Cwiczenie 2

Zaawansowane architektury komputerowe

Potoki i predykcja w procesorach o architekturze RISC

1 Wprowadzenie

Poprzednie ćwiczenie laboratoryjne polegało na projektowaniu architektury szeregowej. Bieżące ćwiczenie laboratoryjne pozwoli na przeprowadzenie projektowania architektury potokowej. Po zakończeniu ćwiczenia student będzie mógł samodzielnie zaprojektować architekturę potokową prostego procesora RISC. Jako przykład zostanie wykorzystany kod z poprzednich zajęć. Najistotniejszym elementem tego ćwiczenia jest przyjrzenie się metodzie rozbijania kodu na poszczególne potoki, oraz obserwacja tego jak kod wysokopoziomowy może zostać zoptymalizowany z wykorzystaniem mechanizmu potoków i predykcji:

- zmiany nazw rejestrow
- przestawiania kolejności wykonywania instrukcji
- techniki *forwarding* (*register forwarding*, przełączanie rejestrów)

W architekturze potokowej faza ID odpowiedzialna jest za przekazanie parametrów formalnych rozkazu do rejestrow A,B oraz za zdekodowanie rozkazu (STn). W fazie EX odbywa się wykonanie rozkazu w ALU i COMP, faza MEM odpowiedzialna jest za współpracę z pamięcią programu (czytanie rozkazów odbywa się w jednym cyklu zegarowym) we/wy, WB uaktualnia rejestry.

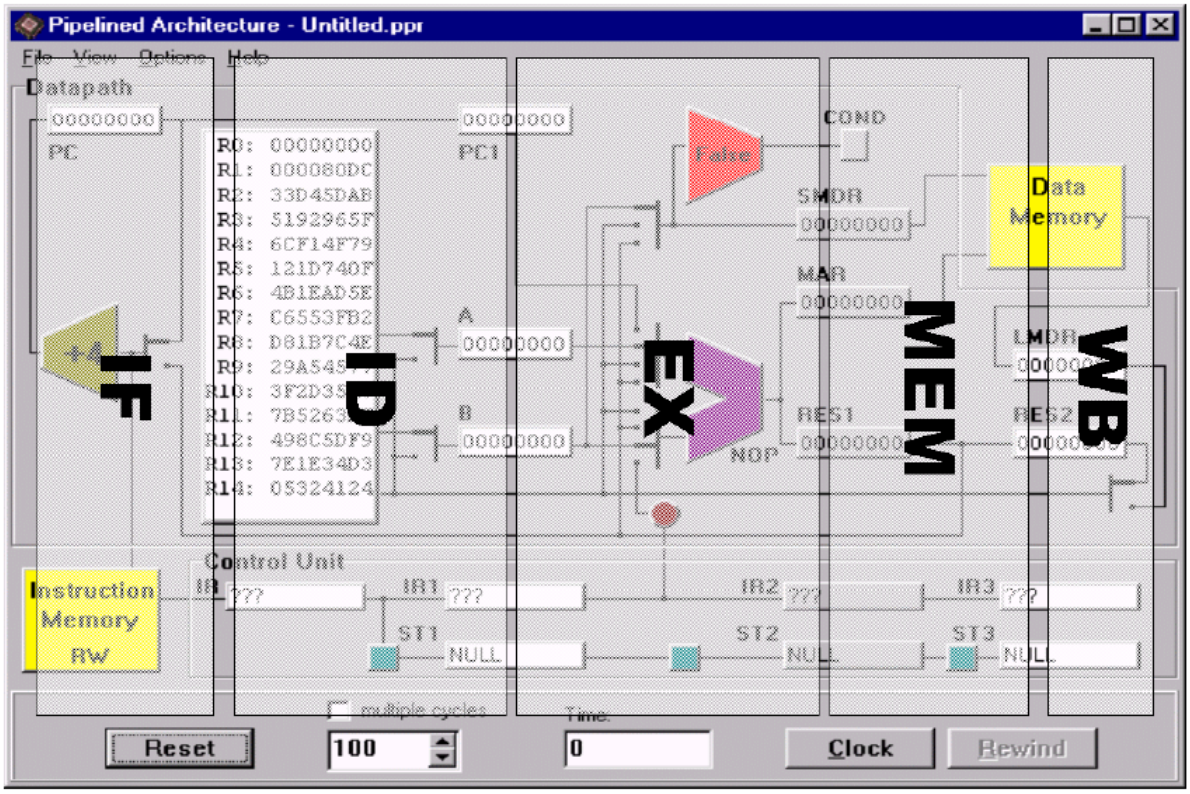
1.1 Przygotowanie ćwiczenia

• Pobierz materiały ze strony internetowej: .zip
• Rozpakuj materiały w katalogu roboczym
• Uruchom symulator

2 Zadania do wykonania

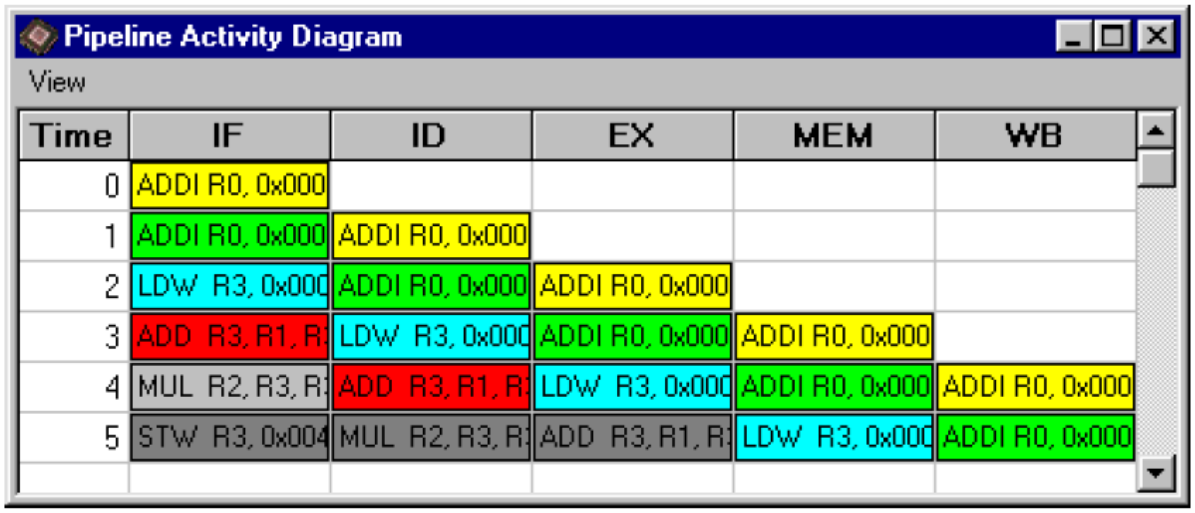
2.1 Konfiguracja wstępna

Uruchamiając symulator, przeprowadź konfigurację środowiska wykorzystując plik konfiguracyjny **pipe.ecf**. Konfiguracja środowiska przedstawiona jest na Rys. 1. Następnie uruchom projekt **pipe2.ppr**.



Rys 1: Główne okno programu systemu o architekturze potokowej Zaznaczone są na nim główne fazy potoku

Zwróć uwagę na rejestry w tej architekturze (A, B, STn, RESn,...).



Rys 2: Okno Activity Diagram – Architektury potokowej

Tabela 1: Potok – zakleszczenie

Typ zakleszczenia	Czas	Przyczyna
EX	2	ADD R1,R6,R1 czeka na wejscie: R6; spowodowane ADDI R0, 0, R6
WB	11-13	LDW R5, 0xFFFC(R2) oczekiwanie na zwolnienie pamieci spowodowane przez STW R6, 0x0000(R3)

2.2 Implementacja

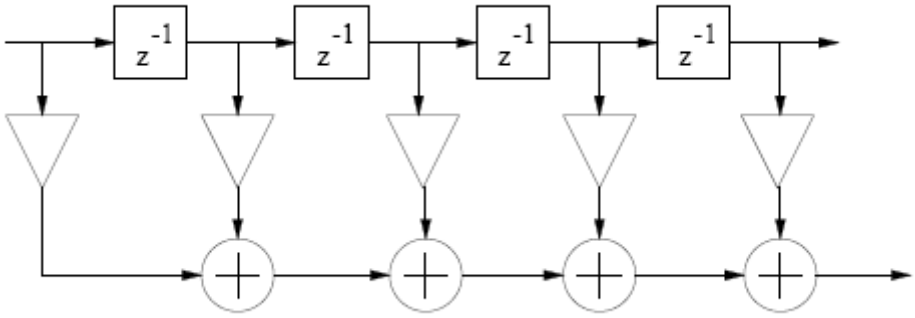
Zaimplementuj kod z poprzedniego ćwiczenia lub po prostu wykorzystaj ten sam kod. Wiadomo jaki jest rezultat wykonania poprzedniego zadania. Dla przypomnienia:

2.3 Struktura filtru SOI

Dla każdej próbki wejściowej x(n) (zapisanej w pamięci), obliczany jest splot. W rezultacie otrzymujemy n próbek na wyjściu:

$$y(n) = \sum_{k=0}^{L-1} h(k) \times (n-k)$$

2.4 Implementacja



Rysunek : Linia opóźniająca filtru SOI

W naszym przypadku N (= 32) próbki wejściowe są przechowywane w pamięci danych (= 0x200 - 0x27F). Kopiowane są one do bufora cyklicznego reprezentującego linię opóźniającą (linia opóźniająca Rys. 1) (L=16, 0x280 - 0x2BF). Splot jest obliczany za pomocą współczynników przechowywanych w pamięci pod adresami 0x2c0 - 0x2FF. 0x2c0 - 0x2FF. Skumulowane wyniki są przechowywane w tablicy wyników (0x300 - 0x37F). Pamięć jest adresowana bajtowo a wszystkie próbki posiadają 32-bity.

Tabela 1

Rejestry	Wykorzystanie
R1	licznik próbek
R2	licznik pętli splotu
R3	akumulator
R4	adres aktualnej próbki do zapisy
R5	adres aktualnej próbki do odczytu
R6-R11	rejestry tymczasowe

Tabela 2. Symbole wykorzystane w pseudokodzie

Nazwa	wartość	Znaczenie
N	32	Liczba próbek wejściowych
L	16	Długość linii opóźniającej, rząd filtru

INPUT	0x200	Adres bazowy próbek wejściowych
DELAY	0x280	Ciąg próbek opóźnionych. Adres bazowy elementów linii opóźniającej
COEFF	0x2C0	Adres bazowy współczynników filtru
OUTPUT	0x300	Adres bazowy pamięci próbek na wyjściu

2.5 Pseudokod

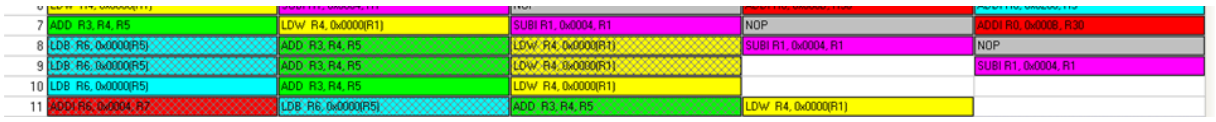
Zobacz rejestry z Tab. 1 oraz zobacz tablicę symboli: Tab. 2

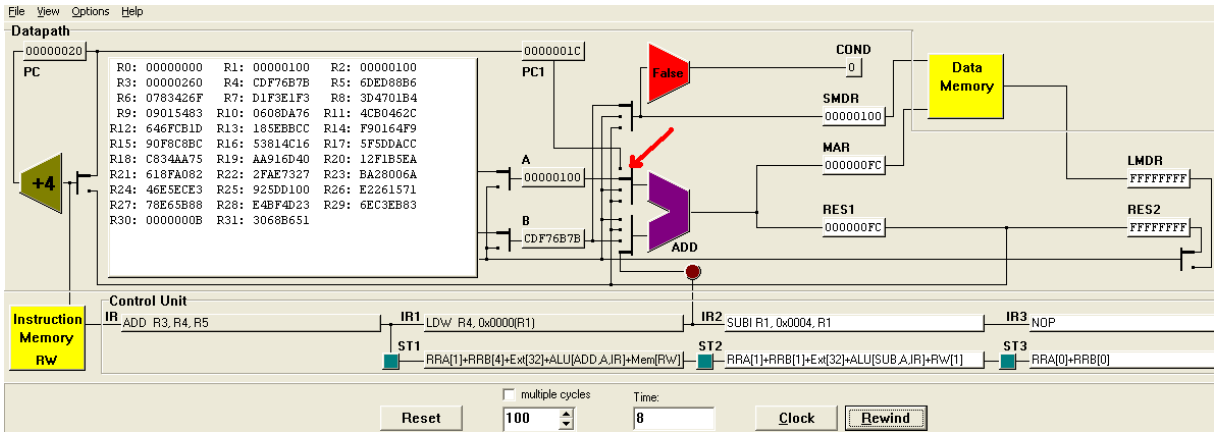
```
R4 = 0
        FOR R1 = 0 TO 4*(N-1) krok o 4
        R7 = MEM[INPUT + R1]
        MEM[DELAY + R4] = R7
        R3 = 0
        R5 = R4
        FOR R2 = 0 TO 4*(L-1) krok o 4
                R9 = MEM[DELAY + R5]
                R10 = MEM[COEFF + R2]
                R3 = R3 + R9*R10
                R5 = R5 + 4
                IF ( R5 > 4*(L-1) )
                THEN R5 = 0
        END FOR
        MEM[OUTPUT + R1] = R3
        R4 = R4 - 4
        IF ( R4 < 0 )
        THEN R4 = 4*(L-1)
END FOR
```

Ustaw czas dostępu do pamięci na 2 lub wiecej cykli zegarowych (jeśli był ustawiony na 1) a następnie zainicjalizuj architekturę (Reset). Przeanalizuj program uruchamiając *Clock*. Zwróć uwagę na zachowanie instrukcji LD, blokującej wykonanie potoku. Zwróć również uwagę na to, że inaczej wygląda sytuacja w przypadku wykonania instrukcji ST. Podaj wyjaśnienie?

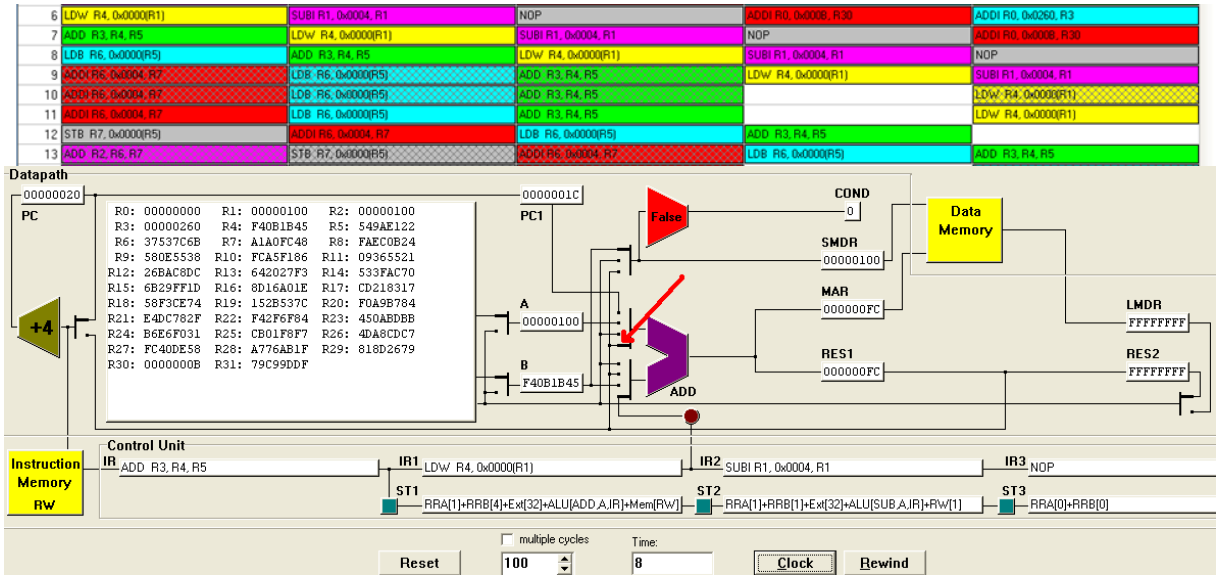
2.3 Optymalizacja

Przeanalizuj zdarzenia występujące podczas wykonania programu. Co dzieje się w przypadku uaktywnienia opcji *forwarding* - dodatkowej opcji uruchamianej przez zestaw multiplekserów i eliminującej hazard wynikający z przejścia instrukcji do fazy WB, jaka jest różnica i z czego wynika.

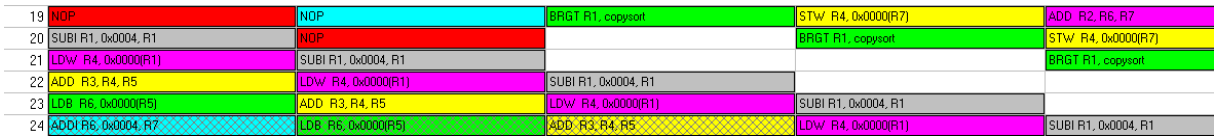


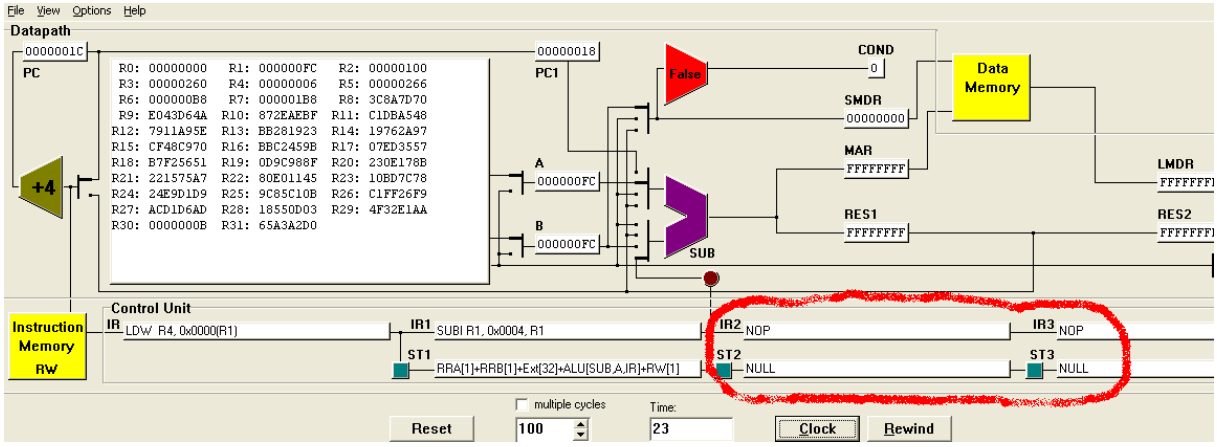


Rys.3 Bez opcji *Forwarding* i *double slot delay*

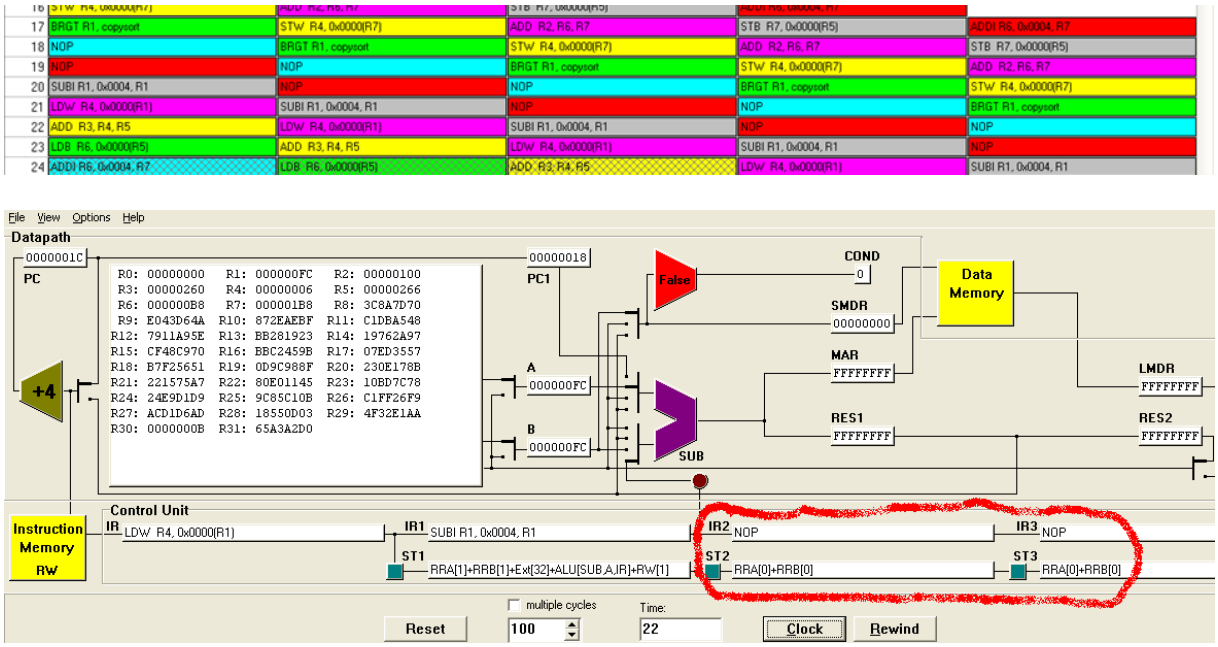


Rys.4 Z *forwarding* bez *double slot delay*





Rys. 5 Bez *double slot delay*



Rys. 6 Z *double slot delay*

Odpowiedź powinna przyjąć formę Tabeli 1.

Porównaj rezultat swojej implementacji z rezultatem **pipe2.ppr** (dostępny u prowadzącego, po zajęciach) implementującym w pewnej wersji powyższy program. Następnie wykonaj kolejne zadania przy założeniu ustawienia czasu dostępu do pamięci na 2 cykle zegarowe. Opcja *double delay* ustawiona powinna być na *enabled*.

1. Zmodyfikuj kod oryginalny wykorzystując opcję *double delay slot*. Opcja *Forwarding* powinna być ustawiona na *enabled*. Zachowaj swój program w katalogu roboczym.

2. Dodaj minimalna liczbę instrukcji NOP – w celu uniknięcia hazardu, do kodu oryginalnego, żeby uniknąć zastoju przy wyłączonej opcji *Forwarding* . Zwróć uwagę na to, że zakleszczenia pojawią się przy włączonej opcji *Forwarding* Mechanizm predykcji powinien być wyłączony. Możesz dokonać zmiany nazw rejestrów i zmienić kolejność wykonania instrukcji, jeśli to poprawi szybkość wykonania instrukcji! Zachowaj wykonany program. Porównaj ilości cykli dla wykonanych programów.

3. Kod z poprzedniego punktu zawiera wiele operacji NOP. Wyeliminujemy je wykorzystując statyczną optymalizację, w przeciwieństwie do optymalizacji dynamicznej. Opcja *Forwarding* jest wyłączona.

- Rozwijanie pętli: Rozpocznij modyfikacje kodu oryginalnego lub kodu z punktu 2. Wykorzystaj możliwie największą liczbę rejestrów. Stwórz możliwie najszybciej działający kod, zwróć uwagę ze instrukcje LDB-ADDI-STB mogą być przeplatane z różnych potoków. Zachowaj kod z tego punktu.
- Programowa implementacja potoku

Która z metod optymalizacji jest lepsza.

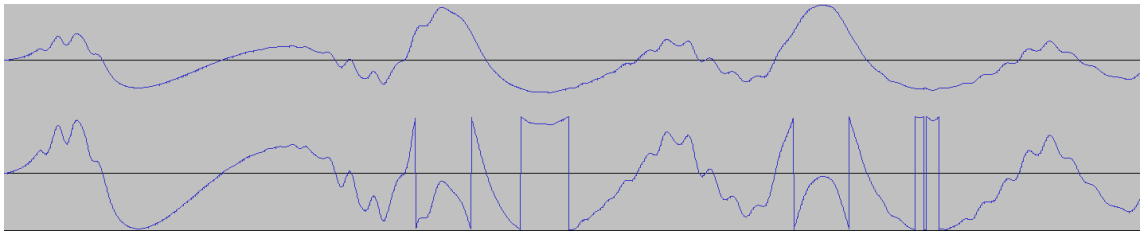
4. Porównaj szybkość wykonania kodu z poprzedniego ćwiczenia z kodem bieżącym, co się zmieniło? Ilukrotnie wzrosła, dlaczego?

2.3 Zaawansowane

Przy aktywnej opcji *delayed branching* część instrukcji wykonywana jest po wykonaniu skoku. Wykonaj powyższy kod z wykorzystaniem mechanizmu predykcji.

2.4 Zaawansowane - wykorzystanie procesora do realizacji zadania filtracji

Do zadania dolaczony zostal plik wav. Należy przygotowac odpowiednio plik wejściowy dla symulatora *Escape-wav (oryginalny Escape posiada ograniczenie pamieci do32kB)*. Próbki wejściowe należy zapisac pod adresem 300h.



Rysunek 1. Graficzna reprezentacja próbek wyjściowych programu

Na powyższym zrzucie z programu *AudaCity* możemy porównać graficzną reprezentacje małego fragmentu takiej próbki, przed filtracją i po filtracji.