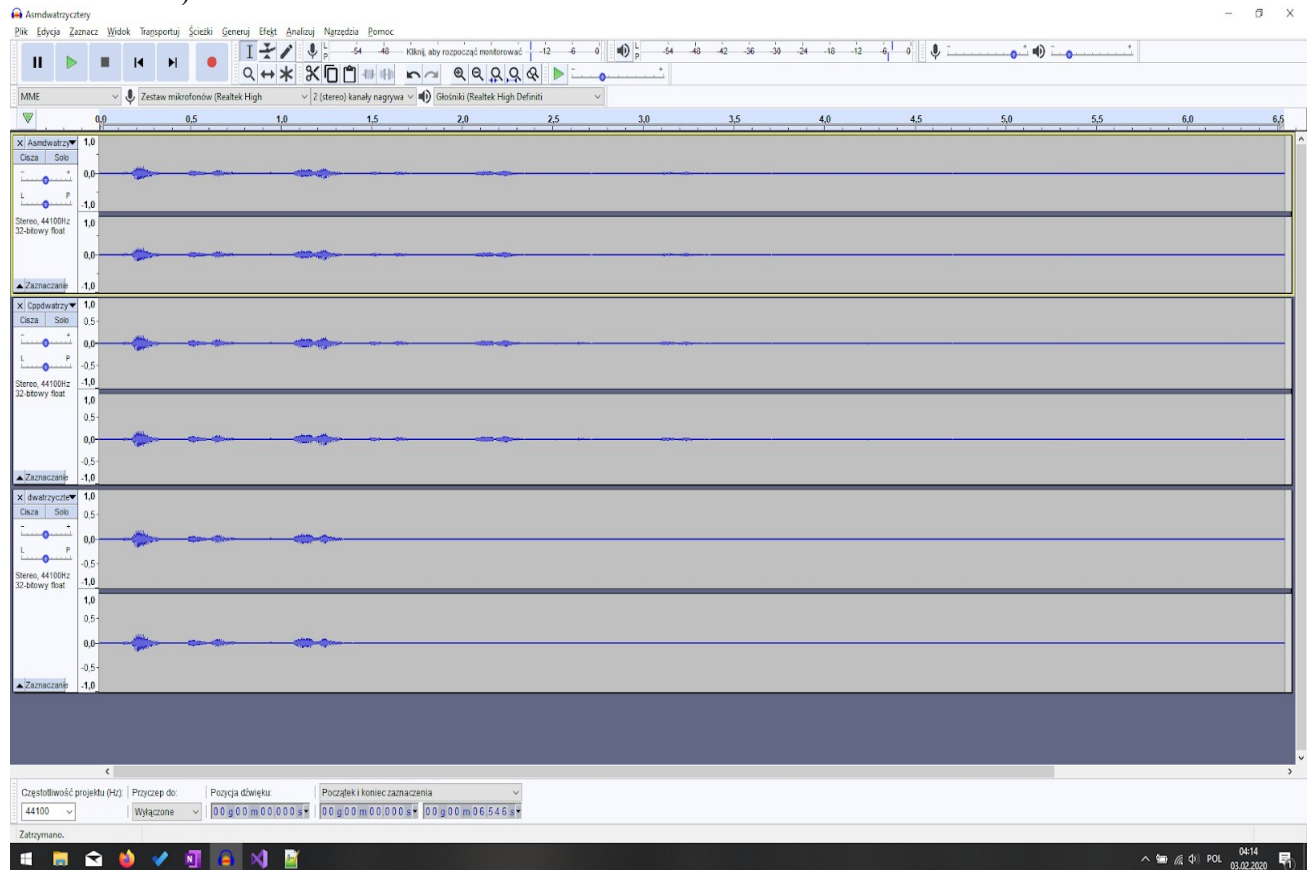
Delay ENDP

```

## Specyfikacja zewnętrzna

Program uruchamiany jest z linii poleceń z użytymi przełącznikami. W przypadku ich braku wybrane zostają pliki domyślne oraz optymalna ilość rdzeni.

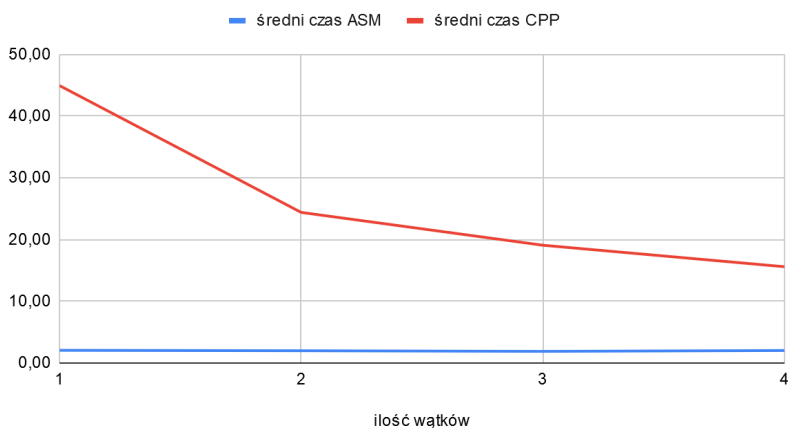
Wyniki działania programu wyglądają następująco: (na dole oryginalny plik, nad nim wyniki działania obu bibliotek)



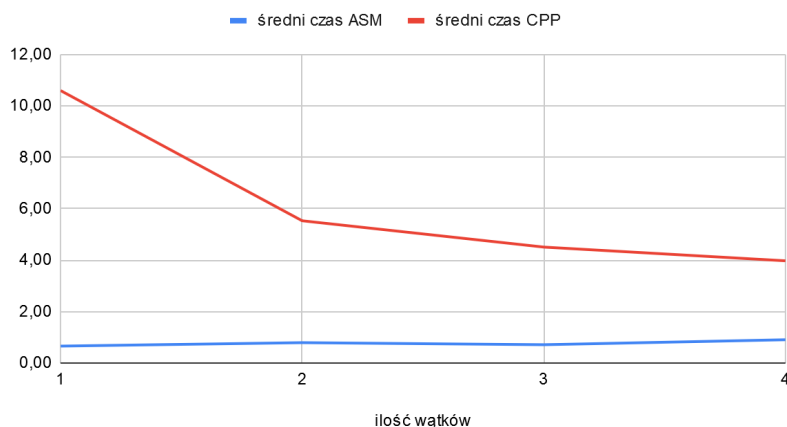
# Testowanie

Program został przetestowany dla plików krótkich (kilka sekund), dłuższych (kilkadziesiąt sekund) oraz bardzo długich (ponad godzina).

Czas dla długiego pliku

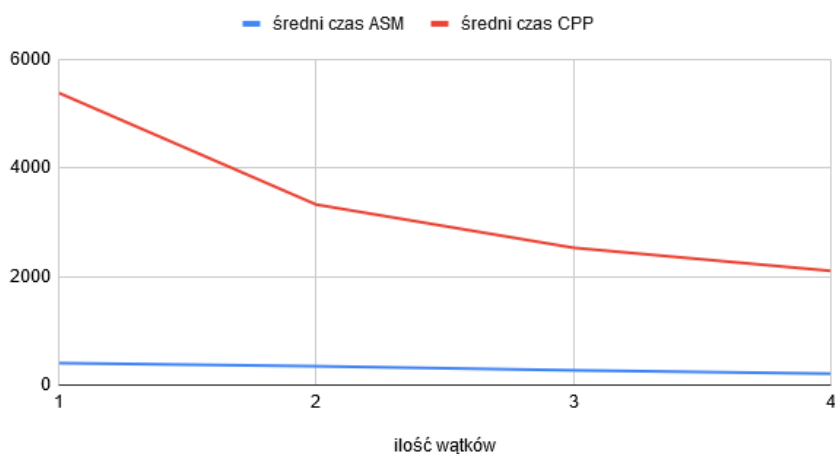


Czas dla krótkiego pliku



H	I	J	K	L
	ilość wątków	średni czas ASM	średni czas CPP	
	1	403,65	5381,2	
	2	346,11	3 324,56	
	3	270,60	2 528,21	
	4	208,81	2 102,69	

Czas dla bardzo długiego pliku



# Wnioski

Korzystanie z assemblera podczas tworzenia programów jest relatywnie trudne oraz czasochłonne ze względu na słabszą dokumentację w porównaniu do języków wysokiego poziomu, ale w zamian może oferować olbrzymi wzrost prędkości działania programu (w moim przypadku aż 10 – krotny). Podczas pracy z dźwiękiem dosyć proste jest odkrycie tego że błąd istnieje, ale znalezienie jego przyczyny jest już dosyć długim procesem.