



Politechnika Warszawska

---

**Architektura komputerów**  
**Materiały pomocnicze do laboratorium**

**Zbigniew Szymański**

marzec 2015

**Instytut Informatyki**

---

Nowowiejska 15 / 19, 00-665 Warszawa

**Architektura komputerów**  
**Materiały pomocnicze do laboratorium**  
 Zbigniew Szymański <z.szymanski@ii.pw.edu.pl>

**SPIS TREŚCI**

<b>1</b>	<b>ŚRODOWISKO SYMULATORA MARS .....</b>	<b>3</b>
1.1	ŚLEDZENIE WYKONANIA PROGRAMU .....	4
<b>2</b>	<b>MAŁE PROGRAMY DLA PROCESORA MIPS.....</b>	<b>7</b>
<b>3</b>	<b>PROJEKTY DLA PROCESORA MIPS.....</b>	<b>9</b>
3.1	KOMPRESJA DANYCH .....	10
3.2	CO WIDZI ROBOT .....	12
3.3	Z-BUFOR.....	14
3.4	EKG.....	16
3.5	MINI ENIGMA .....	18
3.6	MAPA WYSOKOŚCI .....	21
3.7	GRAFIKA ŻÓŁWOWA.....	23
3.8	ODLEGŁOŚĆ HAMMINGA .....	24
3.9	KOD KRESKOWY CODE 128 .....	25
3.10	KOD KRESKOWY CODE 39 .....	28
3.11	UWAGI DOTYCZĄCE PROJEKTÓW DLA PROCESORA MIPS .....	30
<b>4</b>	<b>MAŁE PROGRAMY DLA PROCESORA INTEL .....</b>	<b>31</b>
<b>5</b>	<b>PROJEKTY DLA PROCESORA INTEL .....</b>	<b>32</b>
5.1	KOMPRESJA DANYCH .....	33
5.2	CO WIDZI ROBOT .....	34
5.3	Z-BUFOR.....	37
5.4	EKG.....	39
5.5	MINI ENIGMA .....	41
5.6	MAPA WYSOKOŚCI .....	43
5.7	GRAFIKA ŻÓŁWOWA.....	46
5.8	ODLEGŁOŚĆ HAMMINGA .....	48
5.9	KOD KRESKOWY CODE 128 .....	50
5.10	KOD KRESKOWY CODE 39 .....	51
<b>6</b>	<b>LITERATURA .....</b>	<b>52</b>

# 1 Środowisko symulatora MARS

Program Mars jest zintegrowanym środowiskiem programistycznym umożliwiającym tworzenie programów w asemblerze procesora MIPS. Program jest napisany w języku Java, zatem może być uruchamiany zarówno w systemie Linux jak i Windows. Można go pobrać ze strony WWW:

<http://courses.missouristate.edu/kenvollmar/mars/>

Użycie programu Mars zostanie zaprezentowane na przykładzie programu przedstawionego na listingu 1.1. Jest dostępny do pobrania pod adresem:

<http://galera.ii.pw.edu.pl/~zsz/arko/mips.asm>

**Listing 1.1.** Kod przykładowego programu w asemblerze procesora MIPS.

```
#-----
#autor: Zbigniew Szymanski
#data : 2002.11.01
#opis : Program wczytuje ciag znakowy z klawiatury i wyswietla go
#-----
        .data
input:   .space 80
prompt:  .asciiz "\nPodaj ciag znakow> "
msg1:    .asciiz "\nWczytany ciag    > "

        .text
main:
#wyswietlenie zapytania
        li $v0, 4          #system call for print_string
        la $a0, prompt     #address of string
        syscall

#wczytanie ciagu znakowego
        li $v0, 8          #system call for read_string
        la $a0, input      #address of buffer
        syscall

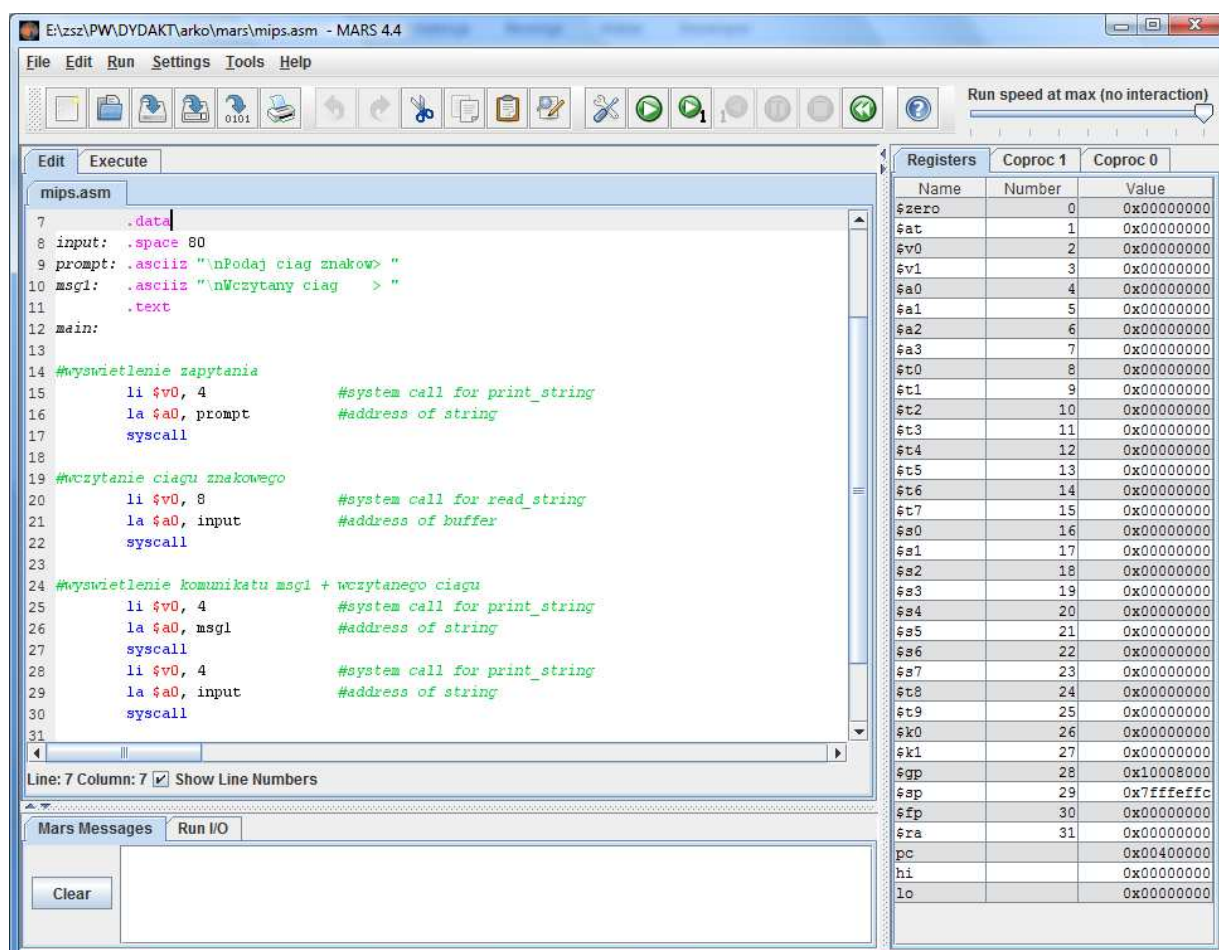
#wyswietlenie komunikatu msg1 + wczytanego ciagu
        li $v0, 4          #system call for print_string
        la $a0, msg1       #address of string
        syscall
        li $v0, 4          #system call for print_string
        la $a0, input      #address of string
        syscall

exit:    li $v0,10          #Terminate
        syscall
```

Celem działania programu ma być wyświetlenie komunikatu **Podaj ciąg znaków>** , wczytanie z klawiatury ciągu znaków, wyświetlenie komunikatu **Wczytany ciąg >** oraz wyświetlenie wczytanego ciągu znaków. W programie został popełniony intencjonalnie jeden błąd aby pokazać proces debugowania kodu.

## 1.1 Śledzenie wykonania programu

Na rys. 1.1 przedstawiono okno programu Mars po wczytaniu przykładowego programu (menu File | Open). Centralną część okna zajmuje kod programu assemblerowego (zakładka Edit). W dolnej części okna programu, w zakładce Mars Messages, pojawiają się komunikaty programu Mars np. dotyczące powodzenia lub niepowodzenia asemblacji kodu. W prawej części okna znajduje się podgląd wartości rejestrów procesora wykorzystywany przy debugowaniu programu.

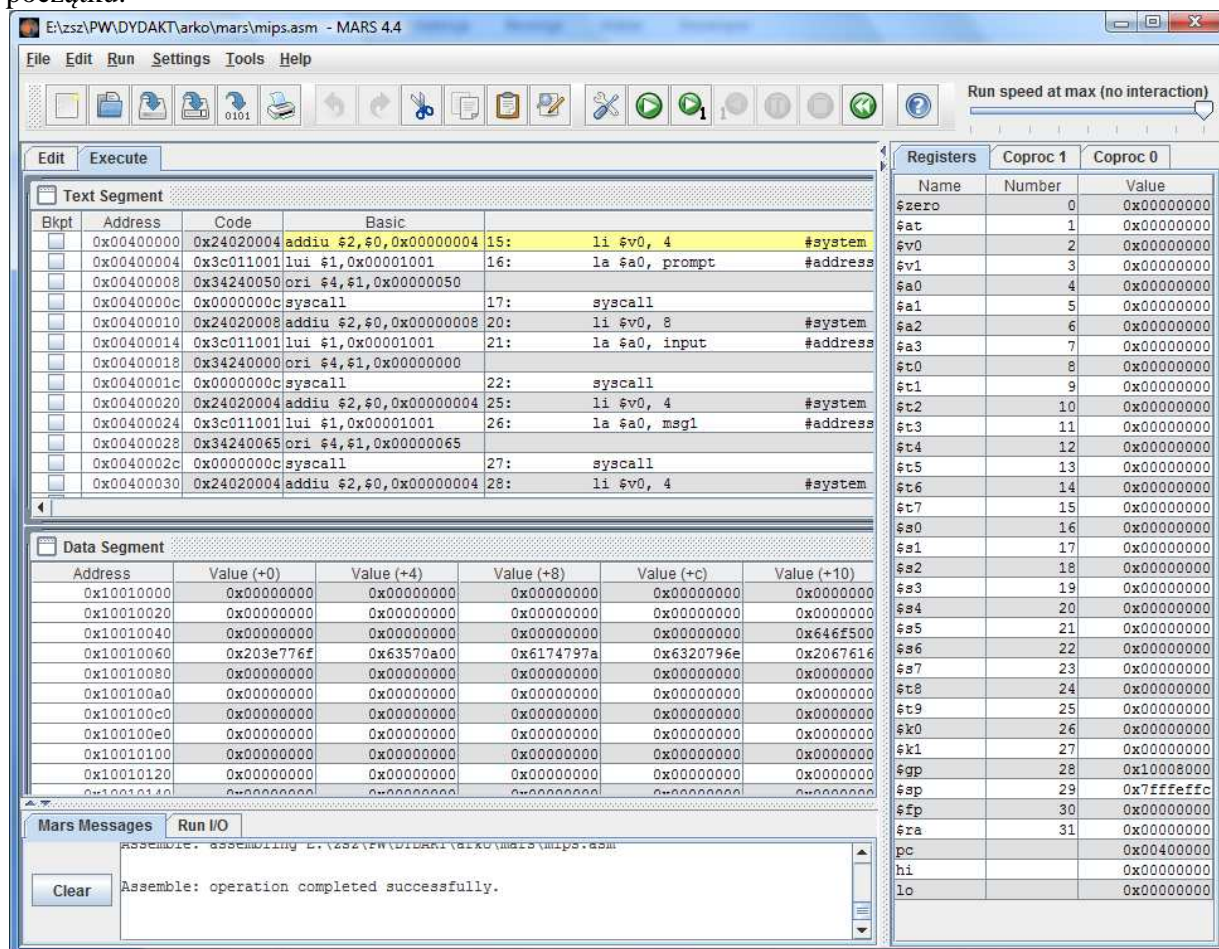


Rys. 1.1 Środowisko Mars – widok edycji kodu assemblerowego

Przed uruchomieniem programu należy dokonać asemblacji kodu (Run | Assemble, lub klawisz F3). Powodzenie operacji sygnalizuje komunikat „Assemble: operation completed successfully.”

w zakładce Mars Messages (rys. 1.2). Uruchomienia programu można dokonać wybierając z menu polecenie Run | Go lub wciskając klawisz F5. W zakładce Run I/O pojawi się komunikat „Podaj ciąg znaków >”. Program umożliwi wprowadzenie tylko jednego znaku i przerwie swoje działanie, co nie jest zgodne z założeniami.

Powtórne wykonanie programu wymaga wybrania polecenia Run | Reset (klawisz F12). Proszę zauważyć (patrz zakładka Registers), że po wybraniu tego polecenia rejestr PC zmienia swoją wartość z 0x00400048 na 0x00400000 co oznacza, że program będzie wykonywany od początku.



Rys. 1.2 Środowisko Mars – praca w trybie wykonania programu

Aby móc prześledzić działanie wykonania programu tuż przed wywołaniem funkcji systemowej wczytującej ciąg znaków z klawiatury zostanie ustawiona pułapka (breakpoint) pod adresem 0x00400010. W tym celu należy zaznaczyć pole wyboru w zakładce Execute w kolumnie Bkpt w linii odpowiadającej wymienionemu adresowi i uruchomić program (Run | Go). Wykonanie programu zostanie zatrzymane przed wykonaniem instrukcji umieszczonej pod adresem 0x00400010.

Po wybraniu polecenia Run | Step (klawisz F7) zostanie wykonana w trybie krokowym tylko jedna instrukcja. Proszę zauważyć, że w zakładce Registers zmianie uległa zawartość rejestru \$v0 i wynosi 8<sub>dec</sub>. Krokowe wykonanie kolejnych dwóch instrukcji spowoduje załadowanie do rejestru \$a0 adresu bufora, w którym mają być umieszczone wczytane dane. Program jest teraz zatrzymany na instrukcji syscall pod adresem 0x0040001c.

Wybierając polecenie Help | Help (klawisz F1) można sprawdzić (wybierając zakładkę Syscalls w dolnej części okna pomocy) dokumentację funkcji read\_string w tabeli „Table of Available Services”. W kolumnie „arguments” wymieniono rejestry, których wartość musi być ustawiona do poprawnego działania funkcji. Są to rejestry \$a0 zawierający adres bufora danych i \$a1 zawierający rozmiar tego bufora. Okazuje się, że w analizowanym programie nie ustawiono wartości rejestru \$a1. Omawiany program należy zmodyfikować w zakładce Edit tak jak pokazano to na listingu 1.2 (dodany fragment zaznaczono kolorem szarym).

**Listing 1.2.** Zmodyfikowany fragment przykładowego programu.

```
#-----  
...  
#wczytanie ciagu znakowego  
    li $v0, 8      #system call for read_string  
    la $a0, input  #address of buffer  
    li $a0, 80  
    syscall  
...
```

Następnie należy dokonać ponownej asemblacji i uruchomienia. Zmodyfikowany program poprawnie wczytuje ciąg znaków i następnie go wyświetla.

Przy uruchamianiu programów operujących na danych tekstowych przydatne może być zapisanie zawartości segmentu danych do pliku tekstowego. Wybranie polecenia File | Dump Memory (skrót CTRL d) powoduje otwarcie okna zatytułowanego „Dump Memory To File”. Z listy Memory Segment należy wybrać nazwę segmentu pamięci, którego zawartość ma być zapisana w pliku, zaś z listy Dump Format sposób zapisu. W przypadku wyboru „Ascii text” każdy wiersz pliku będzie zawierał zawartość jednego 32 bitowego słowa pamięci.

## 2 Małe programy dla procesora MIPS

Napisać program w asemblerze procesora MIPS. Program powinien wczytać z klawiatury ciąg znaków, podać konwersji ciąg znaków, wyświetlić wynik przetwarzania na ekranie.

Można skorzystać z szablonu programu umieszczonego pod adresem [galera.ii.pw.edu.pl/~zsz/arko/mips.asm](http://galera.ii.pw.edu.pl/~zsz/arko/mips.asm)

1a	Wszystkie małe litery w ciągu powinny zostać zamienione na *. Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> A** M* K***
1b	Wszystkie wielkie litery w ciągu powinny zostać zamienione na *. Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> *la *a *ota
1c	Wszystkie cyfry w ciągu powinny zostać zamienione na *. Podaj ciąg znakow> tel. 12-34-55 Ciąg po konwersji> tel. **-**-**
1d	Wszystkie znaki nie będące literami powinny zostać zamienione na *. Podaj ciąg znakow> Ala*ma*1*kota, 2 psy. Ciąg po konwersji> Ala*ma***kota****psy*
2a	Zamienić pozycję znaków w kolejnych parach. Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> lA aaMK toa
2b	Odwrócić kolejność znaków w ciągu. Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> atoK aM ala
2c	Na początku ciągu umieścić znaki z pozycji nieparzystych, następnie z parzystych. Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> AaM oal aKt
3a	Zamienić litery występujące po znaku - na wielkie (znak - jest usuwany z ciągu wyjściowego). Podaj ciąg znakow> A-l-a M-a Kota Ciąg po konwersji> ALA MA Kota
3b	Zamienić wszystkie litery na wielkie z wyjątkiem występujących po znaku - (znak - jest usuwany z ciągu wyjściowego). Podaj ciąg znakow> Ala M-a Kot-a Ciąg po konwersji> ALA Ma KOTa
4a	Policzyć sumę wszystkich cyfr występujących w ciągu znakowym. Podaj ciąg znakow> tel. 12-00-55 Wynik > 13
4b	Podać liczbę liter występujących w ciągu znakowym. Podaj ciąg znakow> tel. 12-00-55 Wynik > 3

4c	<p>Podać liczbę cyfr występujących w ciągu znakowym.</p> <p>Podaj ciąg znakow&gt; tel. 12-00-55</p> <p>Wynik &gt; 6</p>
5a	<p>Zamienić wszystkie znaki występujące od pierwszego w ciągu znaku &gt; do pierwszego następującego po nim znaku &lt; na *.</p> <p>Podaj ciąg znakow&gt; Ala &gt;ma&lt; kota.</p> <p>Ciąg po konwersji&gt; Ala &gt;***&lt; kota.</p>
5b	<p>Zamienić wszystkie znaki występujące przed pierwszym znakiem &gt; i po pierwszym następującym po nim znaku &lt; na *.</p> <p>Podaj ciąg znakow&gt; Ala &gt;ma&lt; kota.</p> <p>Ciąg po konwersji&gt; ****&gt;ma&lt;*****</p>
5c	<p>Przenieść wszystkie znaki występujące przed pierwszym znakiem ! na koniec ciągu</p> <p>Podaj ciąg znakow&gt; Ala !ma kota</p> <p>Ciąg po konwersji&gt; !ma kota Ala</p>
6a	<p>Policzyć liczbę grup cyfr występujących w ciągu znakowym.</p> <p>Podaj ciąg znakow&gt; tel. 12-00-55</p> <p>Wynik &gt; 3</p>
6b	<p>Policzyć liczbę grup liter (rozdzielonych nie literami) występujących w ciągu znakowym.</p> <p>Podaj ciąg znakow&gt; Ala ma 1 kota.</p> <p>Wynik &gt; 3</p>



## 3 Projekty dla procesora MIPS

### **Wymagania dotyczące dokumentacji**

Wraz z działającym programem należy przedstawić dokumentację projektu. Powinna zawierać m.in.:

- opis struktury programu,
- opis struktur danych i ich implementacji,
- opis implementacji algorytmu,
- opis formatu stosowanych plików (jeśli format nie wynika wprost z treści zadania),
- dobrze zdokumentowane wyniki testowania dowodzące, że program działa poprawnie. Powinny zawierać opis procedury testowania - z dokumentacji powinno jasno wynikać co było testowane, na jakich danych przeprowadzano testy, jakie były wyniki testowania, czy test był poprawny/niepoprawny (forma tabelaryczna będzie mile widziana) i ew. stopień kompresji. Na podstawie dostarczonych plików i opisu testowania osoba sprawdzająca program powinna być w stanie powtórzyć testy.

## 3.1 Kompresja danych

Zrealizować programy do kompresji (o nazwie *compress*) oraz dekompresji pliku (o nazwie *decompress*) w oparciu o kodowanie Huffmana w assemblerze procesora MIPS. Kodowanie Huffmana polega na utworzeniu słów kodowych (ciągów bitowych), których długość jest odwrotnie proporcjonalna do prawdopodobieństwa ich występowania. Im częściej dany symbol występuje w ciągu danych, tym mniej zajmuje bitów. Kod Huffmana jest kodem prefiksowym - żadne słowo kodowe nie jest początkiem innego słowa.

### Opis algorytmu kompresji

Algorytm kompresji składa się z następujących etapów:

1. przygotowanie tablicy częstości (lub prawdopodobieństwa) występowania poszczególnych symboli,
2. przygotowanie drzewa binarnego służącego do określenia kodów symboli,
3. określenie kodowania poszczególnych symboli na podstawie drzewa utworzonego w poprzednim kroku,
4. kodowanie danych – zastąpienie symboli ich kodami.

Etap 2 i 3 algorytmu kompresji zostanie omówiony na przykładzie przedstawionym na rysunku 1. W każdym kroku budowania drzewa łączone są w jeden węzeł dwa elementy o najmniejszym prawdopodobieństwie. Lewa krawędź stworzonego węzła prowadzi do elementu o wyższym prawdopodobieństwie, a prawa o niższym.

W pierwszym kroku algorytmu dane są symbole wraz z prawdopodobieństwami ich występowania. Wybierane są dwa o najmniejszym prawdopodobieństwie ( $j = 0.002$ ,  $b = 0.031$ ) i łączone w jeden węzeł, któremu przypisane jest sumaryczne prawdopodobieństwo symboli  $j$ ,  $b$  ( $0.033$ ). Lewa krawędź od nowo utworzonego węzła prowadzi do symbolu  $b$  o wyższym prawdopodobieństwie (reprezentuje etykietę 0), a prawa do symbolu  $j$  o mniejszym prawdopodobieństwie (reprezentuje etykietę 1).

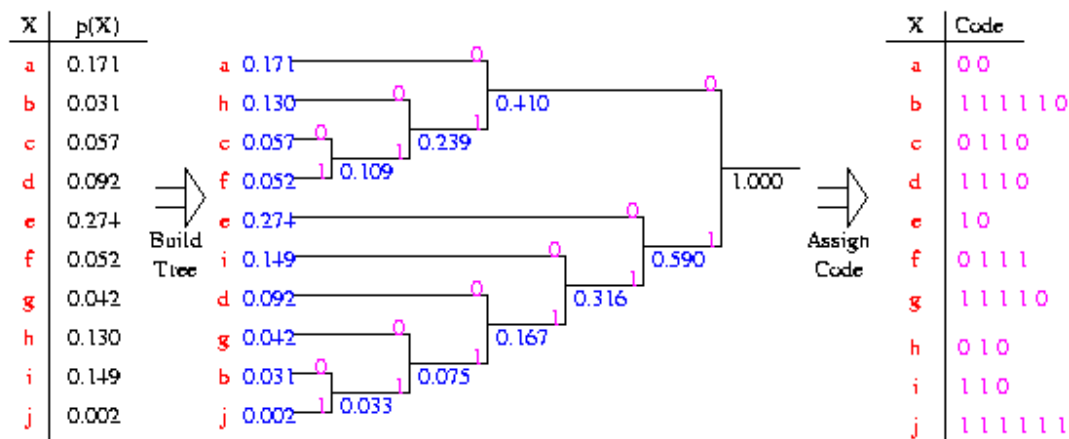
W kroku drugim do połączenia wybrany jest węzeł utworzony w kroku 1 ( $0.033$ ) i symbol  $g$  ( $0.042$ ). Utworzonemu węzłowi odpowiada prawdopodobieństwo  $0.075$ . Lewa krawędź od nowo utworzonego węzła do symbolu o wyższym prawdopodobieństwie reprezentuje etykietę 0 ( $g$ ), a do węzła o mniejszym prawdopodobieństwie reprezentuje etykietę 1.

W kroku trzecim łączone są kolejne symbole o najmniejszym prawdopodobieństwie -  $f$  ( $0.052$ ) oraz  $c$  ( $0.057$ ). Algorytm jest kontynuowany, aż zostanie zbudowane pełne drzewo posiadające jeden korzeń.

Kody poszczególnych symboli określają kolejne etykiety krawędzi drzewa na ścieżce od korzenia do wskazanego symbolu. Np. kod symbolu  $d$  to 1110.

### Opis algorytmu dekompresji

Dekompresja wymaga posiadania drzewa kodów, a odtworzenie danych polega na czytaniu kolejnych bitów ze strumienia danych i jednoczesnym przechodzeniu drzewa zgodnie z przyjętą konwencją. Po dojściu do liścia i wypisaniu związanego z nim symbolu powracamy do korzenia.



Rysunek 1. Tablica prawdopodobieństwa występowania symboli, drzewo służące do określenia kodów, tablica z kodami poszczególnych symboli [3].

### Wejście / Wyjście

Wejściem do programu kompresującego *compress* jest plik o nazwie *dane* o dowolnej wielkości (nie można zatem wczytać całego pliku na jeden raz do pamięci). W wyniku kompresji powstaje plik o nazwie *archiwum* zawierający zakodowane dane oraz wszystkie dodatkowe informacje potrzebne do poprawnej dekompresji.

Wejściem do programu dekompresującego *decompress* jest plik o nazwie *archiwum*. W wyniku dekompresji powstaje plik o nazwie *dane1*.

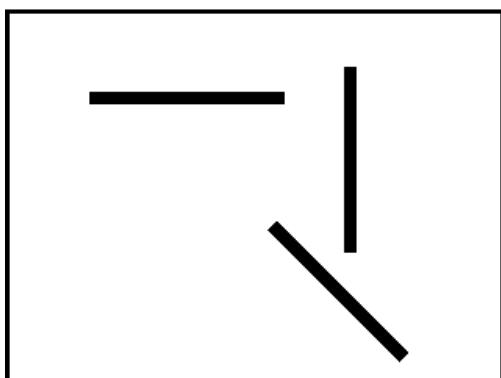
## 3.2 Co widzi robot

Zrealizować program symulujący laserowy skaner robota służący do pomiaru odległości. Otoczenie robota (przeszkody) jest opisane w pliku w formacie BMP [5], a parametry aktualnego położenia w pliku tekstowym. Należy wykreślić ciągłą czerwoną linią przebieg wiązek lasera oraz zapisać w pliku tekstowym ich długości.

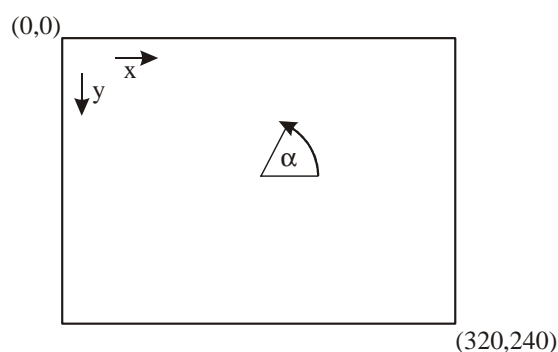
### Wejście

Dane wejściowe dla programu znajdują się w plikach o nazwie **otoczenie.bmp** oraz **parametry.txt**.

Plik **otoczenie.bmp** zawiera informacje o rozmieszczeniu przeszkód i ma rozmiar 320 na 240 pikseli. Piksel o kolorze białym oznacza wolną przestrzeń, zaś o kolorze czarnym - przeszkodę.



Rys. 1. Przykładowa zawartość pliku *otoczenie.bmp*.



Rys. 2. Zastosowany układ współrzędnych.

Plik **parametry.txt** zawiera informację o położeniu robota (kolejne liczby są rozdzielone spacjami i umieszczone są w jednym wierszu):

- $x, y$  - współrzędne położenia robota – dwie liczby całkowite z zakresu  $\langle 0, 320 \rangle$  i  $\langle 0, 240 \rangle$ ,
- $\alpha$  - kierunek, w którym skierowany jest robot (wyrażony w stopniach) – liczba całkowita z zakresu  $\langle 0, 360 \rangle$

### Przykładowa zawartość pliku parametry.txt

```
100 100 45
```

Parametry stałe:

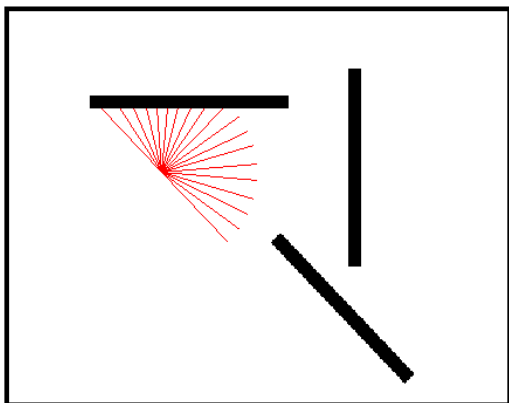
- maksymalna długość wiązki laserowej – 60 pikseli,
- liczba wiązek laserowych – 19. Wiązki są rozmieszczone równomiernie co  $10^\circ$  w zakresie  $\langle 90^\circ, -90^\circ \rangle$  od kierunku, w którym skierowany jest robot.

## Wyjście

Wyniki działania programu powinny być zapisane w dwóch plikach o nazwach **symulacja.bmp** i **wyniki.txt**.

Plik **symulacja.bmp** (rys. 3) zawiera obraz otoczenia z pliku wejściowego **otoczenie.bmp** oraz naniesione czerwonymi ciągłymi liniami przebiegi wiązek laserowych. Obraz ma stałe rozmiary 320x240 pikseli.

Plik **wyniki.txt** zawiera w kolejnych wierszach długości wiązek laserowych (odległości od przeszkody). W przypadku, gdy wiązka nie natrafi na przeszkodę do pliku powinna być wstawiona wartość 255. Skanowanie rozpoczyna się od kąta  $90^\circ$  (lewa strona robota).



**Rys.3.** Przykładowy obraz z pliku **symulacja.bmp**

Przykładowa wyniki.txt	zawartość	pliku
55		
47		
43		
40		
39		
39		
40		
43		
47		
55		
255		
255		
255		
255		
255		
255		
255		
255		
255		

## Uwagi

1. Do kreślenia odcinków można wykorzystać algorytm Bresenhama [6].

## Warianty zadania:

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora MIPS.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone). Należy oszacować jak duże będą błędy w porównaniu z programem, który operowałby na arytmetyce zmiennoprzecinkowej.

### 3.3 Z-bufor

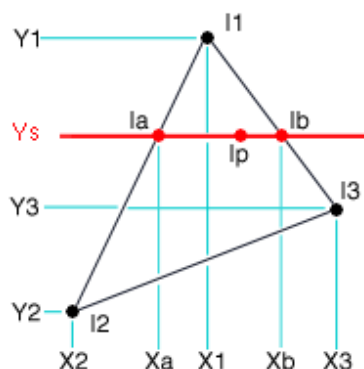
Należy napisać program symulujący działanie mechanizmu Z-bufora [7]. Program ma za zadanie rysować trójkąty, których parametry wczytane są z pliku, zaś efekty działania programu powinny być zapisane do plików BMP. Przy określaniu kolorów pikseli należy użyć metody cieniowania interpolowanego (takiego jak w metodzie cieniowania Gouraud).

#### Z-bufor

Z-bufor [7] jest mechanizmem pozwalającym na rozwiązanie problemu przesłaniania rysowanych obiektów w grafice trójwymiarowej. Jest to bufor o takich samych rozmiarach jak bufor obrazu przechowujący współrzędną Z każdego piksela obrazu. Na początku wartości w Z-buforze inicjowane są wartością odpowiadającą nieskończoności (0xFFFF FFFF). Przed narysowaniem piksela w buforze obrazu sprawdzane jest, czy jego składowa Z jest mniejsza niż wartość w Z-buforze. Jeśli tak (oznacza to, że piksel jest bliżej obserwatora niż poprzednio narysowany) aktualizowana jest wartość piksela w buforze obrazu i w Z-buforze.

#### Cieniowanie interpolowane

Cieniowanie interpolacyjne polega na przypisaniu pikselom cieniowanego wielokąta koloru obliczonego poprzez interpolację wartości kolorów z poszczególnych wierzchołków. Na rys.1 pokazano ideę obliczania koloru dla punktu o współrzędnych ( $X_p$ ,  $Y_s$ ). Najpierw dokonywana jest interpolacja koloru na krawędziach trójkąta (czerwone kropki na rys.1), a następnie pomiędzy tymi punktami w jego wnętrzu.  $I_a$  oznacza wartość składowej koloru w punkcie ( $X_a$ ,  $Y_s$ ),  $I_b$  oznacza wartość składowej koloru w punkcie ( $X_b$ ,  $Y_s$ ). Obliczenia trzeba wykonać trzy razy: dla składowej czerwonej, zielonej i niebieskiej.



Rys.1 Idea cieniowania wielokąta metodą interpolacyjną [8].

Wartości składowych koloru można obliczyć korzystając ze wzorów:

$$I_a = (Y_s - Y_2) / (Y_1 - Y_2) * I_1 + (Y_1 - Y_s) / (Y_1 - Y_2) * I_2$$

$$I_b = (Y_s - Y_3) / (Y_1 - Y_3) * I_1 + (Y_1 - Y_s) / (Y_1 - Y_3) * I_3$$

$$I_p = (X_b - X_p) / (X_b - X_a) * I_a + (X_p - X_a) / (X_b - X_a) * I_b$$

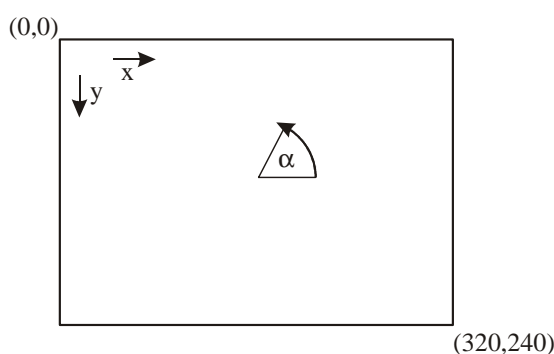
## Wejście

Dane wejściowe dla programu znajdują się w pliku tekstowym o nazwie **opis.txt** o następującym formacie:

- Pierwszy wiersz zawiera wartości trzech składowych RGB koloru tła rozdzielone spacjami. Wartości zawierają się w przedziale  $\langle 0, 255 \rangle$ .
- Kolejne wiersze zawierają opis trójkątów:

$X_1 Y_1 Z_1 R_1 G_1 B_1 X_2 Y_2 Z_2 R_2 G_2 B_2 X_3 Y_3 Z_3 R_3 G_3 B_3$

gdzie:  $X_i, Y_i, Z_i$  są współrzędnymi wierzchołka  $X, Y$  i odpowiadającą mu składową  $Z$ , zaś  $R_i, G_i, B_i$  są składowymi RGB koloru wierzchołka. Wartości współrzędnej  $X$  zawierają się w przedziale  $\langle 0, 319 \rangle$ , współrzędnej  $Y$   $\langle 0, 239 \rangle$ . Składowa  $Z$  może przyjmować wartości z zakresu  $\langle 0x0000\ 0000, 0xFFFF\ FFEE \rangle$ . Wartości RGB zawierają się w przedziale  $\langle 0, 255 \rangle$ .



Rys. 2. Zastosowany układ współrzędnych.

## Wyjście

Wynikiem działania programu powinny być dwa pliki w formacie BMP [5] o nazwach **scena.bmp** i **zbufor.bmp**. Pliki te mają stały rozmiar 320x240 pikseli.

Plik **scena.bmp** zawiera narysowane trójkąty opisane w pliku **opis.txt**.

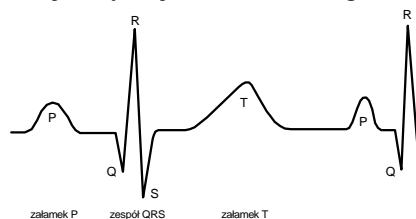
Plik **zbufor.bmp** zawiera wizualizację zawartości Z-bufora w odcieniach szarości. Wartość 0 reprezentowana przez kolor biały, nieskończoność przez kolor czarny.

## Warianty zadania:

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora MIPS.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone). Należy oszacować jak duże będą błędy w porównaniu z programem, który operowałby na arytmetyce zmiennoprzecinkowej.

## 3.4 EKG

Elektrokardiogram [10] (EKG) jest zapisem sumarycznego napięcia (wytworzonego przez włókna mięśniowe serca) na powierzchni ciała względem osi czasu. Poniższy rysunek przedstawia kształt wyidealizowanej krzywej elektrokardiograficznej.



Wychylenia w górę lub w dół nazywa się załamkami i oznacza się dużymi literami P, Q, R, S, T, U. Załamki Q, R, S określa się wspólnym mianem zespołu QRS. Odpowiadają one procesowi depolaryzacji komórek mięśnia sercowego. Podstawowym badaniem wykonywanym przez systemy do automatycznej analizy sygnału EKG jest określenie rytmu serca. Odbywa się to na podstawie rozmieszczenia zespołów QRS.

Proces rejestracji sygnału EKG wygląda następująco. Sygnał EKG z elektrod rozmieszczonych na ciele pacjenta jest wzmacniany przy pomocy przedwzmacniacza (1000x). Następnie jest próbkowany za pomocą 12-bitowego przetwornika analogowo-cyfrowego z częstotliwością 1kHz. Próbkki są zapisywane do plików w postaci całkowitych liczb 16-bitowych ze znakiem.

### Cel programu

Realizowany program powinien dokonać detekcji zespołów QRS w sygnale EKG. Sygnał jest zapisany w pliku. Opis algorytmu detekcji, który należy zaimplementować znajduje się w kolejnym punkcie. Wyniki analizy należy wyświetlić na standardowym wyjściu w zadanym formacie.

### Opis algorytmu detekcji

Algorytm Holsingera [11] działa na ciągu próbek  $X(n)$  sygnału EKG z jednego odprowadzenia. Pozycja zespołu QRS jest określana na podstawie przekroczenia wartości progowej przez pierwszą pochodną sygnału.

Algorytm składa się z 3 kroków:

1. Do bufora wejściowego wczytywany jest ciąg próbek  $X(n)$  sygnału EKG z jednego odprowadzenia.
2. Liczona jest pierwsza pochodna sygnału według wzoru

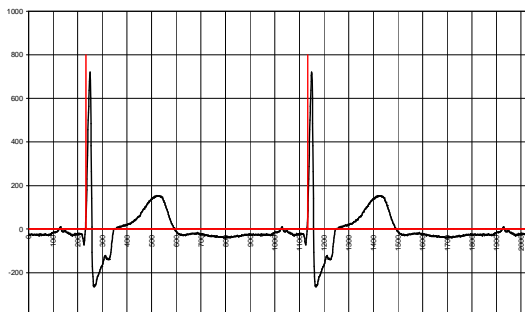
$$Y(n) = X(n+4) - X(n-4).$$

Pochodną oblicza się jako różnicę próbki oddalanej o 1 ms do przodu (4 próbki przy częstotliwości próbkowania 1 kHz) od analizowanej próbki i próbki odległej o 1 ms wstecz.

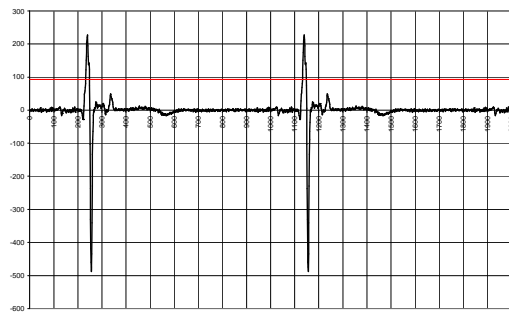
3. W sygnale pochodnej poszukiwany jest punkt przekraczający wartość progową  $P$  (ustaloną doświadczalnie na 93). Jeśli dla kolejnych 12 próbek wartość pochodnej jest



większa od P przynajmniej w 4 przypadkach, to miejsce, w którym próg został przekroczony po raz pierwszy uznaje się za punkt detekcji.



*Wczytany sygnał EKG z zaznaczonymi punktami detekcji (kolor czerwony).*



*Pochodna sygnału EKG oraz próg (kolor czerwony) wykorzystywany przy detekcji zespołów QRS*

## Wejście

Dane wejściowe dla programu znajdują się w pliku o nazwie **ekg.bin** zawierającym ciąg próbek sygnału EKG. Próbką sygnału EKG jest 16-bitową liczbą całkowitą ze znakiem.

Przykładowy plik jest dostępny pod adresem:

**<http://galera.ii.pw.edu.pl/~zsz/arko/projekt-mips>**

## Wyjście

Program powinien wypisywać wyniki na standardowym wyjściu. Pojedynczy wiersz wyniku ma następujący format:

***nr próbki:wartość próbki:wartość pochodnej:znacznik detekcji***

*Numer próbki* jest kolejnym numerem odczytanej z pliku próbki (numeracja od zera). *Wartość próbki* jest wartością odczytaną z pliku. *Wartość pochodnej* jest wartością obliczoną zgodnie ze wzorem na pochodną w algorytmie Holsingera. Jeśli dana próbka została uznana za punkt detekcji to *znacznik detekcji* przyjmuje wartość 800. W przeciwnym przypadku ma wartość 0. Przykładowy fragment danych generowanych przez program:

```
...
230:-9:69:0
231:8:80:0
232:29:93:0
233:56:122:800
234:84:151:0
...
```

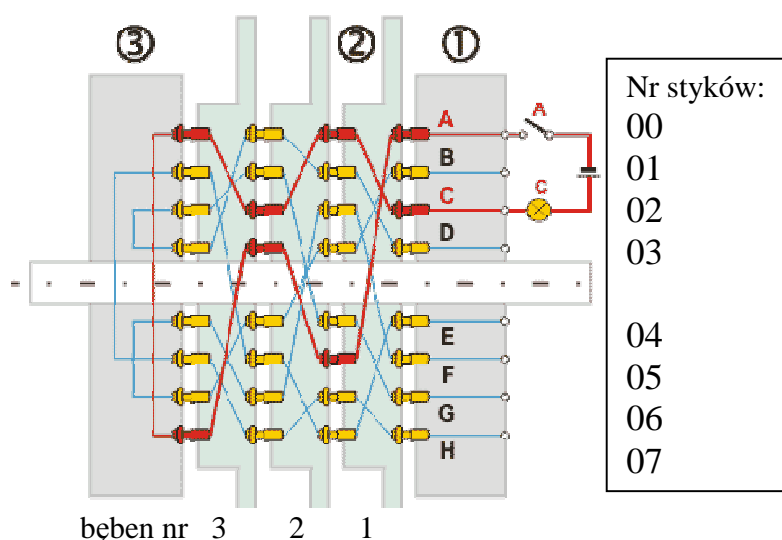
## Uwagi

Pliki wejściowe mogą mieć różną długość. Sygnał EKG należy analizować fragmentami o ustalonej długości.

### 3.5 Mini Enigma

Zrealizować program szyfrujący ciągi znakowe (pobierane z pliku) algorytmem używanym w maszynie Enigma opisanym w następnym punkcie.

#### Opis algorytmu



Rys 1. Zasada działania bębnow szyfrujących maszyny Enigma [12]. Oznaczenia: 1 – styki wejściowe/wyjściowe, 2 – bębny szyfrujące, 3 – bęben odwracający

Każdy ze styków wejściowo/wyjściowych odpowiada jednemu znakowi z pewnego podzbioru znaków ASCII. Zakładamy, że styki są ponumerowane od 00. Jeden bęben szyfrujący wykonuje proste podstawienie dwóch znaków.

Na rys. 1 pokazano zasadę działania algorytmu. Sygnał odpowiadający literze A podawany jest poprzez styk wejściowy (w powyższym przykładzie o numerze 00), następnie propaguje poprzez styk (nr 00) po prawej stronie pierwszego bębna do styku (nr 05) po lewej stronie bębna. W bębnie nr 2 wchodzi na styk nr 05 po prawej stronie i przechodzi na styk nr 03 po lewej stronie. W bębnie nr 3 wchodzi na styk nr 03 po prawej stronie i przechodzi na styk nr 07 po lewej stronie. Następnie w bębnie odwracającym sygnał propaguje ze styku nr 07 na styk o numerze 00, zaś dalej z powrotem przez bębny nr 3, 2 oraz 1 (wchodząc z lewej strony i wychodząc z prawej). Zaszzyfrowany znak otrzymujemy po podaniu sygnału na styk wejściowo/wyjściowy o numerze 02 (odpowiadający w przykładzie literze C).

Po zaszzyfrowaniu znaku bęben nr 1 obraca się o jedną pozycję. Zatem styk o numerze 01 bębna nr 1 będzie teraz sąsiadował ze stykiem wejściowo/wyjściowym nr 00 (a styk 00 bębna nr 1, ze stykiem wej./wyj. nr 07). Gdy bęben nr 1 wykona pełen obrót, następuje obrót bębna nr 2 o jedną pozycję, itd. Bęben odwracający jest nieruchomy.

## Wejście

### Ciągi znakowe do zaszyfrowania:

Ciągi znakowe do zaszyfrowania (tekst jawny) znajdują się w kolejnych wierszach pliku o nazwie **plaintext.txt**. Zakładamy, że:

- długość wiersza tekstu razem ze znakiem końca wiersza (o kodzie  $10_{\text{dec}}$ ) jest nie większa niż 1024 znaki.
- tekst jawny zawiera znaki o kodach ASCII z przedziału  $<32_{\text{dec}}, 95_{\text{dec}}>$
- wszystkie znaki, których kody ASCII nie mieszczą się w przedziale wymienionym w poprzednim punkcie są ignorowane
- plik będzie wczytywany wiersz po wierszu (a nie na jeden raz w całości)
- po wczytaniu kolejnego wiersza algorytm szyfrujący kontynuuje pracę (nie powraca do stanu początkowego)
- dane kończą się pustym wierszem zawierającym tylko znak o kodzie  $10_{\text{dec}}$

### Połączenia bębnow szyfrujących:

Plik o nazwie **rotors.txt** zawiera opis połączeń bębnow szyfrujących. Plik składa się z czterech sekcji – trzy pierwsze opisują połączenia bębnow standardowych (w stałej kolejności: bęben nr 1, 2, 3), ostatnia bębna odwracającego. Każda sekcja składa się z etykiety (dowolny ciąg znakowy) oraz 64 linii opisujących połączenia styków po prawej i po lewej stronie bębna (dla bębnow standardowych) i 32 linie (dla bębna odwracającego).

---

#### Przykład sekcji opisującej połączenia bębna nr 1

---

```
Rotor 1: nr styku po lewej stronie- nr styku po prawej stronie
00-32
01-34
02-36
...
62-03
63-01
```

---

Powyższe dane oznaczają, że np. lewy styk 00 jest połączony z prawym stykiem 32.

### Początkowe położenie bębnow szyfrujących:

Początkowe położenie bębnow szyfrujących podane jest w pliku **init.txt**. W kolejnych wierszach podane jest początkowe położenie bębnow nr 1, 2, 3.

---

#### Przykładowa zawartość pliku init.txt

---

```
00
17
28
```

---

Dane z powyższego przykładu oznaczają, że:

- styk 00 pierwszego bębna sąsiaduje ze stykiem wej./wyj. o numerze 00,
- styk 17 drugiego bębna sąsiaduje ze stykiem 00 pierwszego bębna i stykiem 28 trzeciego bębna
- styk 28 trzeciego bębna sąsiaduje ze stykiem 17 drugiego bębna i stykiem 00 bębna odwracającego.

**Wyjście**

Zaszyfrowane ciągi znakowe powinny być umieszczone w kolejnych wierszach pliku o nazwie **ciphertext.txt**. Znak końca wiersza ma kod 10<sub>dec</sub>. Plik powinien kończyć się pustym wierszem zawierającym wyłącznie znak końca wiersza.

**Uwagi**

Powyższy algorytm szyfruje w sposób odwracalny. Zatem:

$F(F(X))=X$ , gdzie  $X$  – ciąg wejściowy,  $F$  – operacja szyfrowania

## 3.6 Mapa wysokości

Zrealizować program generujący na podstawie fragmentu lotniczej mapy wysokości zawartej w pliku tekstowym graficzną wizualizację tej mapy w formacie BMP. Dodatkowo należy wykreślić ciągłą linią przekrój (wysokość) terenu pomiędzy dwoma wskazanymi punktami leżącymi na brzegach mapy, zaś na mapie należy nanieść ciągłą linię przekroju.

### Wejście

Dane wejściowe dla programu znajdują się w plikach o nazwie **mapa.txt** oraz **przekroj.txt**.

Plik **mapa.txt** zawiera informacje o wysokości terenu nad poziomem morza (w postaci tekstowej) w postaci macierzy o rozmiarze 201 na 201. Każdy element macierzy jest liczbą całkowitą z zakresu  $\langle 0, 9999 \rangle$  i reprezentuje wysokość dla kwadratu o boku 20 metrów. Liczby rozdzielone są pojedynczymi spacjami.

#### Przykładowa zawartość pliku **mapa.txt**

```
110 113 115 117 119 121 123 126 128 130 132 134 136 138 140 142...
113 115 117 119 122 124 126 128 130 132 135 137 139 141 143 145...
115 117 119 122 124 126 128 131 133 135 137 139 142 144 146 148...
117 119 122 124 126 129 131 133 135 138 140 142 144 146 149 151...
...
```

Plik **przekroj.txt** zawiera współrzędne dwóch punktów pomiędzy którymi należy sporządzić przekrój (zakładamy, że punkty te zawsze będą leżały na brzegu mapy). Pierwsza linia pliku zawiera współrzędne x,y punktu początkowego, zaś druga linia punktu końcowego. Współrzędne są liczbami całkowitymi z zakresu  $\langle 0, 200 \rangle$  i są rozdzielone pojedynczymi spacjami.

#### Przykładowa zawartość pliku **przekroj.txt**

```
0 200
200 0
```

Należy przyjąć, że dolny lewy róg mapy ma współrzędne (0,0).

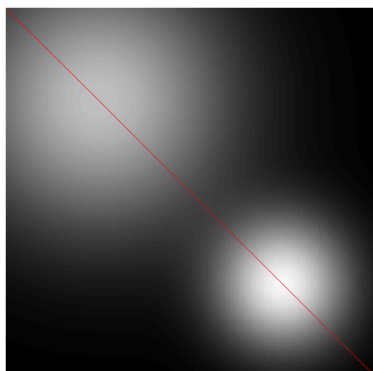
## Wyjście

Wyniki działania programu powinny być zapisane w dwóch plikach w formacie BMP[5] o nazwach **mapa.bmp** i **przekroj.bmp**.

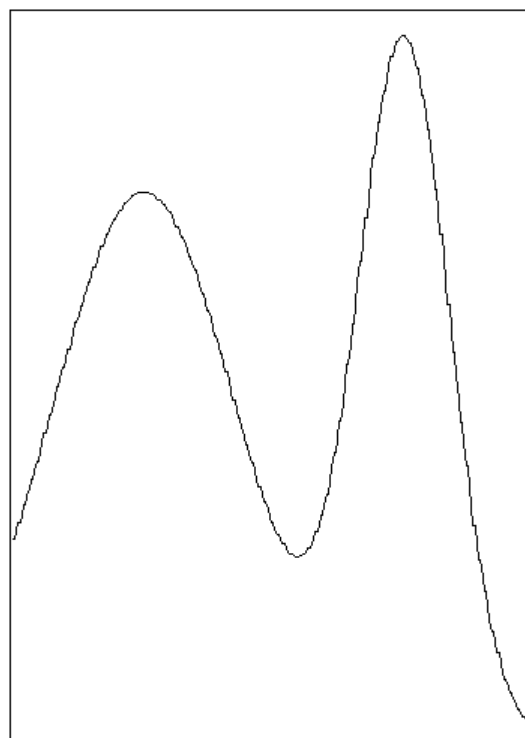
Plik **mapa.bmp** (rys. 1) zawiera obraz mapy wysokości w skali szarości. Obraz ma stałe rozmiary 201x201 pikseli. Wartość 0 z mapy wysokości jest reprezentowana przez kolor czarny. Wartości większe równe 400 reprezentowane są przez kolor biały. Pośrednie wartości reprezentowane są jako odcienie szarości.

Kolorem czerwonym zaznaczono linię wzdłuż której wykonany jest przekrój (linia powinna być ciągła, a nie składać się ze zbioru punktów).

Plik **przekroj.bmp** (rys. 2) zawiera obraz przekroju terenu. Obraz ma stałe rozmiary 285x400 pikseli. Rysunek powinien być sporządzony w tej samej skali co rysunek zawarty w pliku mapa.bmp (np. długość przekątnej z rys. 1 wynosi 284 piksele i taką szerokość ma wykres przekroju). Na osi pionowej 1 piksel odpowiada 1 metrowi. Dolny wiersz pikseli reprezentuje wysokość 0. Lewa strona wykresu odpowiada punktowi początkowemu, prawa – końcowemu. Wykres powinien być linią ciągłą.



Rys.1. Przykładowy obraz z pliku  
mapa.bmp



Rys. 2. Przykładowy obraz z pliku  
przekroj.bmp

## Warianty zadania:

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora MIPS.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone). Należy oszacować jak duże będą błędy w porównaniu z programem, który operowałby na arytmetyce zmiennoprzecinkowej.

## 3.7 Grafika żółwiowa

Zrealizować program generujący na podstawie pliku tekstowego plik w formacie BMP. Plik wejściowy zawiera polecenia grafiki żółwiowej (ang. turtle graphics) według podanej w następnym punkcie składni.

### Wejście

Dane wejściowe dla programu znajdują się w pliku o nazwie **turtle.txt** zawierającym ciąg komend grafiki żółwiowej. Składnia pliku jest następująca:

```
<plik opisu> ::= {<komenda>}
<komenda> ::= <komenda_prosta>
<komenda_prosta> ::= <komenda_ustaw> | <komenda_podnieś> |
                     <komenda_opuść> | <komenda_naprzód> |
                     <komenda_obrót>
<komenda_ustaw> ::= ustaw <punkt> , <kąt> ;
<komenda_podnieś> ::= podnies ;
<komenda_opuść> ::= opusc ;
<komenda_naprzód> ::= naprzod <liczba> ;
<komenda_obrót> ::= obrot <liczba> ;
<komenda_kolor> ::= kolor <kolor> ;
<punkt> ::= [ <liczba> , <liczba> ]
<kąt> ::= <liczba>
<liczba> ::= [0..9] {[0..9]}
```

Należy przyjąć, że dolny lewy róg obrazu powinien mieć współrzędne (0,0). Kąty mierzone są w stopniach.

### Wyjście

Wyniki przetwarzania powinny być zapisane w pliku w formacie BMP o rozmiarach 160x120. Obrazek powinien być biało-czarny (tło białe, rysowane obiekty czarne). Format pliku BMP jest dostępny pod adresem [5].

### Uwagi

Do rysowania linii proszę użyć algorytmu Bresenham'a. Opis można znaleźć w [6].

## 3.8 Odległość Hamminga

Dane są dwa obrazy biało-czarne w dwóch plikach BMP. Zrealizować program, który policzy minimalną odległość Hamminga między tymi obrazami. Odległość Hamminga jest liczbą pikseli, na których różnią się obrazy, z uwzględnieniem możliwego przesunięcia w osi poziomej i pionowej obrazów względem siebie. Zakłada się, że przesunięcie zawiera się w przedziale  $\langle -7, 7 \rangle$  pikseli.

Implementowany algorytm powinien, dla każdego możliwego przesunięcia obrazów względem siebie, wyznaczyć liczbę pikseli, na których różnią się obrazy i wybrać wartość minimalną, jako wynik końcowy.

### Wejście

Dane wejściowe dla programu znajdują się w plikach w formacie BMP (1-bitowych) [5] o nazwie **obraz1.bmp** oraz **obraz2.bmp**. Rozmiar obrazów jest stały i wynosi 64 x 64 piksele.

### Wyjście

Wynikiem działania programu powinny być dwa pliki tekstowe o nazwach **hamming.txt** i **tablica.txt**.

Plik **hamming.txt** zawiera minimalną odległość Hamminga między obrazami dla dowolnego, dopuszczalnego przesunięcia w obu osiach.

Plik **tablica.txt** zawiera tabelę odległości Hamminga dla wszystkich możliwych przesunięć.

### Uwagi

Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone).



## 3.9 Kod kreskowy Code 128

Zrealizować program dekodujący wybrany podzbiór kodów kreskowych zgodnych ze specyfikacją Code 128.

### Opis kodów kreskowych Code 128

Kody kreskowy Code 128 [13, 14] umożliwia zakodowanie 128 symboli z jednego z trzech podzbiorów (Code Set A, Code Set B, Code Set C). Znak startu kodu kreskowego zawiera informację o użytym podziorze znaków. Wykorzystywane są kreski i przerwy o czterech różnych grubościach. Każdy znak (poza znakiem stopu) składa się z trzech kresek i trzech przerw. Znak stopu składa się z czterech kresek i trzech przerw. Szerokości kresek i przerw są całkowitymi wielokrotnościami szerokości najwęższej kreski lub przerwy.

Elementy składowe kodu kreskowego Code 128 to:

- pusta przestrzeń - jej szerokość powinna być dziesięciokrotnie szersza od najwęższej kreski,
- znak startu,
- dane,
- znak kontrolny,
- znak stopu,
- pusta przestrzeń - jej szerokość powinna być dziesięciokrotnie szersza od najwęższej kreski.

### Obliczanie kodu znaku kontrolnego

Znak sumy kontrolnej określany jest na podstawie sumy kodów wszystkich znaków (pomnożonych przez numer pozycji) modulo 103. Pozycja pierwszego znaku danych po lewej stronie ma numer 1.

$$kod\_znaku\_kontrolnego = \left( kod\_startu + \sum_{i=1}^{liczba\ znaków\ danych} i * kod\_znaku[i] \right) mod\ 103$$

### Kody znaków

W tabeli 1 pokazano znaki oraz ich kody dla trzech podzbiorów - Code Set A, Code Set B, Code Set C. Ostatnia kolumna zawiera względne długości kresek (K) i przerw (P).

Tabela 1. Kody znaków

Kod znaku	Code set A	Code set B	Code set C	<u>K</u>	P	<u>K</u>	P	<u>K</u>	P
0	SP	SP	00	2	1	2	2	2	2
1	!	!	01	2	2	2	1	2	2
2	"	"	02	2	2	2	2	2	1
3	#	#	03	1	2	1	2	2	3
4	\$	\$	04	1	2	1	3	2	2
5	%	%	05	1	3	1	2	2	2
6	&	&	06	1	2	2	2	1	3
7	'	'	07	1	2	2	3	1	2
8	(	(	08	1	3	2	2	1	2
9	)	)	09	2	2	1	2	1	3
10	*	*	10	2	2	1	3	1	2
11	+	+	11	2	3	1	2	1	2
12	,	,	12	1	1	2	2	3	2
13	-	-	13	1	2	2	1	3	2
14	.	.	14	1	2	2	2	3	1
15	/	/	15	1	1	3	2	2	2
16	0	0	16	1	2	3	1	2	2
17	1	1	17	1	2	3	2	2	1
18	2	2	18	2	2	3	2	1	1
19	3	3	19	2	2	1	1	3	2
20	4	4	20	2	2	1	2	3	1
21	5	5	21	2	1	3	2	1	2
22	6	6	22	2	2	3	1	1	2
23	7	7	23	3	1	2	1	3	1
24	8	8	24	3	1	1	2	2	2
25	9	9	25	3	2	1	1	2	2
26	:	:	26	3	2	1	2	2	1
27	;	;	27	3	1	2	2	1	2
28	<	<	28	3	2	2	1	1	2
29	=	=	29	3	2	2	2	1	1
30	>	>	30	2	1	2	1	2	3
31	?	?	31	2	1	2	3	2	1
32	@	@	32	2	3	2	1	2	1
33	A	A	33	1	1	1	3	2	3
34	B	B	34	1	3	1	1	2	3
35	C	C	35	1	3	1	3	2	1
36	D	D	36	1	1	2	3	1	3
37	E	E	37	1	3	2	1	1	3
38	F	F	38	1	3	2	3	1	1
39	G	G	39	2	1	1	3	1	3
40	H	H	40	2	3	1	1	1	3
41	I	I	41	2	3	1	3	1	1
42	J	J	42	1	1	2	1	3	3
43	K	K	43	1	1	2	3	3	1
44	L	L	44	1	3	2	1	3	1
45	M	M	45	1	1	3	1	2	3
46	N	N	46	1	1	3	3	2	1
47	O	O	47	1	3	3	1	2	1
48	P	P	48	3	1	3	1	2	1
49	Q	Q	49	2	1	1	3	3	1
50	R	R	50	2	3	1	1	3	1
51	S	S	51	2	1	3	1	1	3
52	T	T	52	2	1	3	3	1	1
53	U	U	53	2	1	3	1	3	1
54	V	V	54	3	1	1	1	2	3
55	W	W	55	3	1	1	3	2	1
56	X	X	56	3	3	1	1	2	1

57	Y	Y	57	3	1	2	1	1	3	
58	Z	Z	58	3	1	2	3	1	1	
59	[	[	59	3	3	2	1	1	1	
60	\	\	60	3	1	4	1	1	1	
61	]	]	61	2	2	1	4	1	1	
62	^	^	62	4	3	1	1	1	1	
63	_	_	63	1	1	1	2	2	4	
64	NUL	`	64	1	1	1	4	2	2	
65	SOH	a	65	1	2	1	1	2	4	
66	STX	b	66	1	2	1	4	2	1	
67	ETX	c	67	1	4	1	1	2	2	
68	EOT	d	68	1	4	1	2	2	1	
69	ENQ	e	69	1	1	2	2	1	4	
70	ACK	f	70	1	1	2	4	1	2	
71	BEL	g	71	1	2	2	1	1	4	
72	BS	h	72	1	2	2	4	1	1	
73	HT	i	73	1	4	2	1	1	2	
74	LF	j	74	1	4	2	2	1	1	
75	VT	k	75	2	4	1	2	1	1	
76	FF	l	76	2	2	1	1	1	4	
77	CR	m	77	4	1	3	1	1	1	
78	SO	n	78	2	4	1	1	1	2	
79	SI	o	79	1	3	4	1	1	1	
80	DLE	p	80	1	1	1	2	4	2	
81	DC1	q	81	1	2	1	1	4	2	
82	DC2	r	82	1	2	1	2	4	1	
83	DC3	s	83	1	1	4	2	1	2	
84	DC4	t	84	1	2	4	1	1	2	
85	NAK	u	85	1	2	4	2	1	1	
86	SYN	v	86	4	1	1	2	1	2	
87	ETB	w	87	4	2	1	1	1	2	
88	CAN	x	88	4	2	1	2	1	1	
89	EM	y	89	2	1	2	1	4	1	
90	SUB	z	90	2	1	4	1	2	1	
91	ESC	{	91	4	1	2	1	2	1	
92	FS		92	1	1	1	1	4	3	
93	GS	}	93	1	1	1	3	4	1	
94	RS	~	94	1	3	1	1	4	1	
95	US	DEL	95	1	1	4	1	1	3	
96	FNC 3	FNC 3	96	1	1	4	3	1	1	
97	FNC 2	FNC 2	97	4	1	1	1	1	3	
98	SHIFT	SHIFT	98	4	1	1	3	1	1	
99	CODE C	CODE C	99	1	1	3	1	4	1	
100	CODE B	FNC 4	CODE B	1	1	4	1	3	1	
101	FNC 4	CODE A	CODE A	3	1	1	1	4	1	
102	FNC 1	FNC 1	FNC 1	4	1	1	1	3	1	
103	Start A	Start A	Start A	2	1	1	4	1	2	
104	Start B	Start B	Start B	2	1	1	2	1	4	
105	Start C	Start C	Start C	2	1	1	2	3	2	
106	Stop	Stop	Stop	2	3	3	1	1	1	2

**Wejście**

Dane wejściowe dla programu znajdują się w 24 bitowym pliku BMP[5] bez kompresji o nazwie **kod.bmp**. Rozmiar plików jest stały i wynosi 600pikseli na 50 pikseli. Pliki nie spełniające w/w wymagań nie podlegają przetwarzaniu.

Kreski narysowane są równoległe do pionowej krawędzi kolorem czarnym, tło ma kolor biały. W obrazie nie występują żadne zniekształcenia (np. przekos, niedokładności druku itp.).

**Wyjście**

Program powinien wypisywać wyniki dekodowania kodu kreskowego na standardowym wyjściu.

**Warianty zadania:**

1. Program dekoduje znaki z zestawu Code Set A
2. Program dekoduje znaki z zestawu Code Set B
3. Program dekoduje znaki z zestawu Code Set C
4. Program dekoduje znaki z dowolnego zestawu.

**Uwagi:**

Rozpoznawanie kodów kreskowych on-line:

<http://online-barcode-reader.inliteresearch.com/default.aspx>

Generowanie kodów kreskowych on-line:

<http://www.barcode-generator.org/>

### 3.10 Kod kreskowy Code 39

Zrealizować program dekodujący kod kreskowy zgodny ze specyfikacją Code 39.

#### Opis kodów kreskowych Code 39

Kody kreskowy Code 39 [15] umożliwia zakodowanie 43 symboli. Znak startu i stopu kodu kreskowego to \* (znak ten nie może być wykorzystany w kodowanych danych). Wykorzystywane są kreski i przerwy o dwóch grubościach - stosunek grubości kresek i przerw szerokich do kresek i przerw wąskich może wynosić od 2.2:1 do 3:1 (w obrębie jednego kodu jest stały). Każdy znak składa się z 9 pól (5 kresek i 4 przerw) z czego 3 pola są szerokie. Szerokość przerwy między znakami nie jest zdefiniowana - w praktyce przyjmuje się, że jest równa wąskiej przerwie.

Elementy składowe kodu kreskowego Code 39 to:

- znak startu \*,
- dane,
- znak kontrolny,
- znak stopu \*,

#### Obliczanie kodu znaku kontrolnego

Znak sumy kontrolnej określany jest na podstawie sumy kodów wszystkich znaków modulo 43.


$$kod\_znaku\_kontrolnego = \left( \sum_{i=1}^{liczba\ znaków\ danych} kod\_znaku[i] \right) \bmod 43$$

#### Kody znaków

W tabeli 1 pokazano znaki oraz ich kody. Ostatnia kolumna zawiera względne długości kresek (K) i przerw (P). Cyfra 2 oznacza przerwę lub kreskę szeroką, zaś 1 wąską.

Tabela 1. Kody znaków

Kod znaku	Char	<u>K</u>	<u>P</u>	<u>K</u>	<u>P</u>	<u>K</u>	<u>P</u>	<u>K</u>
0	0	1	1	1	2	2	1	2
1	1	2	1	1	2	1	1	2
2	2	1	1	2	2	1	1	2
3	3	2	1	2	2	1	1	1
4	4	1	1	1	2	2	1	2
5	5	2	1	1	2	2	1	1
6	6	1	1	2	2	2	1	1
7	7	1	1	1	2	1	1	2
8	8	2	1	1	2	1	1	1
9	9	1	1	2	2	1	1	2
10	A	2	1	1	1	2	1	2
11	B	1	1	2	1	1	2	1
12	C	2	1	2	1	1	2	1
13	D	1	1	1	1	2	2	1
14	E	2	1	1	1	2	2	1
15	F	1	1	2	1	2	2	1
16	G	1	1	1	1	1	2	2
17	H	2	1	1	1	1	2	2
18	I	1	1	2	1	1	2	2
19	J	1	1	1	1	2	2	2
20	K	2	1	1	1	1	1	2

21	L	1	1	2	1	1	1	2	2
22	M	2	1	2	1	1	1	2	1
23	N	1	1	1	1	2	1	2	2
24	O	2	1	1	1	2	1	2	1
25	P	1	1	2	1	2	1	2	1
26	Q	1	1	1	1	1	2	2	2
27	R	2	1	1	1	1	2	2	1
28	S	1	1	2	1	1	2	2	1
29	T	1	1	1	1	2	1	2	2
30	U	2	2	1	1	1	1	1	2
31	V	1	2	2	1	1	1	1	2
32	W	2	2	2	1	1	1	1	1
33	X	1	2	1	1	2	1	1	2
34	Y	2	2	1	1	2	1	1	1
35	Z	1	2	2	1	2	1	1	1
36	-	1	2	1	1	1	2	1	2
37	.	2	2	1	1	1	2	1	1
38	space	1	2	2	1	1	2	1	1
39	\$	1	2	1	2	1	2	1	1
40	/	1	2	1	2	1	1	2	1
41	+	1	2	1	1	1	2	1	2
42	%	1	1	1	2	1	2	1	2
	*	1	2	1	1	2	1	2	1

## Wejście

Dane wejściowe dla programu znajdują się w 24 bitowym pliku BMP[5] bez kompresji o nazwie **kod.bmp**. Rozmiar plików jest stały i wynosi 600pikseli na 50 pikseli. Pliki nie spełniające w/w wymagań nie podlegają przetwarzaniu.

Kreski narysowane są równoległe do pionowej krawędzi kolorem czarnym, tło ma kolor biały. W obrazie nie występują żadne zniekształcenia (np. przekos, niedokładności druku itp.).

## Wyjście

Program powinien wypisywać wyniki dekodowania kodu kreskowego na standardowym wyjściu.

## Uwagi:

Rozpoznawanie kodów kreskowych on-line:

<http://online-barcode-reader.inliteresearch.com/default.aspx>

Generowanie kodów kreskowych on-line:

<http://generator.onbarcode.com/online-code-39-barcode-generator.aspx>

## 3.11 Uwagi dotyczące projektów dla procesora MIPS

Fragmenty kodu służącego do odczytu danych z pliku [2]:

```
#otwarcie pliku
    li $v0, 13          #system call for file_open
    la $a0, filename    #address of filename string
    li $a1, 0           #in mars set to 0
    li $a2, 0           #in mars set to 0
    syscall             #file descriptor of opened file in v0

#sprawdzenie czy plik udało sie otworzyc file_descriptor<>-1
    ...

#odczyt danych z pliku
    li $v0, 14          #system call for file_read
    move $a0, $...      #move file descr from ... to a0
    la $a1, buf         #address of data buffer
    li $a2, 4048        #amount to read (bytes)
    syscall

#sprawdzenie ile danych wczytano
    beq $zero,$v0, fclose #branch if no data is read
    ...

#zamknięcie pliku
    li $v0, 16          #system call for file_close
    move $a0, $...      #move file descr from ... to a0
    syscall
```

## 4 Małe programy dla procesora Intel

Korzystając z szablonu programu umieszczonego pod adresem [galera.ii.pw.edu.pl/~zsz/arko/intel-wstep.tar.gz](http://galera.ii.pw.edu.pl/~zsz/arko/intel-wstep.tar.gz) napisać program w asemblerze procesora x86. Program powinien wczytać z klawiatury ciąg znaków, poddać konwersji ciąg znaków, wyświetlić zmodyfikowany ciąg znaków na ekranie.

1a	Zamienić wszystkie litery umieszczone pomiędzy skrajnymi znakami * na wielkie. Podaj ciąg znakow> Ala *Ma*Kot*a Ciąg po konwersji> Ala *MA*KOT*a
1b	Zamienić wszystkie litery umieszczone przed i po skrajnym znaku * na małe. Podaj ciąg znakow> ALA *MA*KOT*A Ciąg po konwersji> ala *MA*KOT*a
1c	Zamienić k znaków po pierwszej cyfrze w ciągu na małe litery (k jest wartością pierwszej cyfry). Podaj ciąg znakow> ALA 3MA KOTA Ciąg po konwersji> ALA 3ma KOTA
1d	Zamienić k znaków przed ostatnią cyfrą w ciągu na małe litery (k jest wartością pierwszej cyfry). Podaj ciąg znakow> ALA 3MA KOTA Ciąg po konwersji> Ala 3MA KOTA
2a	Zastąpić najczęściej występujący znak na *. Podaj ciąg znakow> ALA MA KOTA Ciąg po konwersji> *L* M* KOT*
2b	Zastąpić znaki w najdłuższej grupie liter na * (zakładamy, że grup jest max. 256). Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> Ala Ma ****
2c	Zastąpić znaki w najkrótszej grupie liter na * (zakładamy, że grup jest max. 256). Podaj ciąg znakow> Ala Ma Kota Ciąg po konwersji> Ala ** Kota

## 5 Projekty dla procesora Intel

Projety dla procesora Intel x86 tworzone są z wykorzystaniem kompilatora gcc oraz asemblera NASM. W celu automatyzacji procesu kompilacji poszczególnych plików, a następnie linkowania programu wynikowego wykorzystywany jest plik Makefile. Celem przedstawionych w niniejszym opracowaniu ćwiczeń jest wywołanie z programu napisanego w języku C funkcji asemblerowych.

### **Wymagania dotyczące dokumentacji**

Wraz z działającym programem należy przedstawić dokumentację projektu. Powinna zawierać m.in.:

- opis struktury programu,
- opis struktur danych i ich implementacji,
- opis implementacji algorytmu,
- opis formatu stosowanych plików (jeśli format nie wynika wprost z treści zadania),

dobre zdokumentowane wyniki testowania dowodzące, że program działa poprawnie. Powinny zawierać opis procedury testowania - z dokumentacji powinno jasno wynikać co było testowane, na jakich danych przeprowadzano testy, jakie były wyniki testowania, czy test był poprawny/niepoprawny (forma tabelaryczna będzie mile widziana) i ew. stopień kompresji. Na podstawie dostarczonych plików i opisu testowania osoba sprawdzająca program powinna być w stanie powtórzyć testy.



## 5.1 Kompresja danych

Zrealizować programy do kompresji (o nazwie *compress*) oraz dekompresji pliku (o nazwie *decompress*) w oparciu o kodowanie Huffmana. Opis algorytmu, który należy zaimplementować został podany w treści zadania 1.1 (projekt realizowany w assemblerze procesora MIPS).

### Wejście / Wyjście

Wejściem do programu kompresującego *compress* jest plik, którego nazwa jest podana jako pierwszy parametr wywołania programu. Plik wejściowy może mieć dowolną wielkość (nie można zatem wczytać całego pliku na jeden raz do pamięci). W wyniku kompresji powstaje plik o nazwie podanej jako drugi parametr wywołania programu, zawierający zakodowane dane oraz wszystkie dodatkowe informacje potrzebne do poprawnej dekompresji.

Wejściem do programu dekompresującego *decompress* jest plik o nazwie podanej jako pierwszy parametr wywołania programu. W wyniku dekompresji powstaje plik, którego nazwa podana jest jako drugi parametr wywołania programu.

### Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. W programie należy zaimplementować w assemblerze Intela następujące funkcje:
  - **build\_table** – funkcja tworząca tablicę częstości występowania symboli
  - **build\_tree** – funkcja tworząca drzewo
  - **compress** – funkcja dokonująca kompresji
  - **decompress** – funkcja dokonująca dekompresji

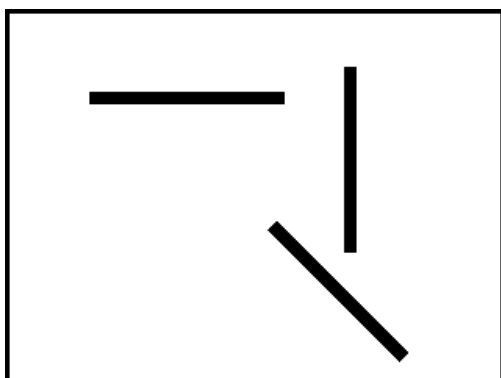
## 5.2 Co widzi robot

Zrealizować program symulujący laserowy skaner robota służący do pomiaru odległości. Otoczenie robota (przeszkody) jest opisane w pliku w formacie BMP [5], a parametry aktualnego położenia w pliku tekstowym. Należy wykreślić ciągłą czerwoną linią przebieg wiązek lasera oraz zapisać w pliku tekstowym ich długości. Program powinien być napisany w języku C/C++ z wybranymi funkcjami zaimplementowanymi w asemblerze procesora Intel (32 bitowym).

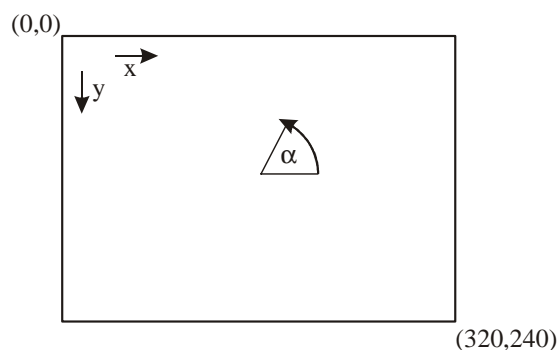
### Wejście

Dane wejściowe dla programu znajdują się w plikach o nazwie **otoczenie.bmp** oraz **parametry.txt**.

Plik **otoczenie.bmp** zawiera informacje o rozmieszczeniu przeszkód i ma rozmiar 320 na 240 pikseli. Piksel o kolorze białym oznacza wolną przestrzeń, zaś o kolorze czarnym - przeszkodę.



**Rys. 1.** Przykładowa zawartość pliku *otoczenie.bmp*.



**Rys. 2.** Zastosowany układ współrzędnych.

Plik **parametry.txt** zawiera w kolejnych wierszach informację o położeniu robota (liczby są rozdzielone spacjami) w kolejnych chwilach czasu:

- $x, y$  - współrzędne położenia robota – dwie liczby całkowite z zakresu  $\langle 0, 320 \rangle$  i  $\langle 0, 240 \rangle$ ,
- $\alpha$  - kierunek, w którym skierowany jest robot (wyrażony w stopniach) – liczba całkowita z zakresu  $\langle 0, 360 \rangle$

#### Przykładowa zawartość pliku parametry.txt

```
100 100 45
105 105 45
```

Parametry stałe:

- maksymalna długość wiązki laserowej – 60 pikseli,

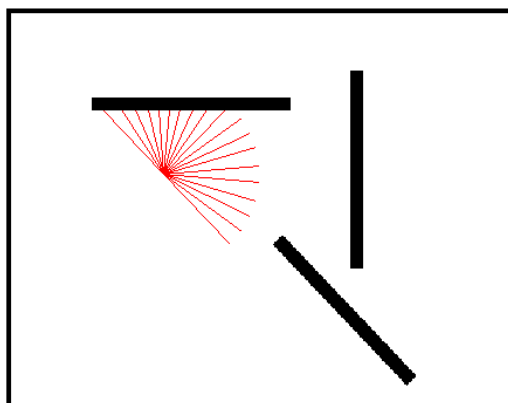
- liczba wiązek laserowych – 19. Wiązki są rozmieszczone równomiernie co  $10^\circ$  w zakresie  $\langle 90^\circ, -90^\circ \rangle$  od kierunku, w którym skierowany jest robot.

## Wyjście

Wyniki działania programu powinny być zapisane w plikach o nazwach **symulacjaXX.bmp** i **wyniki.txt**. Gdzie XX jest indeksem chwili czasu z pliku parametry.txt. Dla przykładowego pliku parametry.txt powstaną trzy pliki: symulacja00.bmp, symulacja01.bmp oraz wyniki.txt.

Plik **symulacjaXX.bmp** (przykład na rys. 3) zawiera obraz otoczenia z pliku wejściowego otoczenie.bmp oraz naniesione czerwonymi ciągłymi liniami przebiegi wiązek laserowych. Obraz ma stałe rozmiary 320x240 pikseli.

Plik **wyniki.txt** zawiera tyle wierszy ile położeń robota zostało określonych w pliku parametry.txt. Wiersz zawiera długości wiązek laserowych (odległości od przeszkody) rozdzielone spacjami. W przypadku, gdy wiązka nie natrafi na przeszkodę do pliku powinna być wstawiona wartość 255. Skanowanie rozpoczyna się od kąta  $90^\circ$  (lewa strona robota).



### Przykładowa zawartość pliku wyniki.txt

55 47 43 40 39 39 40 43 47 55 255...
...

Rys.3. Przykładowy obraz z pliku  
symulacja00.bmp

## Uwagi

1. W programie należy zaimplementować w asemblerze Intel'a następujące funkcje:

### Funkcja obliczająca długość wiązki laserowej

```
int RayLength(unsigned char *image, int x, int y, int alfa,
              int ray, RayStruct *rs);
```

Wartość zwracana: długość wiązki lub 255.

Parametry:

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówek BMP)
- **int x, y, alfa** – położenie robota
- **int ray** – numer wiązki:  $\langle 0-18 \rangle$
- **RayStruct \*rs** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego działania funkcji (jeśli potrzebne).

### Funkcja tworząca obraz wiązek

```
int draw(int *rays, unsigned char *image,
         ImageStruct *is);
```

Wartość zwracana: zawsze zero.

Parametry:

- **int \*rays** – tablica zawierająca długości wiązek
- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówek BMP)
- **ImageStruct \*im** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego działania funkcji (jeśli potrzebne).

2. Odczyt i zapis danych z plików realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

#### **Warianty zadania:**

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone). Należy oszacować jak duże będą błędy w porównaniu z programem, który operowałby na arytmetyce zmiennoprzecinkowej.

## 5.3 Z-bufor

Należy napisać program symulujący działanie mechanizmu Z-bufora [7]. Program ma za zadanie rysować trójkąty, których parametry wczytane są z pliku, zaś efekty działania programu powinny być zapisane do plików BMP. Przy określaniu kolorów pikseli należy użyć metody cieniowania interpolowanego (takiego jak w metodzie cieniowania Gouraud).

Program powinien być napisany w języku C/C++ z wybranymi funkcjami zaimplementowanymi w asemblerze procesora Intel (32 bitowym).

### Z-bufor

Opis mechanizmu Z-bufora zawarty jest w rozdziale 3.3.

### Cieniowanie interpolowane

Opis cieniowania interpolowanego zawarty jest w rozdziale 3.3.

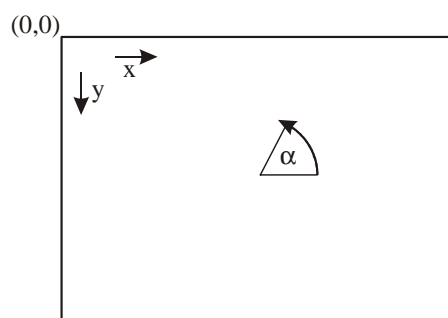
### Wejście

Dane wejściowe dla programu znajdują się w pliku tekstowym o nazwie **opis.txt** o następującym formacie:

- Pierwszy wiersz zawiera wartości rozmiar w poziomie (Xsize) i pionie (Ysize) generowanego obrazu. Są to dwie liczby całkowite rozdzielone spacjami.
- Drugi wiersz zawiera wartości trzech składowych RGB koloru tła rozdzielone spacjami. Wartości zawierają się w przedziale  $\langle 0, 255 \rangle$ .
- Kolejne wiersze zawierają opis trójkątów:

X1 Y1 Z1 R1 G1 B1 X2 Y2 Z2 R2 G2 B2 X3 Y3 Z3 R3 G3 B3

gdzie:  $X_i$ ,  $Y_i$ ,  $Z_i$  są współrzędnymi wierzchołka  $X, Y$  i odpowiadającą mu składową  $Z$ , zaś  $R_i$ ,  $G_i$ ,  $B_i$  są składowymi RGB koloru wierzchołka. Wartości współrzędnej  $X$  zawierają się w przedziale  $\langle 0, Xsize-1 \rangle$ , współrzędnej  $Y$   $\langle 0, Ysize-1 \rangle$ . Składowa  $Z$  może przyjmować wartości z zakresu  $\langle 0x0000\ 0000, 0xFFFF\ FFFE \rangle$  (w pliku wyrażone w systemie dziesiętnym). Wartości RGB zawierają się w przedziale  $\langle 0, 255 \rangle$ . Wierzchołki trójkątów mogą występować w dowolnej kolejności.



Rys. 2. Zastosowany układ współrzędnych.

## Wyjście

Wynikiem działania programu powinny być dwa pliki w formacie BMP [5] o nazwach **scena.bmp** i **zbufor.bmp**. Plik **scena.bmp** zawiera narysowane trójkąty opisane w pliku **opis.txt**. Plik **zbufor.bmp** zawiera wizualizację zawartości Z-bufora w odcieniach szarości. Wartość 0 reprezentowana przez kolor biały, nieskończoność przez kolor czarny.

## Uwagi

1. W programie należy zaimplementować w assemblerze Intela następujące funkcje:

### Funkcja rysująca trójkąt

```
int DrawTriangle(unsigned char *image,
                 unsigned char *zbuf, int xsize, int ysize,
                 int *vertices, int *rgb);
```

Wartość zwracana: 0 w przypadku powodzenia lub kod błędu w przeciwnym przypadku.

Parametry:

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówków BMP)
- **unsigned char \*zbuf** – wskazanie na bufor zawierający z-bufor
- **int xsize** – rozmiar poziomy obrazu w pikselach
- **int ysize** – rozmiar pionowy obrazu w pikselach
- **int \*vertices** – wskazanie na tablicę zawierającą wierzchołki trójkąta (w kolejności X1, Y1, Z1, X2, Y2, Z2, X3, Y3, Z3). Wierzchołki przekazywane są w takiej samej kolejności jak zostały wczytane z pliku.
- **int \*rgb** – wskazanie na tablicę zawierającą składowe RGB wierzchołków umieszczonych w tablicy vertices.

### Funkcja inicjująca bufor

```
int InitBuffers(unsigned char *image, unsigned char *zbuf,
                int xsize, int ysize, int *rgb);
```

Wartość zwracana: 0 w przypadku powodzenia lub kod błędu w przeciwnym przypadku.

Parametry:

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówków BMP)
- **unsigned char \*zbuf** – wskazanie na bufor zawierający z-bufor
- **int xsize** – rozmiar poziomy obrazu w pikselach
- **int ysize** – rozmiar pionowy obrazu w pikselach
- **int \*rgb** – wskazanie na tablicę zawierającą składowe RGB tła obrazu.

2. Odczyt i zapis danych z plików realizowany jest na poziomie języka C.

3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

## Warianty zadania

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone).

## 5.4 EKG

Realizowany program powinien dokonać detekcji zespołów QRS w sygnale EKG. Sygnał jest zapisany w pliku. Opis algorytmu detekcji, który należy zaimplementować został podany w treści zadania 3.4 (projekt realizowany w asemblerze procesora MIPS). Wyniki analizy należy wyświetlić na standardowym wyjściu w zadanym formacie.

### Wejście

Dane wejściowe dla programu znajdują się w pliku zawierającym ciąg próbek sygnału EKG. Próbką sygnału EKG jest 16-bitową liczbą całkowitą ze znakiem.

Przykładowy plik jest dostępny pod adresem:

**`http://galera.ii.pw.edu.pl/~zsz/arko/projekt-intel/ekg.bin`**

### Wyjście

Program powinien wypisywać wyniki na standardowym wyjściu. Pojedynczy wiersz wyniku ma następujący format:

***nr próbki:wartość próbki:wartość pochodnej:znacznik detekcji***

*Numer próbki* jest kolejnym numerem odczytanej z pliku próbki (numeracja od zera). *Wartość próbki* jest wartością odczytaną z pliku. *Wartość pochodnej* jest wartością obliczoną zgodnie ze wzorem na pochodną w algorytmie Holsingera. Jeśli dana próbka została uznana za punkt detekcji to *znacznik detekcji* przyjmuje wartość 800. W przeciwnym przypadku ma wartość 0. Przykładowy fragment danych generowanych przez program:

```
...
230:-9:69:0
231:8:80:0
232:29:93:0
233:56:122:800
234:84:151:0
...
```

### Uwagi

1. W programie należy zaimplementować funkcję dokonującą detekcji zespołów QRS na buforze zawierającym próbki sygnału wczytane z pliku. Prototyp funkcji w języku C:

```
int detekcja_qrs(short int *bufor,  

                int rozmiar,  

                DetectionStruct *dt);
```

Wartość zwracana: liczba wykrytych zespołów QRS w buforze.

Parametry:

- **short int \*bufor** – wskazanie na bufor zawierający wczytane z pliku próbki sygnału EKG

- **int rozmiar** – liczba próbek sygnału EKG w buforze
- **DetectionStruct \*dt** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego procesu detekcji np.:

```
struct DetectionStruct
{
    short int *pochodna; //wskazanie na bufor
                        //przechowujacy wartosc pochodnej
    int rozmiar;        //rozmiar bufora
    //... i ewentualnie inne potrzebne pola
}
```

2. Odczyt danych z pliku realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
4. Nazwa pliku zawierającego próbki sygnału EKG podawana jest jako pierwszy parametr wywołania programu.
5. W celu zapisania wyników działania programu należy uruchmić go z wiersza poleceń w następujący sposób:

```
./mojprogram ekg.bin > wyniki.txt
```



## 5.5 Mini Enigma

### Cel programu

Zrealizować program szyfrujący ciągi znakowe (pobierane z pliku) algorytmem używanym w maszynie Enigma. Opis algorytmu, który należy zaimplementować został podany w treści zadania 3.5 (projekt realizowany w assemblerze procesora MIPS).

### Wejście

#### Ciągi znakowe do zaszyfrowania:

Ciągi znakowe do zaszyfrowania (tekst jawny) znajdują się w kolejnych wierszach pliku o nazwie **plaintext.txt**. Zakładamy, że:

- długość wiersza tekstu razem ze znakiem końca wiersza (o kodzie 10<sub>dec</sub>) jest nie większa niż 1024 znaki.
- tekst jawny zawiera znaki o kodach ASCII z przedziału <32<sub>dec</sub>, 95<sub>dec</sub>>
- wszystkie znaki, których kody ASCII nie mieszczą się w przedziale wymienionym w poprzednim punkcie są ignorowane
- plik będzie wczytywany wiersz po wierszu (a nie na jeden raz w całości)
- po wczytaniu kolejnego wiersza algorytm szyfrujący kontynuuje pracę (nie powraca do stanu początkowego)
- dane kończą się pustym wierszem zawierającym tylko znak o kodzie 10<sub>dec</sub>

#### Połączenia bębnow szyfrujących:

Plik o nazwie **rotors.txt** zawiera opis połączeń bębnow szyfrujących. Plik składa się z czterech sekcji – trzy pierwsze opisują połączenia bębnow standardowych (w stałej kolejności: bęben nr 1, 2, 3), ostatnia bębna odwracającego. Każda sekcja składa się z etykiety (dowolny ciąg znakowy) oraz 64 linii opisujących połączenia styków po prawej i po lewej stronie bębna (dla bębnow standardowych) i 32 linie (dla bębna odwracającego).

#### Przykład sekcji opisującej połączenia bębna nr 1

Rotor 1;nr styku po lewej stronie- nr styku po prawej stronie
00-32
01-34
02-36
...
62-03
63-01

Powyższe dane oznaczają, że np. lewy styk 00 jest połączony z prawym stykiem 32.

#### Początkowe położenie bębnow szyfrujących:

Początkowe położenie bębnow szyfrujących podane jest w pliku **init.txt**. W kolejnych wierszach podane jest początkowe położenie bębnow nr 1, 2, 3.

---

**Przykładowa zawartość pliku init.txt**


---

00  
17  
28

---

Dane z powyższego przykładu oznaczają, że:

- styk 00 pierwszego bębna sąsiaduje ze stykiem wej./wyj. o numerze 00,
- styk 17 drugiego bębna sąsiaduje ze stykiem 00 pierwszego bębna i stykiem 28 trzeciego bębna
- styk 28 trzeciego bębna sąsiaduje ze stykiem 17 drugiego bębna i stykiem 00 bębna odwracającego.

### Wyjście

Zaszyfrowane ciągi znakowe powinny być umieszczone w kolejnych wierszach pliku o nazwie **ciphertext.txt**. Znak końca wiersza ma kod 10<sub>dec</sub>. Plik powinien kończyć się pustym wierszem zawierającym wyłącznie znak końca wiersza.

### Uwagi

1. W programie należy zaimplementować w assemblerze Intela funkcję dokonującą szyfrowania zadany algorytmem jednego wiersza tekstu. Prototyp funkcji w języku C:

```
int enigma(char *intxt, char *outtxt,  
            EnigmaStruct *en);
```

Wartość zwracana: liczba zaszyfrowanych znaków.

Parametry:

- **char** \***intxt** – wskazanie na bufor zawierający tekst wejściowy (do zaszyfrowania bądź odszyfrowania)
- **char** \***outtxt** – wskazanie na bufor zawierający tekst wyjściowy (zaszyfrowany bądź odszyfrowany)
- **EnigmaStruct** \***en** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego działania algorytmu np.:

```
struct EnigmaStruct  
{  
    int *rotor1; //opis połączeń pierwszego bębna  
    int initpos1; //pozycja początkowa pierwszego bębna  
    //... i ewentualnie inne potrzebne pola  
}
```

2. Odczyt danych z pliku realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

## 5.6 Mapa wysokości

Zrealizować program generujący na podstawie fragmentu lotniczej mapy wysokości zawartej w pliku tekstowym graficzną wizualizację tej mapy w formacie BMP. Dodatkowo należy wykreślić ciągłą linią przekrój (wysokość) terenu pomiędzy dwoma wskazanymi punktami, zaś na mapie należy nanieść ciągłą linię przekroju.

### Wejście

Dane wejściowe dla programu znajdują się w plikach o nazwie **mapa.txt** oraz **parametry.txt**.

Plik **mapa.txt** zawiera informacje o wysokości terenu nad poziomem morza (w postaci tekstowej) w postaci macierzy o rozmiarze 201 na 201. Każdy element macierzy jest liczbą całkowitą z zakresu  $\langle 0, 9999 \rangle$  i reprezentuje wysokość dla kwadratu o boku 20 metrów. Liczby rozdzielone są pojedynczymi spacjami.

#### Przykładowa zawartość pliku **mapa.txt**

```
110 113 115 117 119 121 123 126 128 130 132 134 136 138 140 142...
113 115 117 119 122 124 126 128 130 132 135 137 139 141 143 145...
115 117 119 122 124 126 128 131 133 135 137 139 142 144 146 148...
117 119 122 124 126 129 131 133 135 138 140 142 144 146 149 151...
...
```

Plik **parametry.txt** zawiera:

- współrzędne dwóch punktów pomiędzy którymi należy sporządzić przekrój – pierwsze dwa wiersze
- wartość komórki mapy, która będzie reprezentowana przez kolor czarny (MIN)
- wartość komórki mapy, która będzie reprezentowana przez kolor biały (MAX)

Pierwsza linia pliku zawiera współrzędne  $x, y$  punktu początkowego, zaś druga linia punktu końcowego. Współrzędne są liczbami całkowitymi z zakresu  $\langle 0, 200 \rangle$  i są rozdzielone pojedynczymi spacjami.

Trzecia linia zawiera wartość komórki mapy, która będzie reprezentowana przez kolor czarny. Wszystkie wartości mniejsze od podanej, również mają być reprezentowane przez kolor czarny. Czwarta linia zawiera wartość komórki mapy, która będzie reprezentowana przez kolor biały. Wszystkie wartości większe od podanej, również mają być reprezentowane przez kolor biały.

#### Przykładowa zawartość pliku **parametry.txt**

```
0 200
200 0
10
400
```

Należy przyjąć, że dolny lewy róg mapy ma współrzędne (0,0).

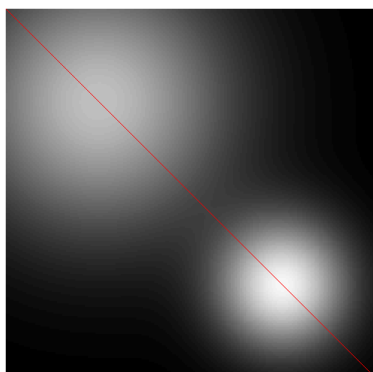
## Wyjście

Wyniki działania programu powinny być zapisane w dwóch plikach w formacie BMP[5] o nazwach, których nazwy są podane jako parametry wywołania programu.

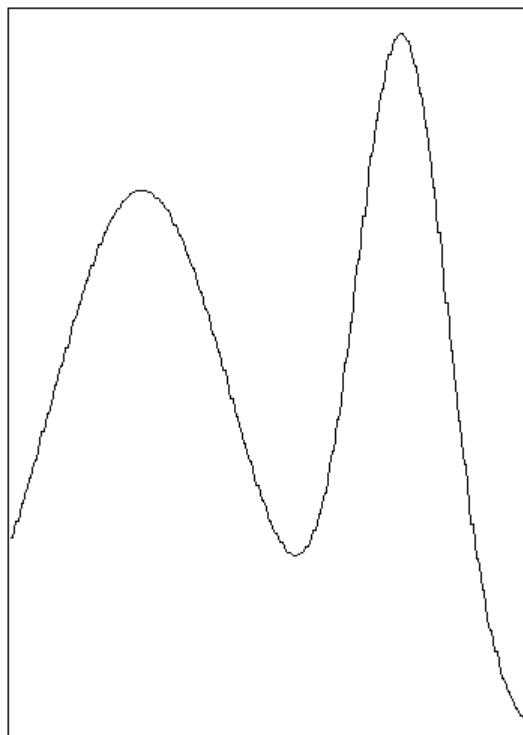
**Plik mapy** (rys. 1) zawiera obraz mapy wysokości w skali szarości. Obraz ma stałe rozmiary 201x201 pikseli. Wartość MIN z mapy wysokości jest reprezentowana przez kolor czarny. Wartości większe równe MAX reprezentowane są przez kolor biały. Pośrednie wartości reprezentowane są jako odcienie szarości.

Kolorem czerwonym zaznaczono linię wzdłuż której wykonany jest przekrój (linia powinna być ciągłą, a nie składać się ze zbioru punktów).

**Plik przekroju** (rys. 2) zawiera obraz przekroju terenu. Rysunek powinien być sporządzony w tej samej skali co rysunek zawarty w pliku mapy (np. długość przekątnej z rys. 1 wynosi 284 piksele i taką szerokość ma wykres przekroju). Na osi pionowej 1 piksel odpowiada 1 metrowi. Dolny wiersz pikseli reprezentuje wysokość 0. Lewa strona wykresu odpowiada punktowi początkowemu, prawa – końcowemu. Wykres powinien być linią ciągłą.



Rys.1. Przykładowy obraz z pliku mapy.



Rys. 2. Przykładowy obraz przekroju.

## Uwagi

1. Wywołanie programu:  
`mapa nazwa_pliku_mapy.bmp nazwa_pliku_przekroju.bmp`
2. W programie należy zaimplementować w assemblerze Intel'a następujące funkcje:

### Funkcja tworząca obraz mapy

```
int mapa(int *mapa, unsigned char *image, MapaStruct *par);
```

Wartość zwracana: zawsze zero.

Parametry:

- `int *mapa` – wskazanie na bufor zawierający dane wejściowe mapy

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówek BMP)
- **MapaStruct \*par** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego działania funkcji.

#### **Funkcja tworząca obraz przekroju**

```
int przekroj(int *mapa, unsigned char *image,  
             PrzekStruct *przek);
```

Wartość zwracana: długość przekroju.

Parametry:

- **int \*mapa** – wskazanie na bufor zawierający dane wejściowe mapy
- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówek BMP)
- **PrzekStruct \*przek** – struktura służąca do przekazania dodatkowych danych potrzebnych do prawidłowego działania funkcji.

3. Odczyt danych z pliku realizowany jest na poziomie języka C.
4. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

#### **Warianty zadania:**

1. Dozwolone jest wykorzystanie dowolnych instrukcji procesora.
2. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone). Należy oszacować jak duże będą błędy w porównaniu z programem, który operowałby na arytmetyce zmiennoprzecinkowej.

## 5.7 Grafika żółwiowa

Zrealizować program (dla procesora x86) generujący na podstawie pliku tekstowego plik w formacie BMP. Plik wejściowy zawiera polecenia grafiki żółwiowej (ang. turtle graphics) według podanej w następnym punkcie składni.

### Wejście

Dane wejściowe dla programu znajdują się w pliku o nazwie **turtle.txt** zawierającym ciąg komend grafiki żółwiowej. Składnia pliku jest następująca:

```
<plik opisu> ::= {<komenda>}
<komenda> ::= <komenda_prosta>
<komenda_prosta> ::= <komenda_ustaw> | <komenda_podnieś> |
                     <komenda_opuść> | <komenda_naprzód> |
                     <komenda_obrót>
<komenda_ustaw> ::= ustaw <punkt> , <kąt> ;
<komenda_podnieś> ::= podnies ;
<komenda_opuść> ::= opusc ;
<komenda_naprzód> ::= naprzod <liczba> ;
<komenda_obrót> ::= obrot <liczba> ;
<komenda_kolor> ::= kolor <kolor> ;
<punkt> ::= [ <liczba> , <liczba> ]
<kąt> ::= <liczba>
<liczba> ::= [0..9] {[0..9]}
```

Należy przyjąć, że dolny lewy róg obrazu powinien mieć współrzędne (0,0). Kąty mierzone są w stopniach.

### Wyjście

Wyniki przetwarzania powinny być zapisane w pliku w formacie BMP o rozmiarach 160x120. Obrazek powinien być biało-czarny (tło białe, rysowane obiekty czarne). Format pliku BMP jest dostępny pod adresem [5].

### Uwagi

1. Należy zaimplementować w assemblerze funkcję wykonującą polecenia grafiki żółwiowej na buforze zawierającym obraz. Stworzoną funkcję należy użyć w programie napisanym w języku C. Prototyp funkcji w języku C:

```
#define komenda_ustaw 1
#define komenda_podnies 2
#define komenda_opusc 3
#define komenda_naprzod 4
#define komenda_obrot 5
```

```
int turtle( int komenda, int param1, int param2,
            DrawingContextStruct *dc);
```

Wartość zwracana: 0 w przypadku powodzenia, liczba większa od 0 oznaczająca numer błędu.

Parametry:

**int komenda** – numer komendy grafiki żółtowej zgodny z podanymi dyrektywami define

**int param1** – parametr komendy (jeśli komenda nie wymaga parametru ==0)

**int param2** – parametr komendy (jeśli komenda nie wymaga parametru ==0)

**DrawingContextStruct \*dc** – struktura przechowująca kontekst rysowania - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania komendy np.:

```
struct DrawingContextStruct
```

```
{
```

```
unsigned char *bufor – wskazanie na bufor zawierający obraz
```

```
//... i ewentualnie inne potrzebne pola
```

```
}
```

2. Odczyt danych z pliku realizowany jest na poziomie języka C.

3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

4. Składnia wywołania programu: **turtle plik\_komend.txt plik.bmp**

5. Do rysowania linii proszę użyć algorytmu Bresenham'a. Opis można znaleźć w [6].

## 5.8 Odległość Hamminga

Dane są dwa obrazy biało-czarne w dwóch plikach BMP. Zrealizować program, który policzy minimalną odległość Hamminga między tymi obrazami. Odległość Hamminga jest liczbą pikseli, na których różnią się obrazy, z uwzględnieniem możliwego przesunięcia w osi poziomej i pionowej obrazów względem siebie. Zakłada się, że przesunięcie zawiera się w przedziale  $\langle -7, 7 \rangle$  pikseli.

Implementowany algorytm powinien, dla każdego możliwego przesunięcia obrazów względem siebie, wyznaczyć liczbę pikseli, na których różnią się obrazy i wybrać wartość minimalną, jako wynik końcowy.

Program powinien być napisany w języku C/C++ z wybranymi funkcjami zaimplementowanymi w asemblerze procesora Intel.

### Wejście

Dane wejściowe dla programu znajdują się w plikach w formacie BMP (1-bitowych) [5] o nazwie **obraz1.bmp** oraz **obraz2.bmp**. Obrazy są kwadratowe (i tych samych rozmiarów).

### Wyjście

Wynikiem działania programu powinny być dwa pliki tekstowe o nazwach **hamming.txt** i **tablica.txt**.

Plik **hamming.txt** zawiera minimalną odległość Hamminga między obrazami dla dowolnego, dopuszczalnego przesunięcia w obu osiach.

Plik **tablica.txt** zawiera tabelę odległości Hamminga dla wszystkich możliwych przesunięć.

### Uwagi

1. Dozwolone jest korzystanie z instrukcji operujących tylko na liczbach całkowitych (wykorzystywanie operacji zmiennoprzecinkowych jest niedozwolone).
2. W programie należy zaimplementować w asemblerze Intela następującą funkcję:

#### Funkcja obliczająca odległość Hamminga

```
int Hamming(unsigned char *image1, unsigned char *image2,  
            int size, int dx, int dy);
```

Wartość zwracana: odległość Hamminga.

Parametry:

- **unsigned char \*image1** – wskazanie na bufor zawierający pierwszy obraz (bez nagłówków BMP)
- **unsigned char \*image2** – wskazanie na bufor zawierający drugi obraz (bez nagłówków BMP)
- **int size** – rozmiar boku obrazu w pikselach (obraz jest kwadratowy)
- **int dx** – przesunięcie w poziomie:  $\langle -7, 7 \rangle$  pikseli
- **int dy** – przesunięcie w pionie:  $\langle -7, 7 \rangle$  pikseli.



**Funkcja obliczająca minimalną odległość Hamminga**

```
int MinHamming(unsigned char *image1, unsigned char *image2,  
               int size, int *table);
```

Wartość zwracana: odległość Hamminga.

Parametry:

- **unsigned char \*image1** – wskazanie na bufor zawierający pierwszy obraz (bez nagłówków BMP)
- **unsigned char \*image2** – wskazanie na bufor zawierający drugi obraz (bez nagłówków BMP)
- **int size** – rozmiar boku obrazu w pikselach (obraz jest kwadratowy)
- **int \*table** – wskazanie na tablicę zawierającą odległości Hamminga dla wszystkich możliwych przesunięć.

3. Odczyt i zapis danych z plików realizowany jest na poziomie języka C.
4. Alokacja pamięci na bufory realizowana jest na poziomie języka C.

## 5.9 Kod kreskowy Code 128

Zrealizować program dekodujący wybrany podzbiór kodów kreskowych zgodnych ze specyfikacją Code 128.

### Opis kodów kreskowych Code 128

Struktura kodu kreskowego Code 128 została opisana w rozdziale 3.9.

### Wejście

Dane wejściowe dla programu znajdują się w 24 bitowym pliku BMP[5] bez kompresji. Pliki nie spełniające w/w wymagania nie podlegają przetwarzaniu.

Kreski narysowane są równolegle do pionowej krawędzi kolorem czarnym, tło ma kolor biały. W obrazie nie występują żadne zniekształcenia (np. przekos, niedokładności druku itp.).

### Wyjście

Program powinien wypisywać wyniki dekodowania kodu kreskowego na standardowym wyjściu.

### Warianty zadania:

1. Program dekoduje znaki z zestawu Code Set A
2. Program dekoduje znaki z zestawu Code Set B
3. Program dekoduje znaki z zestawu Code Set C
4. Program dekoduje znaki z dowolnego zestawu.

### Uwagi:

1. W programie należy zaimplementować w assemblerze Intel'a następującą funkcję:

```
int Decode128(unsigned char *image,
              char *text, int xsize, int ysize,
              int scanline);
```

Wartość zwracana: 0 w przypadku powodzenia lub kod błędu w przeciwnym przypadku.

Parametry:

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówek BMP)
- **char \*text** – wskazanie na bufor zawierający wyniki dekodowania
- **int xsize** – rozmiar poziomy obrazu w pikselach
- **int ysize** – rozmiar pionowy obrazu w pikselach
- **int scanline** – numer linii w obrazie, w której dokonywane jest dekodowanie.

2. Odczyt i zapis danych z plików realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufor realizowana jest na poziomie języka C.
4. Składnia wywołania programu: **decode128 plik.bmp nr\_linii\_w\_obrazie**
5. Rozpoznawanie kodów kreskowych on-line:  
<http://online-barcode-reader.inliteresearch.com/default.aspx>
6. Generowanie kodów kreskowych on-line: <http://www.barcode-generator.org/>

## 5.10 Kod kreskowy Code 39

Zrealizować program dekodujący kod kreskowy zgodny ze specyfikacją Code 39.

### Opis kodów kreskowych Code 39

Struktura kodu kreskowego Code 39 została opisana w rozdziale 3.10.

### Wejście

Dane wejściowe dla programu znajdują się w 24 bitowym pliku BMP[5] bez kompresji. Pliki nie spełniające w/w wymagania nie podlegają przetwarzaniu.

Kreski narysowane są równoległe do pionowej krawędzi kolorem czarnym, tło ma kolor biały. W obrazie nie występują żadne zniekształcenia (np. przekos, niedokładności druku itp.).

### Wyjście

Program powinien wypisywać wyniki dekodowania kodu kreskowego na standardowym wyjściu.

### Uwagi:

1. W programie należy zaimplementować w assemblerze Intela następującą funkcję:

```
int Decode39(unsigned char *image,
             char *text, int xsize, int ysize,
             int scanline);
```

Wartość zwracana: 0 w przypadku powodzenia lub kod błędu w przeciwnym przypadku.

Parametry:

- **unsigned char \*image** – wskazanie na bufor zawierający obraz (bez nagłówków BMP)
- **char \*text** – wskazanie na bufor zawierający wyniki dekodowania
- **int xsize** – rozmiar poziomy obrazu w pikselach
- **int ysize** – rozmiar pionowy obrazu w pikselach
- **int scanline** – numer linii w obrazie, w której dokonywane jest dekodowanie.

2. Odczyt i zapis danych z plików realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
4. Składnia wywołania programu: **decode39 plik.bmp nr\_linii\_w\_obrazie**
5. Rozpoznawanie kodów kreskowych on-line:  
<http://online-barcode-reader.inliteresearch.com/default.aspx>
6. Generowanie kodów kreskowych on-line:  
<http://generator.onbarcode.com/online-code-39-barcode-generator.aspx>

## 6 Literatura

- [1] **“MARS - Mips Assembly and Runtime Simulator”**,  
<http://courses.missouristate.edu/kenvollmar/mars/Help/MarsHelpIntro.html>
- [2] **“SPIM From Wikipedia, the free encyclopedia”**, <http://en.wikipedia.org/wiki/SPIM>
- [3] **“Lossless Data Compression”**, <http://www.data-compression.com/lossless.html>
- [4] W. Muła, **„Statyczne kodowanie Huffmana”**, <http://wm.ite.pl/articles/huffman.html>
- [5] **“Struktura pliku \*.BMP”**,  
[http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp\\_file\\_format.html](http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_file_format.html)
- [6] **„Bresenham's Line-Drawing Algorithm Explained”**,  
<http://www.falloutsoftware.com/tutorials/dd/dd4.htm>
- [7] **„Bufor Z”**, [http://pl.wikipedia.org/wiki/Bufor\\_Z](http://pl.wikipedia.org/wiki/Bufor_Z)
- [8] **„Lighting and Shading”**,  
<http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/LightingAndShading.html>
- [9] **“Cieniowanie Gourauda”**, [http://pl.wikipedia.org/wiki/Cieniowanie\\_Gourauda](http://pl.wikipedia.org/wiki/Cieniowanie_Gourauda)
- [10] D. Aleksandrow, S. Czaplicki, **„Diagnostyka elektrokardiograficzna”**, Państwowy Zakład Wydawnictw Lekarskich
- [11] W.P. Holsinger, **„A QRS preprocessor based on digital differentiation”**, IEEE Transactions on biomedical engineering, vol. BME-18, 1971, pp. 212-217
- [12] T. Sale, **“Komputery Colossus”**, [http://edu.i-lo.tarnow.pl/inf/hist/006\\_col/0001.php](http://edu.i-lo.tarnow.pl/inf/hist/006_col/0001.php)
- [13] L. Allen, **“Code 128 specification”**, <http://www.barcodeman.com/info/c128.php>
- [14] R. Drollinger, M. Jennings, R. Stewart, **“Bar code Encoding of Strings ”**, <http://courses.cs.washington.edu/courses/cse370/01au/minirproject/theBarCoders/barcodes.html>
- [15] L. Allen, **“Code 39 specification”**, [http://www.barcodeman.com/info/c39\\_1.php](http://www.barcodeman.com/info/c39_1.php)

Najaktualniejsza wersja niniejszego opracowania wraz z plikami przykładowymi jest do pobrania ze strony:

**<http://galera.ii.pw.edu.pl/~zsz/arko>**