



Architektura komputerów
Materiały pomocnicze do laboratorium

Zbigniew Szymański

marzec 2021

Instytut Informatyki

Architektura komputerów

Materiały pomocnicze do laboratorium

Zbigniew Szymański <z.szymanski@ii.pw.edu.pl>

SPIS TREŚCI

1	ŚRODOWISKO SYMULATORA MARS	4
1.1	OMÓWIENIE PROGRAMU	5
1.2	ŚLEDZENIE WYKONANIA PROGRAMU	6
1.3	MODYFIKACJA PROGRAMU	8
2	MAŁE PROGRAMY DLA PROCESORA MIPS	10
3	PROJEKTY DLA PROCESORA MIPS	13
3.1	KOMPRESJA DANYCH	14
3.2	CO WIDZI ROBOT	16
3.3	Z-BUFOR	18
3.4	EKG	20
3.5	MINI ENIGMA	22
3.6	MAPA WYSOKOŚCI	25
3.7	GRAFIKA ŻÓŁWIOWA	27
3.8	ODLEGŁOŚĆ HAMMINGA	28
3.9	KOD KRESKOWY CODE 128	29
3.10	KOD KRESKOWY CODE 39	32
3.11	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 1	34
3.12	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 2	37
3.13	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 3	40
3.14	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 4	43
3.15	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 5	46
3.16	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 6	49
3.17	ZNAJDŹ ZNACZNIKI	52
3.18	UWAGI DOTYCZĄCE PROJEKTÓW DLA PROCESORA MIPS	54
3.19	STRUKTURA PLIKU BMP	55
4	LINUX, NASM, GCC	57
4.1	OPROGRAMOWANIE DLA SYSTEMU WINDOWS	57
4.2	PODSTAWOWE KOMENDY SYSTEMU LINUX	61
4.3	PRZYKŁADOWY PROGRAM	66
4.4	PRZYKŁADOWA SESJA URUCHOMIENIOWA GDB	71
5	MAŁE PROGRAMY DLA PROCESORA INTEL	74
6	PROJEKTY DLA PROCESORA INTEL	75
6.1	KOMPRESJA DANYCH	76
6.2	CO WIDZI ROBOT	77
6.3	Z-BUFOR	80
6.4	EKG	82
6.5	MINI ENIGMA	84
6.6	MAPA WYSOKOŚCI	86
6.7	GRAFIKA ŻÓŁWIOWA	89
6.8	ODLEGŁOŚĆ HAMMINGA	91
6.9	KOD KRESKOWY CODE 128	93

6.10	KOD KRESKOWY CODE 39	94
6.11	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 1	95
6.12	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 2	97
6.13	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 3	99
6.14	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 4	101
6.15	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 5	103
6.16	BINARNA GRAFIKA ŻÓŁWIOWA – WERSJA 6	105
6.17	ZNAJDŹ ZNACZNIKI.....	107
7	LITERATURA	108

1 Środowisko symulatora MARS

Program Mars jest zintegrowanym środowiskiem programistycznym umożliwiającym tworzenie programów w assemblerze 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 assemblerze procesora MIPS.

```
#-----
#autor: Zbigniew Szymanski
#data : 2019.11.01
#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
                           #buffer length
        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 Omówienie programu

W programie występują dwie sekcje rozpoczynające się w kodzie źródłowym dyrektywami `.data` i `.text`. Pierwsza sekcja zawiera dane statyczne, druga – kod programu. Etykiety (zaczynające się na początku wiersza i kończące się znakiem `:`) służą do nadania nazwy lokalizacji pamięci występującej bezpośrednio za etykietą. W przykładowym programie są to:

- `input:`, `prompt:`, `msg1:` - skojarzone z obszarami pamięci przechowującymi zmienne statyczne;
- `main:`, `exit:` - skojarzone z adresami instrukcji. Etykieta `main` ma szczególne znaczenie, ponieważ stanowi punkt wejścia do programu. Od skojarzonej z nią instrukcji rozpoczyna się jego wykonywanie.

Programista może dzięki etykietom używać nazw czytelnych dla człowieka, które zostaną zamienione na adresy na etapie asemblacji programu.

Dyrektywa `.space` służy do zadeklarowania danej bez nadania wartości początkowej. W przykładowym programie rezerwowane jest 80 bajtów na bufor wczytywanych danych. Dyrektywa `.ascii` służy do zadeklarowania łańcucha znakowego zakończonego kodem o wartości zero, któremu nadawana jest wartość początkowa. W przykładzie deklarowane są dwa łańcuchy znakowe zawierające komunikaty wyświetlane w konsoli.

Trzy instrukcje za etykietą `main` służą do wyświetlenia w konsoli komunikatu skojarzonego z łańcuchem znakowym `prompt`. Wykorzystywane jest wywołanie funkcji systemowej `print_string` implementowanej przez symulator Mars. Tabela wszystkich funkcji systemowych jest dostępna po wybraniu menu `Help`, następnie pozycji `Help` i zakładki `Syscalls` w oknie pomocy. Wywołanie funkcji systemowych następuje przy pomocy instrukcji `syscall` (jest to bezargumentowa instrukcja pułapki). Przed jej użyciem należy w rejestrze `$v0` umieścić numer wywoływanej funkcji systemowej, zaś argumenty tej funkcji w rejestrach `$a_`.

Instrukcja `li $v0, 4` ładuje stałą natychmiastową (której wartość umieszczona jest w kodzie programu) o wartości 4 do rejestru `$v0`. Nazwa instrukcji `li` jest skrótem od słów `load immediate`.

Instrukcja `la $a0, prompt` ładuje adres danej (skojarzony z etykietą `prompt`) do rejestru `$a0`. Nazwa instrukcji `la` jest skrótem od słów `load address`. Warto zwrócić uwagę, że ładowanie i składowanie danych w pamięci może być wykonywane wyłącznie przez instrukcje z grup instrukcji `load` oraz `store`.

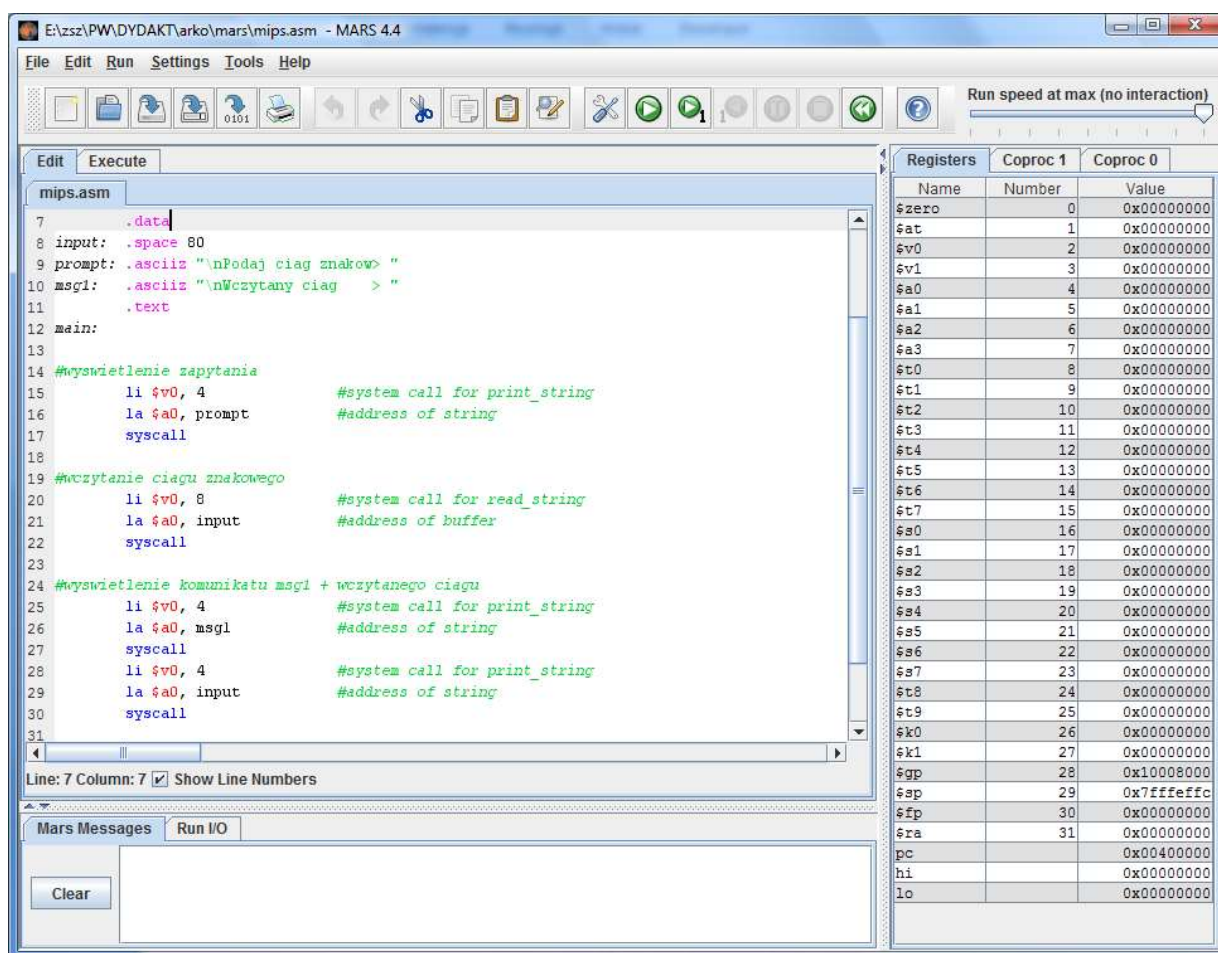
Znak `#` oznacza rozpoczęcie komentarza – tekst za znakiem do końca linii jest ignorowany.

Kolejne trzy instrukcje służą do wywołania funkcji systemowej `read_string` wczytującej z klawiatury ciąg znaków. Następnie wywoływana jest dwukrotnie funkcja systemowa `print_string` – za pierwszym razem wyświetlany jest komunikat skojarzony z etykietą `msg1`, zaś za drugim

razem zawartość bufora skojarzonego z etykietą `input`. Działanie programu kończy się wywołaniem funkcji systemowej `exit`.

1.2 Śledzenie wykonania programu

Na rys. 1.1 przedstawiono okno programu Mars po wczytaniu przykładowego programu (menu `File | Open`). Centralną część okna zajmuje kod programu asemblerowego (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.

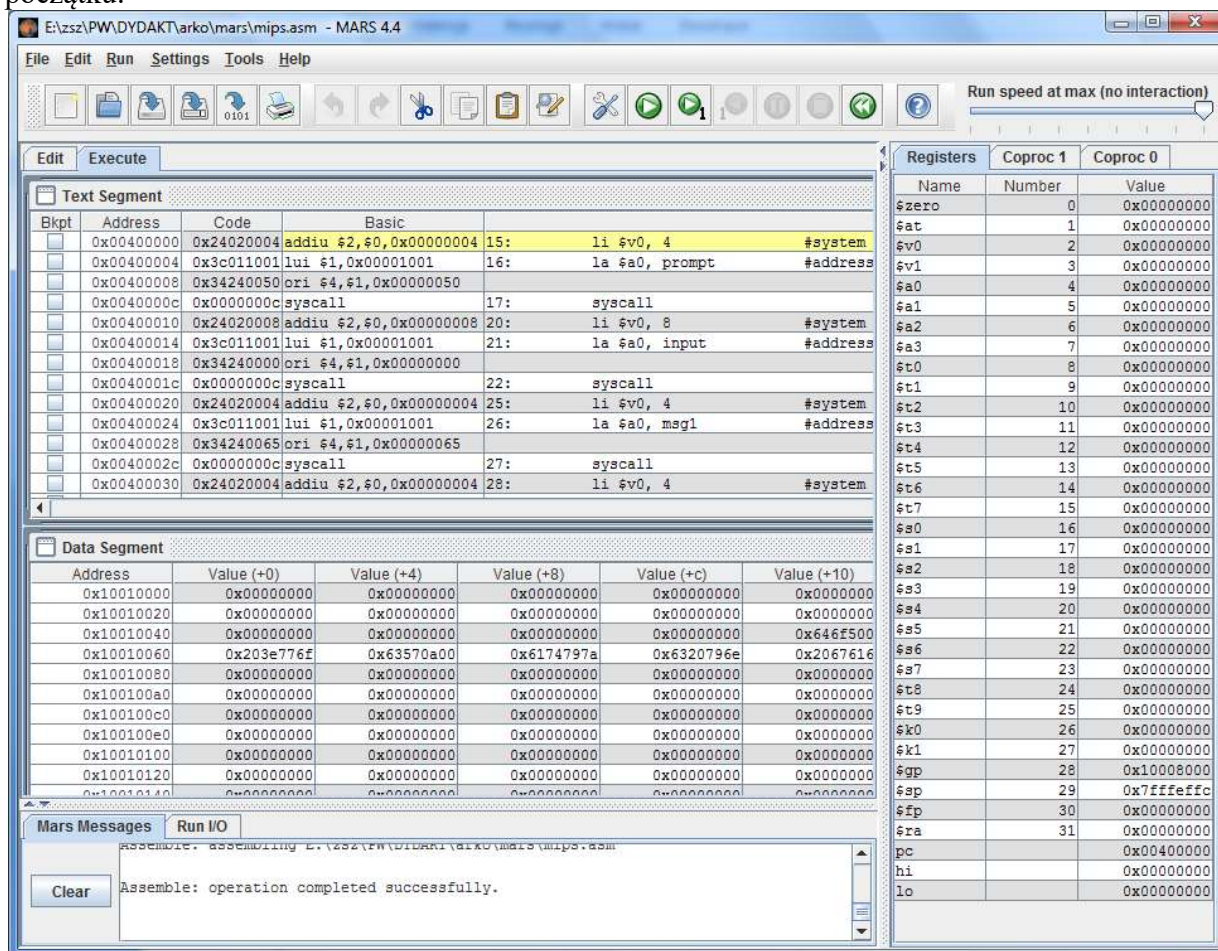


Rys. 1.1 Środowisko Mars – widok edycji kodu asemblerowego

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_{dec}. 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 $a1, 80     #buffer length
    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.

1.3 Modyfikacja programu

Program z listingu 1.1 zostanie teraz rozbudowany w taki sposób, aby liczona była długość wczytanego z klawiatury łańcucha znakowego. W przypadku kodowania w assemblerze (dowolnego procesora) warto najpierw napisać (pseudo)kod w języku C i dopiero na tej podstawie pisać instrukcje assemblerowe. Na listingu 1.3 przedstawiono kod programu do określania długości łańcucha znakowego. Czytelnik powinien samodzielnie dokonać jego analizy.

Listing 1.3. Kod w języku C do określania długości łańcucha znakowego

```
char *string;
int count;

count=0;
string=input;
while (*string!='\0'){
    string++;
    count++;
}
```

Kod assemblerowy utworzony na podstawie programu w języku C pokazano na listingu 1.4. Pokazane instrukcje należy umieścić w przykładowym programie tuż przed etykietą `exit`.

Listing 1.4. Kod asemblerowy do określania długości łańcucha znakowego

```

        li $t1,0           #count=0;
        la $t2,input       #string=input;
loop:
        lbu $t3,($t2)      #while (*string!='\0'){
        beqz $t3,loop_exit
        addi $t2,$t2,1      #    string++;
        addi $t1,$t1,1      #    count++;
        j loop              #    }
loop_exit:
        li $v0,1           #print string length
        move $a0,$t1
        syscall
exit:   li $v0,10           #Terminate
        syscall

```

Przed napisaniem kodu z listingu 1.4 podjęto następujące decyzje projektowe:

- Liczba znaków w łańcuchu znakowym (zmienna count) będzie przechowywana w rejestrze \$t1;
- Adres bieżącego znaku (zmienna string) będzie przechowywany w rejestrze \$t2;
- Wartość bieżącego znaku (wczytanego z pamięci) będzie przechowywana w rejestrze \$t3.

Instrukcja **li \$t1,0** inicjuje zmienną count – ładuje do rejestru \$t1 wartość natychmiastową 0. Instrukcja **la \$t2,input** inicjuje zmienną string – ładuje do rejestru \$t2 adres bufora skojarzonego z etykietą input.

Instrukcja **lbu \$t3,(\$t2)** jest pierwszą instrukcją pętli while. Ładuje jeden bajt (jeden znak ciągu znakowego) z pamięci pod adresem zawartym w rejestrze \$t2 i umieszcza go w rejestrze \$t3. Nawiasy okrągłe wokół rejestru \$t2 oznaczają użycie rejestrowego pośredniego trybu adresowania – argument operacji w takim przypadku znajduje się w pamięci, a w rejestrze jego adres.

Instrukcja **beqz \$t3,loop_exit** jest instrukcją skoku warunkowego (nazwa jest skrótem od *branch if equal zero*). Skok jest wykonywany do etykiety loop_exit jeśli rejestr \$t3 ma wartość zero (oznacza to, że osiągnięto koniec łańcucha znakowego). W przeciwnym przypadku program przechodzi do kolejnej instrukcji.

Instrukcja **addi** dodaje wartość trzeciego argumentu (natychmiastowego) do wartości drugiego argumentu (znajdującego się w rejestrze), a wynik umieszcza w rejestrze będącym pierwszym argumentem. W programie wykorzystano instrukcję addi do inkrementacji licznika znaków i adresu bieżącego znaku.

Instrukcja **j loop** jest instrukcją bezwarunkowego skoku do etykiety loop, gdzie badany jest warunek pętli while.

Za pętlą znajduje się wywołanie funkcji systemowej *print_integer*, której zadaniem jest wyświetlenie długości ciągu znakowego. Do rejestru \$v0 ładowana jest wartość 1 (numer funkcji systemowej), zaś do rejestru \$a0 (przechowującego argument funkcji *print_integer*) kopiowana jest wartość z rejestru \$t1 (zawierającego długość ciągu znakowego) przy pomocy instrukcji move. Instrukcja **syscall** powoduje wykonanie funkcji systemowej.

Po uruchomieniu programu widać, że długość ciągu znakowego jest dłuższa od oczekiwanej. Jest to spowodowane tym, że funkcja systemowa *read_string* na końcu łańcucha znakowego umieszcza również bajty reprezentujące w kodzie ASCII koniec wiersza. Korektę programu pozostawiam czytelnikowi.

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

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-1-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. Podaj ciąg znakow> tel. 12-00-55 Wynik > 6</p>
5a	<p>Zamienić wszystkie znaki występujące od pierwszego w ciągu znaku > do pierwszego następującego po nim znaku < na *. Podaj ciąg znakow> Ala >ma< kota. Ciąg po konwersji> Ala >***< kota.</p>
5b	<p>Zamienić wszystkie znaki występujące przed pierwszym znakiem > i po pierwszym następującym po nim znaku < na *. Podaj ciąg znakow> Ala >ma< kota. Ciąg po konwersji> ****>ma<*****</p>
5c	<p>Przenieść wszystkie znaki występujące przed pierwszym znakiem ! na koniec ciągu Podaj ciąg znakow> Ala !ma kota Ciąg po konwersji> !ma kota Ala</p>
6a	<p>Policzyć liczbę grup cyfr występujących w ciągu znakowym. Podaj ciąg znakow> tel. 12-00-55 Wynik > 3</p>
6b	<p>Policzyć liczbę grup liter (rozdzielonych nie literami) występujących w ciągu znakowym. Podaj ciąg znakow> Ala ma 1 kota. Wynik > 3</p>
7a	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków pomiędzy pierwszym wystąpieniem znacznika początkowego i pierwszym wystąpieniem znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje. Input string > oh:wind on the hill Conversion results> wind *****e hill</p>
7b	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków pomiędzy ostatnim wystąpieniem znacznika początkowego i ostatnim wystąpieniem znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje. Input string > nl:wind on the hill Conversion results> wind o*****</p>
7c	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków przed pierwszym wystąpieniem znacznika początkowego i po pierwszym wystąpieniu znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje. Input string > oh:wind on the hill Conversion results> *****on th*****</p>
7d	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków przed ostatnim wystąpieniem znacznika początkowego i po ostatnim wystąpieniu znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje. Input string > nh:wind on the hill Conversion results> *****n the h***</p>

7e	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków pomiędzy pierwszym wystąpieniem znacznika początkowego i ostatnim wystąpieniem znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje.</p> <pre>Input string > oh:wind on the hill Conversion results> wind *****ill</pre>
7f	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków pomiędzy ostatnim wystąpieniem znacznika początkowego i pierwszym wystąpieniem znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje.</p> <pre>Input string > nl:wind on the hill Conversion results> wind o*****l</pre>
7g	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków przed pierwszym wystąpieniem znacznika początkowego i po ostatnim wystąpieniu znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje.</p> <pre>Input string > oh:wind on the hill Conversion results> *****on the h***</pre>
7h	<p>Pierwszy i drugi znak ciągu znakowego są znacznikami (początkowym i końcowym) definiującymi podciąg tego ciągu. Zadanie polega na zamianie wszystkich znaków przed ostatnim wystąpieniem znacznika początkowego i po pierwszym wystąpieniu znacznika końcowego na znak *. Jeśli w ciągu (od czwartego znaku) nie występuje znacznik początkowy lub końcowy to nie należy dokonywać zmian. Pierwsze trzy znaki ciągu wejściowego należy zamienić na spacje.</p> <pre>Input string > nh:wind on the hill Conversion results> *****n th*****</pre>

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).

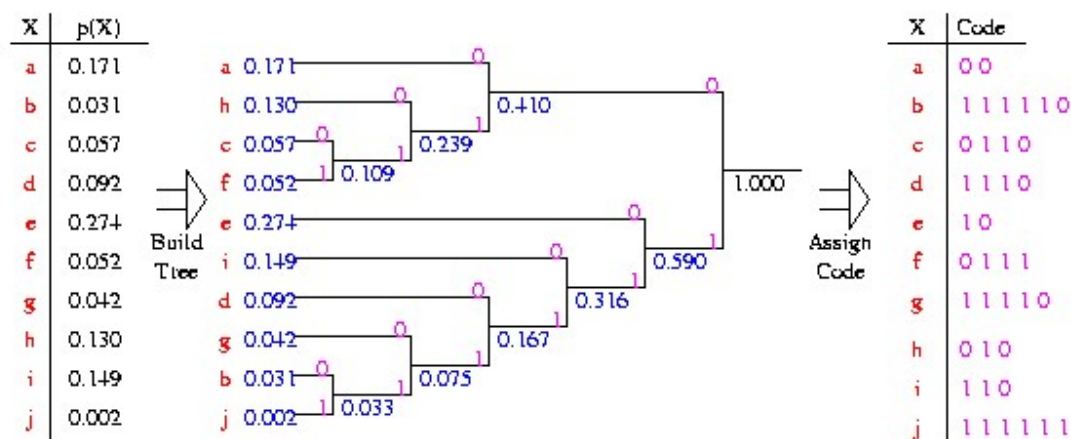
W kroku drugim do połączenia wybrany jest węzeł utworzony w kroku 1 (0.033) i symbol g (0.042). Utworzonemu węzłowi odpowiada prawdopodobieństwo 0.075 . Lewa krawędź od nowo utworzonego węzła do symbolu o wyższym prawdopodobieństwie reprezentuje etykietę 0 (g), a do węzła o mniejszym prawdopodobieństwie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

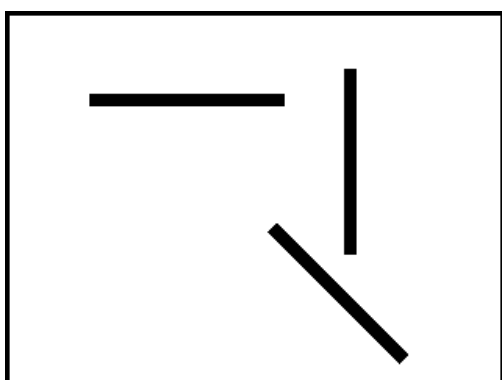
3.2 Co widzi robot

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.

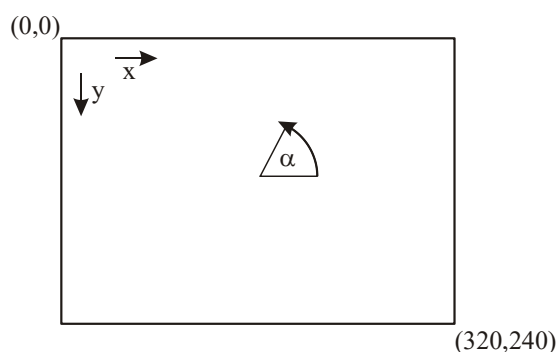
Wejście

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 $\langle 0, 240 \rangle$,
- α - 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:

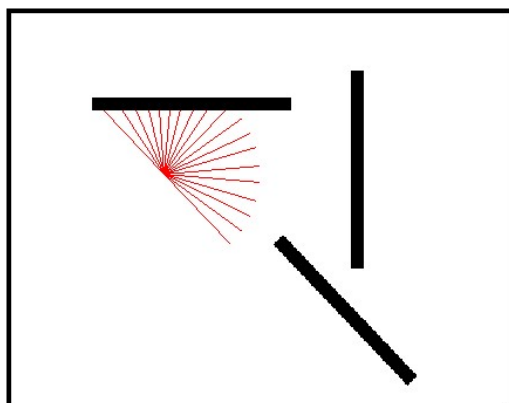
- maksymalna długość wiązki laserowej – 60 pikseli,
- liczba wiązek laserowych – 19. Wiązki są rozmieszczone równomiernie co 10° w zakresie $\langle 90^\circ, -90^\circ \rangle$ od kierunku, w którym skierowany jest robot.

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° (lewa strona robota).



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

Przykładowa zawartość pliku wyniki.txt
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

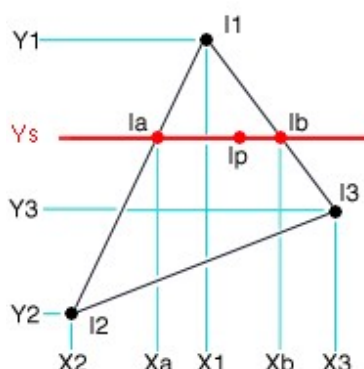
Należy napisać program symulujący działanie mechanizmu Z-bufora [7]. Program ma za zadanie rysować trójkąty, których parametry wczytane są z pliku, zaś efekty działania programu powinny być zapisane do plików BMP. Przy określaniu kolorów pikseli należy użyć metody cieniowania interpolowanego (takiego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

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

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

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

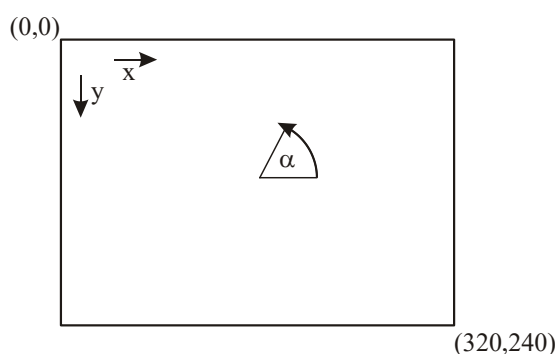
Wejście

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 $<0,255>$.
- 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 $<0,319>$, współrzędnej Y $<0,239>$. Składowa Z może przyjmować wartości z zakresu $<0x0000\ 0000, 0xFFFF\ FFFE>$. Wartości RGB zawierają się w przedziale $<0,255>$.



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**.

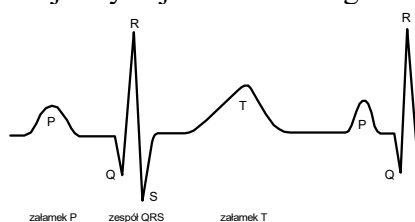
Plik **zbufor.bmp** zawiera wizualizację zawartości Z-bufora w odcieniach szarości. Wartość 0 reprezentowana przez kolor biały, nieskończoność przez kolor czarny.

Warianty zadania:

- Dozwolone jest wykorzystanie dowolnych instrukcji procesora MIPS.
- 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 oznaczają 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. Odbывается 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:

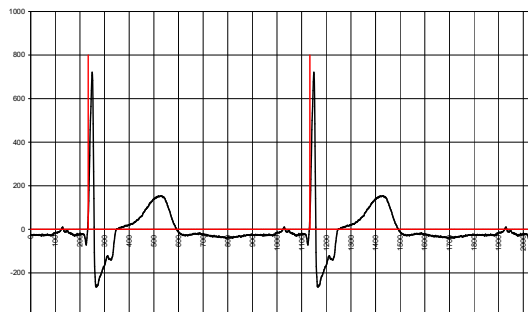
1. Do bufora wejściowego wczytywany jest ciąg próbek $X(n)$ sygnału EKG z jednego odprowadzenia.
2. Liczona jest pierwsza pochodna sygnału według wzoru

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

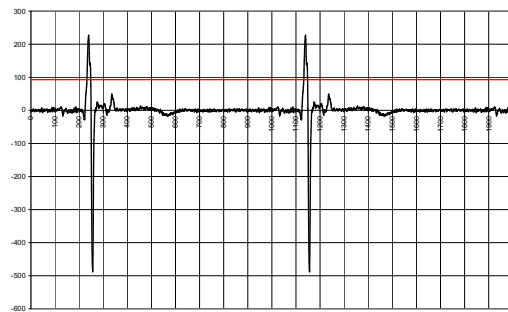
Pochodną oblicza się jako różnicę próbki oddalanej o 1 ms do przodu (4 próbki przy częstotliwości próbkowania 1 kHz) od analizowanej próbki i próbki odległej o 1 ms wstecz.

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/materialy/>

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
...
```

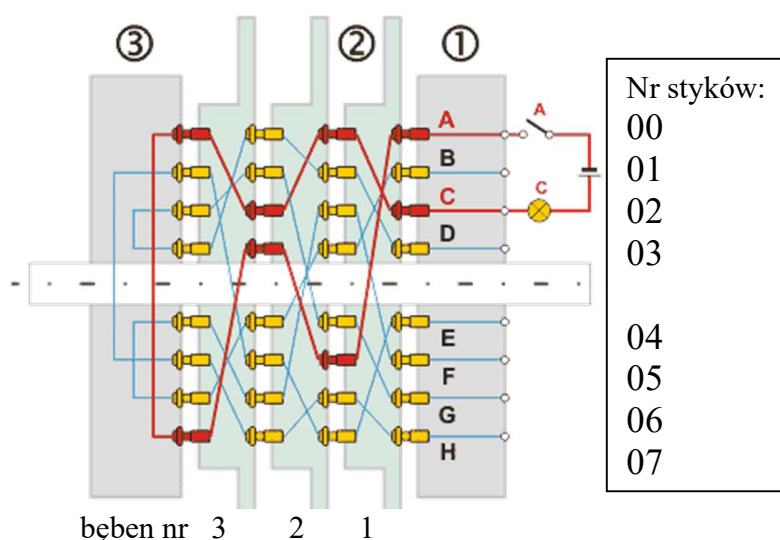
Uwagi

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). Zaszyfrowany znak otrzymujemy po podaniu sygnału na styk wejściowo/wyjściowy o numerze 02 (odpowiadający w przykładzie literze C).

Po zaszyfrowaniu 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_{dec}) jest nie większa niż 1024 znaki.
- tekst jawny zawiera znaki o kodach ASCII z przedziału <32_{dec}, 95_{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_{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_{dec}. 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 $<0,9999>$ 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 $<0,200>$ 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).

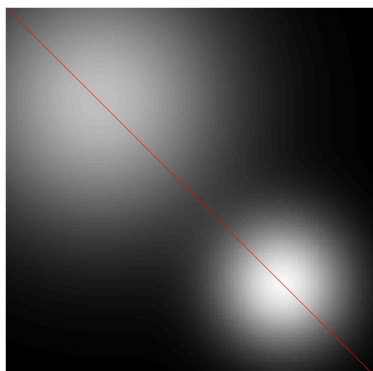
Wyjście

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

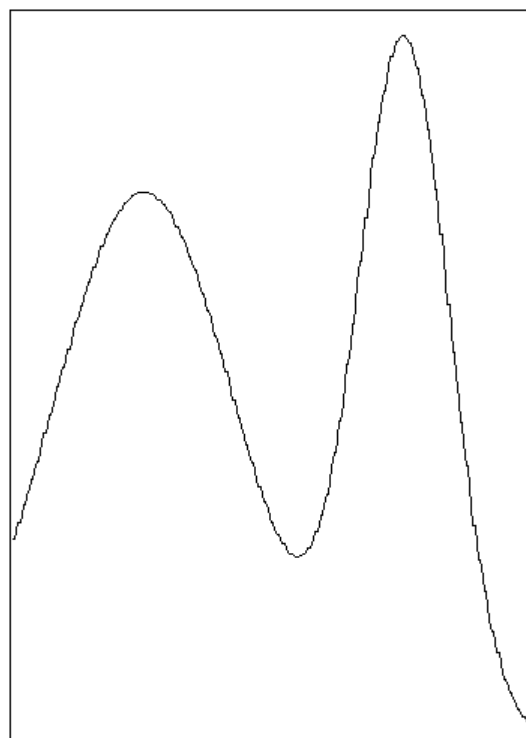
Plik **mapa.bmp** (rys. 1) zawiera obraz mapy wysokości w skali szarości. Obraz ma stałe rozmiary 201x201 pikseli. Wartość 0 z mapy wysokości jest reprezentowana przez kolor czarny. Wartości większe równe 400 reprezentowane są przez kolor biały. Pośrednie wartości reprezentowane są jako odcienie szarości.

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.

$$kod_znaku_kontrolnego = \left(kod_startu + \sum_{i=1}^{liczb\ znaków\ danych} i * kod_znaku[i] \right) mod\ 103$$

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	K	P	K	P	K	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 600 pikseli 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:

1. Przykładowe kody: <http://galera.ii.pw.edu.pl/~zsz/arko/materialy/>
2. Generowanie kodów kreskowych on-line: <http://generator.onbarcode.com/online-code-39-barcode-generator.aspx>
3. Rozpoznawanie kodów kreskowych on-line: <http://online-barcode-reader.inliteresearch.com/default.aspx>
4. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem: http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
5. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

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 kreski i przerwy szerokich do kreski i przerwy wąskich może wynosić od 2.2:1 do 3:1 (w obrębie jednego kodu jest stały). Każdy znak składa się z 9 pól (5 kreski i 4 przerwy) 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 kreski (K) i przerwy (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>	<u>P</u>	<u>K</u>
0	0	1	1	1	2	2	1	2	1	1
1	1	2	1	1	2	1	1	1	1	2
2	2	1	1	2	2	1	1	1	1	2
3	3	2	1	2	2	1	1	1	1	1
4	4	1	1	1	2	2	1	1	1	2
5	5	2	1	1	2	2	1	1	1	1
6	6	1	1	2	2	2	1	1	1	1
7	7	1	1	1	2	1	1	2	1	2
8	8	2	1	1	2	1	1	2	1	1
9	9	1	1	2	2	1	1	2	1	1
10	A	2	1	1	1	1	2	1	1	2
11	B	1	1	2	1	1	2	1	1	2
12	C	2	1	2	1	1	2	1	1	1
13	D	1	1	1	1	2	2	1	1	2
14	E	2	1	1	1	2	2	1	1	1
15	F	1	1	2	1	2	2	1	1	1
16	G	1	1	1	1	1	2	2	1	2
17	H	2	1	1	1	1	2	2	1	1
18	I	1	1	2	1	1	2	2	1	1
19	J	1	1	1	1	2	2	2	1	1
20	K	2	1	1	1	1	1	1	2	2

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

- Przykładowe kody: <http://galera.ii.pw.edu.pl/~zsz/arko/materialy/>
- Generowanie kodów kreskowych on-line: <http://generator.onbarcode.com/online-code-39-barcode-generator.aspx>
- Rozpoznawanie kodów kreskowych on-line: <http://online-barcode-reader.inliteresearch.com/default.aspx>
- Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem: http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
- Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.11 Binarna grafika żółwiowa – wersja 1

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Set position command

Komenda *set position* ustawia nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo definiuje komendę (bity 15-14). Punkt (0,0) znajduje się w lewym dolnym rogu obrazu. Drugie słowo zawiera współrzędne X (bity x9-x0) i Y (bity y5-y0) nowej pozycji.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
0	0	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
y5	y4	y3	y2	y1	y0	x9	x8
bit no. 7	6	5	4	3	2	1	0
x7	x6	x5	x4	x3	x2	x1	x0

Komenda Set direction

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Tabela 3. Komenda *set direction*.

bit no. 15	14	13	12	11	10	9	8
0	1	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	d1	d0

Tabela 4. Opis bitów d1,d0.

Bity d1,d0	Zwrot żółwia
00	Prawo
01	Góra
10	Lewo
11	Dół

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity r3-r0, g3-g0, b3-b0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
1	0	-	-	-	-	m9	m8
bit no. 7	6	5	4	3	2	1	0
m7	m6	m5	m4	m3	m2	m1	m0

Komenda *Set pen state*

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity r3-r0 są najbardziej znaczącymi bitami 8-bitowej czerwonej składowej koloru (pozostałe bity są ustawione na zero). Bity g3-g0 są najbardziej znaczącymi bitami 8-bitowej zielonej składowej koloru (pozostałe bity są ustawione na zero). Bity b3-b0 są najbardziej znaczącymi bitami 8-bitowej niebieskiej składowej koloru (pozostałe bity są ustawione na zero).

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
1	1	Ud	-	b3	b2	b1	b0
bit no. 7	6	5	4	3	2	1	0
g3	g2	g1	g0	r3	r2	r1	r0

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro podniesione (up)
1	Pióro opuszczone (down)

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółwiem
- Nazwa pliku: "input.bin"

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: "output.bmp"

Uwagi:

11. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
12. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.12 Binarna grafika żółwiowa – wersja 2

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Komenda *Set position*

Polecenie *set position* określa nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo określa polecenie (bity 1-0) i współrzędną Y (bity y5-y0) nowej pozycji. Drugie słowo zawiera współrzędną X (bity x9-x0) nowej pozycji. Punkt (0,0) znajduje się w lewym dolnym rogu obrazu.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
y5	y4	y3	y2	y1	y0	1	1

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
-	-	-	-	-	-	x9	x8
bit no. 7	6	5	4	3	2	1	0
x7	x6	x5	x4	x3	x2	x1	x0

Komenda *Set direction*

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Tabela 3. Komenda *set direction*.

bit no. 15	14	13	12	11	10	9	8
d1	d0	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	1	0

Tabela 4. Opis bitów d1, d0.

Bits d1,d0	Zwrot żółwia
00	Prawo
01	Góra
10	Lewo
11	Dół

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity r3-r0, g3-g0, b3-b0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
m9	m8	m7	m6	m5	m4	m3	m2
bit no. 7	6	5	4	3	2	1	0
m1	m0	-	-	-	-	0	1

Komenda *Set pen state*

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity r3-r0 są najbardziej znaczącymi bitami 8-bitowej czerwonej składowej koloru (pozostałe bity są ustawione na zero). Bity g3-g0 są najbardziej znaczącymi bitami 8-bitowej zielonej składowej koloru (pozostałe bity są ustawione na zero). Bity b3-b0 są najbardziej znaczącymi bitami 8-bitowej niebieskiej składowej koloru (pozostałe bity są ustawione na zero).

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
r3	r2	r1	r0	g3	g2	g1	g0
bit no. 7	6	5	4	3	2	1	0
b3	b2	b1	b0	ud	-	0	0

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro podniesione (up)
1	Pióro opuszczone (down)

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółwiem
- Nazwa pliku: "input.bin"

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: "output.bmp"

Uwagi:

1. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
2. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.13 Binarna grafika żółwiowa – wersja 3

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Set position command

Polecenie *set position* określa nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo definiuje komendę (bity 15-14) i współrzędną Y (bity y5-y0) nowej pozycji. Drugie słowo zawiera współrzędną X (bity x9-x0). Punkt (0,0) znajduje się w lewym dolnym rogu obrazu.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
1	1	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
y5	y4	y3	y2	y1	y0	-	-

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
x9	x8	x7	x6	x5	x4	x3	x2
bit no. 7	6	5	4	3	2	1	0
x1	x0	-	-	-	-	-	-

Komenda Set direction

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Tabela 3. Komenda *set direction*.

bit no. 15	14	13	12	11	10	9	8
1	0	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	d1	d0

Tabela 4. Opis bitów d1,d0.

Bity d1,d0	Zwrot żółwia
00	Prawo
01	Góra
10	Lewo
11	Dół

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity c2-c0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
0	1	m9	m8	m7	m6	m5	m4
bit no. 7	6	5	4	3	2	1	0
m3	m2	m1	m0	-	-	-	-

Komenda *Set pen state*

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity c2-c0 wybierają jeden z predefiniowanych kolorów z tabeli kolorów.

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
0	0	ud	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	c2	c1	c0

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro podniesione (up)
1	Pióro opuszczone (down)

Tabela 8. Tabela kolorów.

bity c2,c1,c0	Kolor
000	black
001	red
010	green
011	blue
100	yellow
101	cyan
110	purple
111	white

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółciem
- Nazwa pliku: “input.bin”

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: “output.bmp”

Uwagi:

1. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
2. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.14 Binarna grafika żółwiowa – wersja 4

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Set position command

Komenda *set position* ustawia nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo definiuje komendę (bity 1-0) i współrzędną X (bity x9-x0) nowej pozycji. Drugie słowo zawiera współrzędną Y (bity y5-y0) nowej pozycji. Punkt (0,0) znajduje się w lewym dolnym rogu obrazu.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
x9	x8	x7	x6	x5	x4	x3	x2
bit no. 7	6	5	4	3	2	1	0
x1	x0	-	-	-	-	1	1

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
y5	y4	y3	y2	y1	y0	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Komenda Set direction

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Table 3. The *set direction* command.

bit no. 15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	d1	d0	1	0

Tabela 4. Opis bitów d1,d0.

Bity d1,d0	Zwrot żółwia
00	Prawo
01	Góra
10	Lewo
11	Dół

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity c2-c0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
-	-	-	-	m9	m8	m7	m6
bit no. 7	6	5	4	3	2	1	0
m5	m4	m3	m2	m1	m0	0	1

Komenda *Set pen state*

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity c2-c0 wybierają jeden z predefiniowanych kolorów z tabeli kolorów.

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
c2	c1	c0	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	ud	-	0	0

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro podniesione (up)
1	Pióro opuszczone (down)

Tabela 8. Tabela kolorów.

bity c2,c1,c0	Kolor
000	black
001	red
010	green
011	blue
100	yellow
101	cyan
110	purple
111	white

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółciem
- Nazwa pliku: “input.bin”

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: “output.bmp”

Uwagi:

1. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
2. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.15 Binarna grafika żółwiowa – wersja 5

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Set position command

Polecenie *set position* określa nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo definiuje komendę (bity 15-14). Punkt (0,0) znajduje się w lewym dolnym rogu obrazu. Drugie słowo zawiera współrzędne X (bity x9-x0) i Y (bity y5-y0) nowej pozycji.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
0	1	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
y5	y4	y3	y2	y1	y0	x9	x8
bit no. 7	6	5	4	3	2	1	0
x7	x6	x5	x4	x3	x2	x1	x0

Komenda Set direction

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Tabela 3. Komenda *set direction*.

bit no. 15	14	13	12	11	10	9	8
0	0	-	-	-	-	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	d1	d0

Tabela 4. Opis bitów d1,d0.

Bity d1,d0	Zwrot żółwia
00	Góra
01	Lewo
10	Dół
11	Prawo

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity r3-r0, g3-g0, b3-b0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
1	1	-	-	-	-	m9	m8
bit no. 7	6	5	4	3	2	1	0
m7	m6	m5	m4	m3	m2	m1	m0

Set pen state command

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity r3-r0 są najbardziej znaczącymi bitami 8-bitowej czerwonej składowej koloru (pozostałe bity są ustawione na zero). Bity g3-g0 są najbardziej znaczącymi bitami 8-bitowej zielonej składowej koloru (pozostałe bity są ustawione na zero). Bity b3-b0 są najbardziej znaczącymi bitami 8-bitowej niebieskiej składowej koloru (pozostałe bity są ustawione na zero).

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
1	0	-	ud	b3	b2	b1	b0
bit no. 7	6	5	4	3	2	1	0
g3	g2	g1	g0	r3	r2	r1	r0

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro opuszczone (down)
1	Pióro podniesione (up)

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółwiem
- Nazwa pliku: "input.bin"

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: "output.bmp"

Uwagi:

3. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip

Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.16 Binarna grafika żółwiowa – wersja 6

W grafice komputerowej grafiką żółwiową nazywane jest tworzenie obrazu na płaszczyźnie kartezjańskiej przy użyciu kursora („żółwia”). Żółw ma trzy atrybuty: położenie, orientację (lub kierunek) i pióro. Pióro ma atrybuty: kolor, stan włączony/wyłączony (lub góra/dół) [17].

Żółw porusza się za pomocą poleceń, które odnoszą się do jego własnej pozycji, takich jak „przejdź do przodu o 10 pól” i „skręć w lewo o 90 stopni”. Piórem noszonym przez żółwia można również sterować, włączając go lub ustawiając jego kolor.

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16].

Komendy sterujące żółwiem

Długość wszystkich poleceń żółwia wynosi 16 lub 32 bity. Dwa bity definiują jedno z czterech poleceń (set position, set direction, move, set state). Nieużywane bity we wszystkich poleceniach są oznaczone znakiem -. Nie należy ich brać pod uwagę przy dekodowaniu polecenia.

Set position command

Polecenie *set position* określa nowe współrzędne żółwia. Składa się z dwóch słów. Pierwsze słowo definiuje komendę (bity 15-14) i współrzędną Y (bity y5-y0) nowej pozycji. Drugie słowo zawiera współrzędną X (bity x9-x0). Punkt (0,0) znajduje się w lewym dolnym rogu obrazu.

Tabela 1. Pierwsze słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
1	0	y5	y4	y3	y2	y1	y0
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Tabela 2. Drugie słowo komendy *set position*.

bit no. 15	14	13	12	11	10	9	8
x9	x8	x7	x6	x5	x4	x3	x2
bit no. 7	6	5	4	3	2	1	0
x1	x0	-	-	-	-	-	-

Komenda Set direction

Polecenie *set direction* określa kierunek, w którym żółw będzie się poruszał, gdy wydane zostanie polecenie przesunięcia. Kierunek jest określony przez bity d1, d0.

Tabela 3. Komenda *set direction*.

bit no. 15	14	13	12	11	10	9	8
1	1	-	-	d1	d0	-	-
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Tabela 4. Opis bitów d1,d0.

Bity d1,d0	Zwrot żółwia
00	Góra
01	Lewo
10	Dół
11	Prawo

Komenda *Move*

Polecenie *move* przesuwa żółwia w kierunku określonym przez bity d1-d0. Odległość jest określona przez bity m9-m0. Jeśli punkt docelowy znajduje się poza obszarem rysowania, żółw powinien zatrzymać się na krawędzi rysunku. Nie może opuścić obszaru rysowania. Żółw pozostawia widoczny ślad, gdy pióro jest opuszczone (bit ud). Kolor śladu jest określony przez bity c2-c0.

Tabela 5. Komenda *move*.

bit no. 15	14	13	12	11	10	9	8
0	0	m9	m8	m7	m6	m5	m4
bit no. 7	6	5	4	3	2	1	0
m3	m2	m1	m0	-	-	-	-

Komenda *Set pen state*

Polecenie *set pen state* określa, czy pióro jest podniesione czy opuszczone (bit ud) i kolor śladu. Bity c2-c0 wybierają jeden z predefiniowanych kolorów z tabeli kolorów.

Tabela 6. Komenda *set pen state*.

bit no. 15	14	13	12	11	10	9	8
0	1	-	ud	-	c2	c1	c0
bit no. 7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-

Tabela 7. Opis bitu *ud*.

ud bit	Stan pióra
0	Pióro opuszczone (down)
1	Pióro podniesione (up)

Tabela 8. Tabela kolorów.

bity c2,c1,c0	Kolor
000	black
001	purple
010	cyan
011	yellow
100	blue
101	green
110	red
111	white

Wejście

- Plik binarny zawierający 16-bitowe komendy sterujące żółciem
- Nazwa pliku: “input.bin”

Wyjście

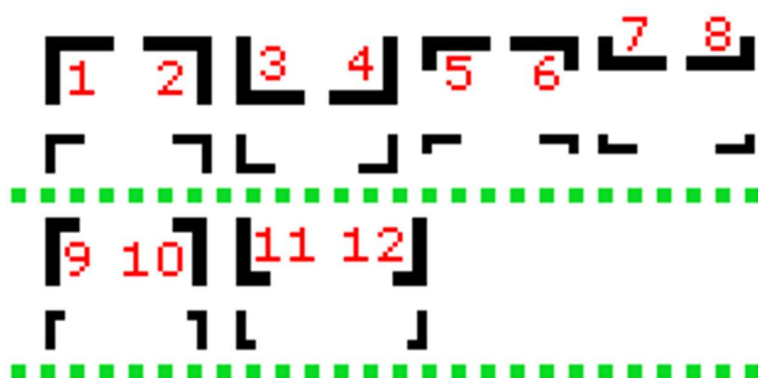
- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 600x50 px,
- Nazwa pliku: “output.bmp”

Uwagi:

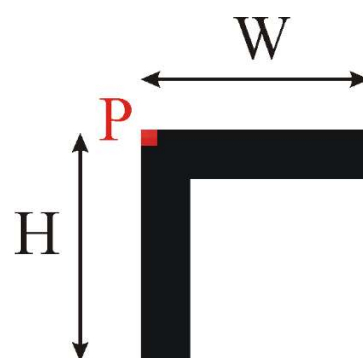
1. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
2. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.17 Znajdź znaczniki

Zadanie polega na napisaniu programu, który w obrazie w formacie BMP [5, 16] wykrywa jeden z dwunastu rodzajów znaczników pokazanych na rys.1.



Rys.1. Przykładowe znaczniki – czerwone numery oznaczają rodzaj znacznika.



Rys.2. Znacznik

Charakterystyka znaczników

Znaczniki są czarne i składają się z dwóch ramion (rys. 2). Dla danego typu znacznika stosunek szerokości W do wysokości H jest stały (patrz tabela 1). Znaczniki danego typu mogą mieć różne rozmiary pod warunkiem zachowania proporcji rozmiarów ramion. Położenie znacznika określa punkt leżący w miejscu przecięcia ramion (oznaczony przez P na rys.2).

Tabela 1. Proporcje rozmiarów znacznika.

Rodzaj znacznika	W/H
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	2
9	$\frac{1}{2}$
10	$\frac{1}{2}$
11	$\frac{1}{2}$
12	$\frac{1}{2}$

Wejście

- Plik BMP zawierający obraz ze znacznikami:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: 320x240 px,
- Nazwa pliku: “input.bmp”

Wyjście

- Terminal – w kolejnych wierszach wyświetlane są współrzędne wykrytych znaczników np. 10, 15. Punkt (0,0) znajduje się w górnym lewym rogu obrazu.

Uwagi:

1. Zakładamy, że na rysunku może wystąpić co najwyżej 50 znaczników danego rodzaju.
2. Przykładowy program odczytujący i zapisujący plik BMP znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/bmp/bmp_mips.zip
3. Przykładowy plik zawierający znaczniki znajduje się pod adresem:
http://galera.ii.pw.edu.pl/~zsz/arko/materialy/find_markers/example_markers.bmp
4. Skrótowy schemat struktury pliku w formacie BMP znajduje się pod koniec rozdziału 3.

3.18 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 udalo 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
    ...

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

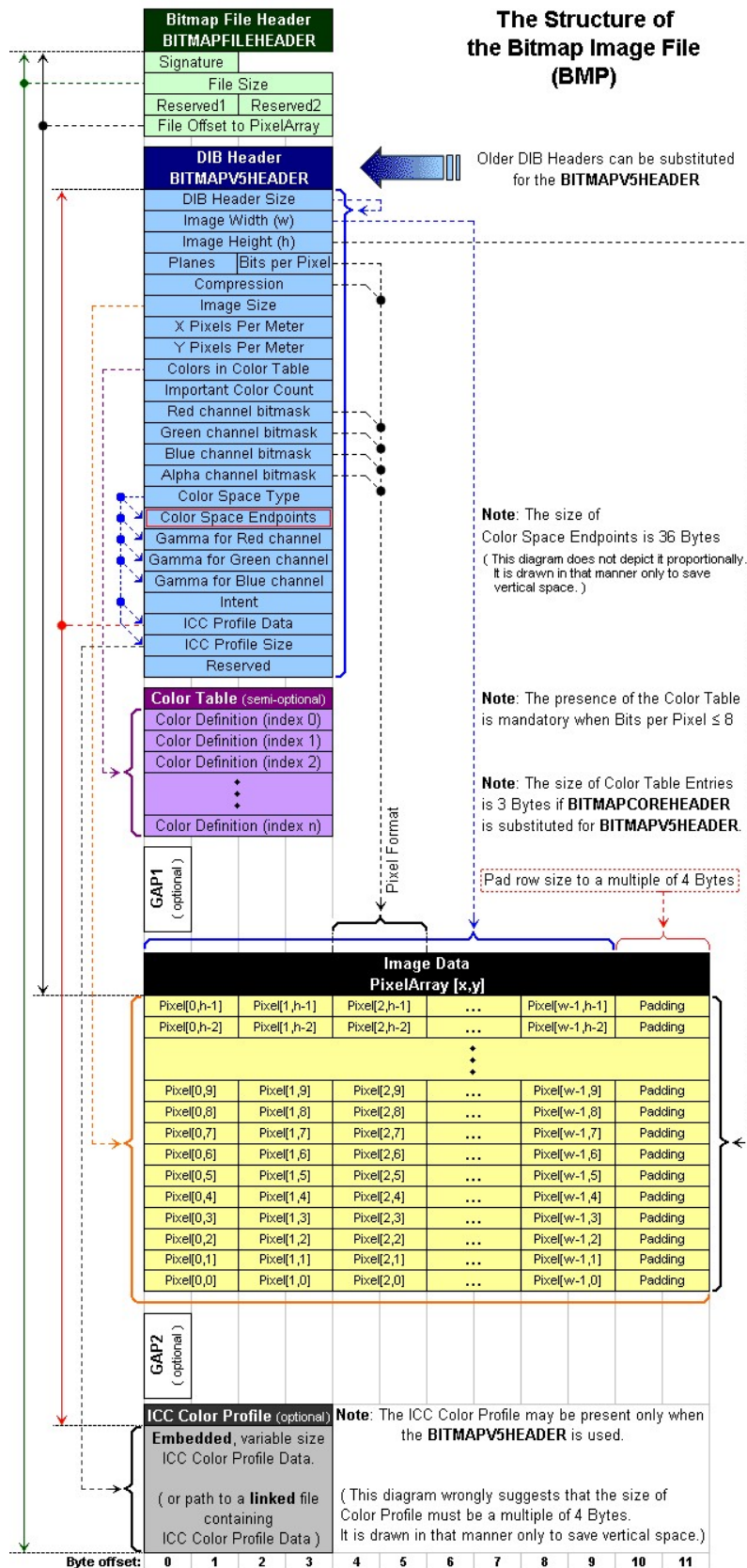
3.19 Struktura pliku BMP

Offset	BMP marker	File size	Reserved	Offset of the pixel data	Header size
00000000	42 4D	9E D2 01 00	00 00 00 00	36 00 00 00	28 00
00000010	00 00	C8 00 00 Width	C7 00 00 Height	01 00 00 Planes	18 00 BPP
00000020	00 00	68 D2 Image size	13 00 X pixels per meter	13 00 Y pixels per meter	00 00 Colors in
00000030	00 00	00 00 Important colors	23 2E Pixel	26 31 Pixel	28 33 Pixel
00000040	33 6C 27 34 6D 29 34 6E	29 34 6F 29 34 6F 26 33			
00000050	71 25 30 6F 25 30 6C 25	30 6B 27 31 6C 2B 35 6D			
00000060	2E 37 70 29 35 6F 25 34	6F 21 31 6D 22 32 6B 23			
00000070	32 69 26 33 6B 25 33 6D	27 35 6D 26 32 6B 25 31			
00000080	6B 26 32 6B 29 35 6D 29	34 6E 25 2F 6B 24 2F 6A			
00000090	24 2F 6B 29 33 6D 2D 37	70 27 32 6F 26 32 6B 26			

Rys. 1. Przykładowa zawartość pliku BMP (wszystkie liczby w notacji szesnastkowej).

Uwagi:

1. Kolejność bajtów - Little endian – pierwszy bajt pola jest najmniej znaczący
2. Kolejność składowych koloru piksela: pierwszy bajt - niebieski, drugi bajt – zielony, trzeci bajt -czerwony.
3. Pole Offset of the pixel data zawiera offset(adres) obszaru zawierającego kolory pikseli



Rys. 2. Struktura pliku BMP [16]

4 Linux, NASM, gcc

W rozdziale zostało opisane środowisko do tworzenia i uruchamiania programów dla procesora Intel x86 wykorzystujące linuksowy serwer Instytutu Informatyki o nazwie `galera.ii.pw.edu.pl`. Przedstawiona przykładowa konfiguracja środowiska pracy zakłada, że użytkownik wykorzystuje lokalnie na swoim komputerze system Windows, gdzie uruchomiony jest edytor kodu, lecz kompilacja projektu i jego uruchomienie odbywa się na zdalnej maszynie z systemem Linux.

4.1 Oprogramowanie dla systemu Windows

Zalecane programy do zainstalowania na systemie Windows to terminal PuTTY, program do transferu plików WinSCP oraz edytor kodu źródłowego Notepad++.

Putty

PuTTY jest programem terminalowym umożliwiającym zdalną pracę w trybie wiersza poleceń na komputerach działających pod kontrolą systemów Unix/Linux. Transmisja danych pomiędzy komputerami jest szyfrowana - wykorzystywany jest protokół SSH (secure shell). Pliki instalacyjne programu Putty można pobrać ze strony WWW:

<https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>

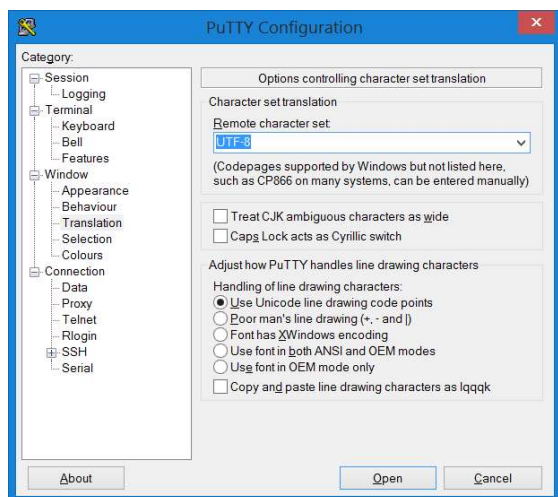
Po uruchomieniu programu proszę upewnić się, że wybrane jest kodowanie znaków UTF-8. Aby to zrobić w panelu *Category* po lewej stronie okna należy wybrać pozycję *Translation* (rys. 4.1). Na liście rozwijanej *Remote character set* powinna być wybrana pozycja *UTF-8*.

W celu nawiązania połączenia ze zdalnym systemem należy w panelu *Category* wybrać pozycję *Session* (rys. 4.2). W polu *Host Name* należy wpisać nazwę/adres komputera, z którym ma być nawiązane połączenie, czyli **`galera.ii.pw.edu.pl`**. Proszę upewnić się, że zaznaczony jest przycisk radiowy *SSH* należący do grupy *Connection type*. Wymienione powyżej ustawienia sesji można zapisać wpisując w polu tekstowym *Saved Sessions* nazwę dla definiowanej sesji np. *galera.ii.pw.edu.pl* i klikając na przycisk *Save*. Zainicjowanie połączenia następuje po kliknięciu na przycisk *Open*.

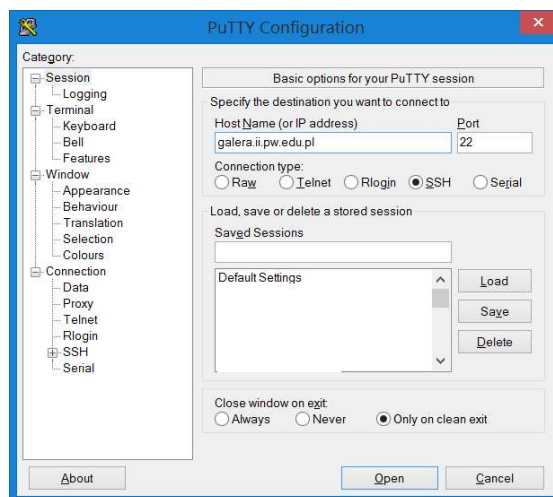
Przy pierwszej próbie podłączenia do danego komputera pojawia się okno PuTTY security alert (Rys. 4.3), należy wówczas kliknąć na przycisk *Yes* potwierdzając, że program PuTTY powinien ufać komputerowi, z którym jest nawiązywane połączenie.

Gdy zostanie wyświetlony komunikat *login as:* należy wprowadzić nazwę użytkownika i nacisnąć *enter*. Gdy pojawi się komunikat *password:* należy wprowadzić hasło (proszę zwrócić

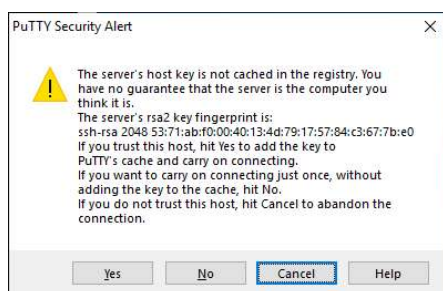
uwagę, że w trakcie wpisywania hasła na ekranie nie będą pojawiać się żadne znaki). Gdy połączenie zostanie nawiązane okno terminala powinno wyglądać jak na rys. 4.4.



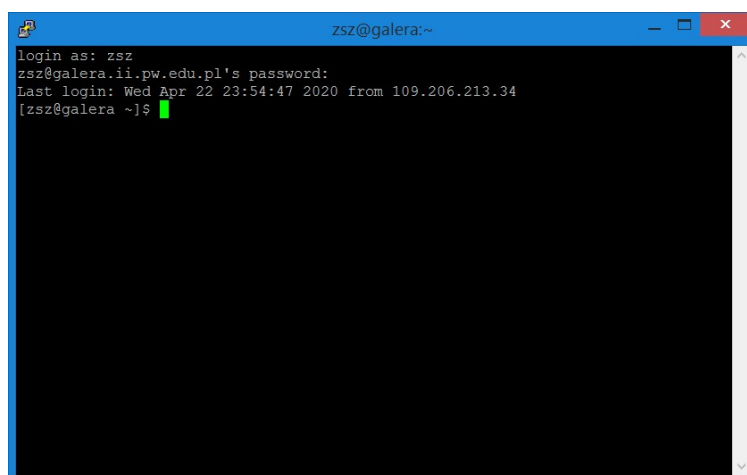
Rys. 4.1. Putty – wybór kodowania znaków.



Rys. 4.2. Putty – parametry sesji.



Rys. 4.3. Okno PuTTY Security Alert



Rys. 4.4. Okno terminala po prawidłowym nawiązaniu połączenia.

Praca z wykorzystaniem wiersza poleceń systemu Linux zostanie skrótowo omówiona w następnych podpunktach.

WinScp

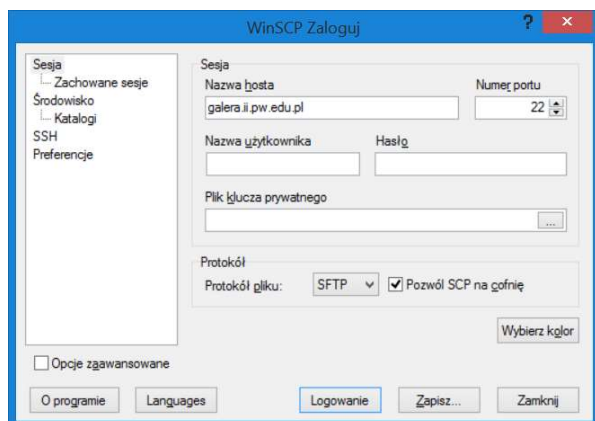
Program WinScp umożliwia szyfrowany transfer plików pomiędzy komputerem z systemem Windows, a komputerem z systemem Unix/Linux. Pliki instalacyjne programu można pobrać ze strony:

<https://winscp.net/eng/download.php>

Po uruchomieniu programu należy w panelu z lewej strony okna wybrać pozycję *Sesja*. Następnie w polu Nazwa hosta wpisać nazwę/adres komputera, z którym ma być nawiązane połączenie, czyli **galera.ii.pw.edu.pl**.

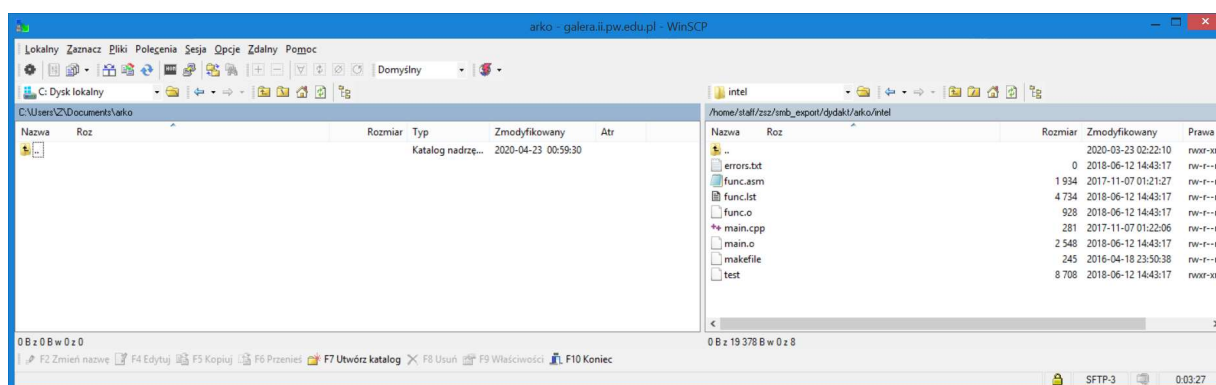
Pola *nazwa użytkownika* i *hasło* mogą pozostać puste. Wówczas użytkownik zostanie poproszony o wprowadzenie tych danych przy nawiązywaniu połączenia. Ustawienia sesji można zapisać

klikając na przycisk *Zapisz...*, a następnie wpisując nazwę dla definiowanej sesji np. *galera.ii.pw.edu.pl*. Zainicjowanie połączenia następuje po kliknięciu na przycisk *Logowanie*.



Rys. 4.5. WinSCP – parametry sesji.

Po nawiązaniu połączenia okno programu WinScp powinno wyglądać podobnie do tego z rys. 4.6. Składa się z dwóch paneli. W lewym wyświetlana jest zawartość folderu z komputera lokalnego, zaś w prawym z komputera zdalnego, z którym nawiązano połączenie. Pliki i foldery można przesyłać przeciągając pomiędzy panelami programu WinSCP lub przeciągając pomiędzy oknami systemowego Eksploratora plików, a panelem komputera zdalnego.



Rys. 4.6. Okno programu WinScp.

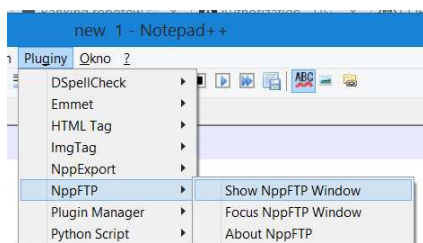
Notepad++ z wtyczką NppFTP

Program *Notepad++* jest edytorem kodu źródłowego obsługującym wyróżnianie elementów składni wielu języków programowania. Wtyczka *NppFTP* umożliwia edycję plików znajdujących się na zdalnych maszynach pracujących pod kontrolą systemów Unix/Linux bez konieczności ich wcześniejszego ściągnięcia np. programem *WinScp* na komputer lokalny. Program można pobrać ze strony:

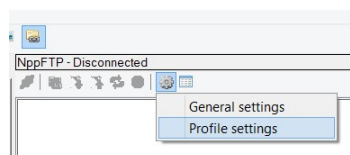
<https://notepad-plus-plus.org/downloads/>

Korzystanie z wtyczki *NppFTP* jest możliwe po skonfigurowaniu parametrów połączenia. Należy najpierw otworzyć okno wtyczki wybierając z menu *Plugins* pozycję *NppFTP*, a następnie *Show NppFTP Window* (rys. 4.7). Polecenie *Profile settings* (rys. 4.8) otwiera okno umożliwiające wpisanie nazwy komputera, do którego należy wykonać połączenie, oraz nazwy użytkownika i hasła. W oknie tym należy kliknąć na przycisk *Add new*. Pojawi się okno *Adding profile*, w którym należy wpisać nazwę profilu (np. *galera.ii.pw.edu.pl*) i nacisnąć *OK*. Następnie w oknie

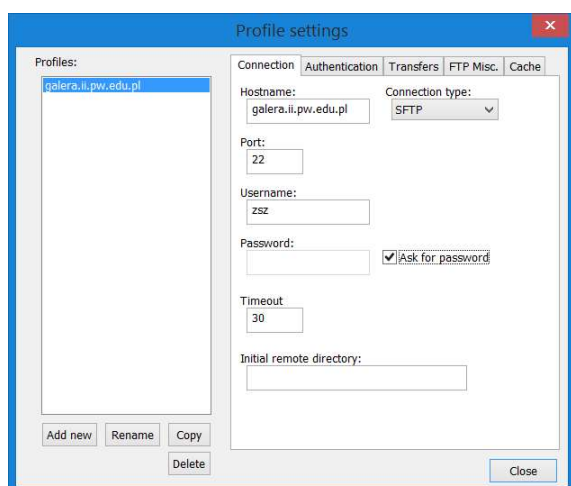
Profile settings (rys. 4.9) należy wpisać w polu *hostname* nazwę komputera, z którym ma być nawiązane połączenie (**galera.ii.pw.edu.pl**), z listy *Connection type* należy wybrać *SFTP*, a w polu *username* wpisać nazwę użytkownika. Zaznaczenie pola *ask for password* spowoduje, że program wyświetli pytanie o hasło przy nawiązywaniu połączenia. Okno ustawień zostanie zamknięte po naciśnięciu przycisku *Close*.



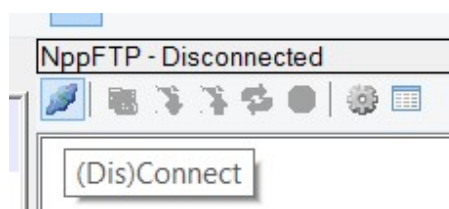
Rys. 4.7. Włączenie okna wtyczki NppFTP.



Rys. 4.8. Ustawienia wtyczki NppFTP.

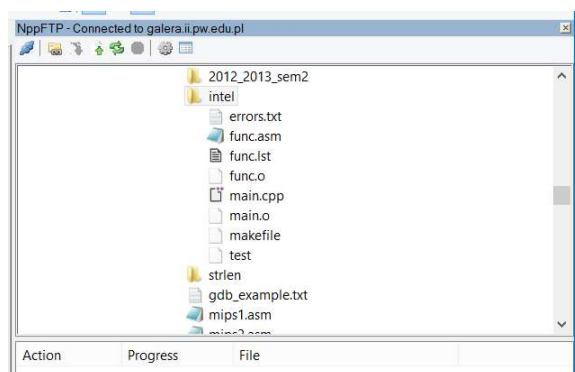


Rys. 4.9. Konfiguracja profilu połączenia.



Rys. 4.10. Przycisk (Dis)Connect.

Nawiązanie połączenia następuje po kliknięciu na przycisku *(Dis)Connect* (rys. 4.10) i wybraniu nazwy profilu połączenia (w powyższym przykładzie jest to *galera.ii.pw.edu.pl*). W oknie wtyczki pojawi się lista plików na zdalnym komputerze (rys. 4.11). Dwukrotne kliknięcie na nazwie pliku spowoduje otwarcie tego pliku w oknie edycji. Dalsza praca z plikiem przebiega tak samo jak z plikiem zapisanym lokalnie na komputerze.



Rys. 4.11. Lista plików komputera zdalnego.

4.2 Podstawowe komendy systemu Linux

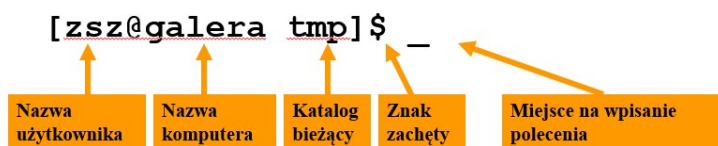
Podczas pracy zdalnej z systemem Linux z wykorzystaniem programu Putty (lub innego klienta protokołu SSH) wykorzystywany jest interfejs wiersza poleceń (ang. command line interface), w którym komendy wpisywane są w formie tekstowej z klawiatury. Każda komenda zatwierdzana jest wciśnięciem klawisza ENTER.

System Linux jest wrażliwy na wielkość liter w nazwach plików. Często powtarzanym błędem przez początkujących użytkowników jest tworzenie plików (np. o nazwie dokument.txt) i następnie odwoływanie się do tych plików np. z własnych programów stosując inną wielkość liter (np. Dokument.txt).

Podstawowe pojęcia i oznaczenia:

- katalog bieżący (ang. current directory) – każda nazwa pliku, która nie jest poprzedzona informacją o katalogu, w którym plik ten się znajduje, odnosi się do pliku w tzw. katalogu bieżącym. Innymi słowy katalog bieżący jest punktem odniesienia przy lokalizowaniu plików w linuksowym systemie plików. Każda sesja terminalowa ma swój katalog bieżący. Katalog bieżący jest oznaczany przez znak kropki „.”.
- katalog nadrzędny (ang. parent directory) – jest to katalog o jeden poziom wyżej w hierarchii systemu plików w stosunku do rozpatrywanego katalogu. Katalog nadrzędny oznaczany jest dwiema kropkami „..”.
- katalog domowy użytkownika (ang. home directory) jest to katalog, wyznaczony przez administratora systemu, do przechowywania plików danego użytkownika. Katalog domowy oznaczany jest znakiem tyldy „~”.
- bezwzględna ścieżka dostępu do pliku (lub katalogu) jednoznacznie identyfikuje lokalizację pliku lub katalogu w systemie plików. W skład ścieżki wchodzi nazwy kolejnych katalogów (rozdzielone znakiem /), począwszy od katalogu głównego, do których trzeba wejść, aby zlokalizować plik. Np. ścieżka **/home/users/xgucio/prog.cpp** oznacza, że w katalogu głównym znajduje się podkatalog home, w nim podkatalog users, w nim podkatalog xgucio, który zawiera plik prog.cpp.

Na rysunku 4.12 pokazano elementy składowe wiersza polecenia. Od lewej strony są to nazwa użytkownika, nazwa komputera, do którego podłączony jest użytkownik (istotna informacji przy pracy zdalnej) oraz ostatni składnik pełnej ścieżki do katalogu bieżącego.



Rys. 4.12. Elementy składowe wiersza polecenia.

Niniejszy podrozdział opisuje kilka podstawowych często używanych komend systemu Linux. Należy zaznaczyć, że omawiane komendy posiadają znacznie więcej opcji niż omówiono to poniżej. Pełen opis działania poleceń linuksowych dostępny w pomocy systemowej uruchamianej komendą: **man nazwa_polecenia**

4.2.1 passwd

Komenda `passwd` umożliwia użytkownikowi zmianę hasła. Inicjalne hasło jest trudne do zapamiętania, bo było wygenerowane przez automat – warto je zmienić. Proszę wymyślić nowe

hasło – nie krótsze niż 6 znaków, zawierające co najmniej jedną wielką literę lub cyfrę lub znak specjalny. Hasło nie może w sobie zawierać nazwy konta (loginu) ani imienia lub nazwiska właściciela.

Hasła do systemów linux obsługiwanych przez laboratorium komputerowe Instytutu Informatyki są przechowywane na serwerze *galera.ii.pw.edu.pl*, więc w celu zmiany hasła należy uruchomić zdalną sesję *ssh* do tego serwera np. przy pomocy programu Putty. Po zalogowaniu się należy uruchomić polecenie *passwd* i dwukrotnie wpisać nowe hasło. W otwartym oknie terminala tekstowego wydajemy poniższe polecenia (rys. 4.13) zaznaczone wytłuszczonym drukiem (UWAGA: zamień przykładowy *jkowalsk* na swój login).

```
[jkowalsk@galera ~]$ passwd      {linuksowe polecenie do zmiany hasła}
New UNIX password:                {podać nowe hasło - nie się nie pojawia, to normalne}
Retype new UNIX password:         {powtórzyć nowe hasło – jak wyżej}
Changing password for user jkowalsk.
passwd: all authentication tokens updated successfully.
[jkowalsk@galera ~]$ exit       {zakończenie zdalnej sesji ssh}
```

Rys. 4.13. Zmiana hasła w systemie Linux.

Zmienione hasło zostanie automatycznie rozpropagowane z serwera *galera* do wszystkich linuksowych komputerów PC w laboratorium (max.opóźnienie: 3 minuty).

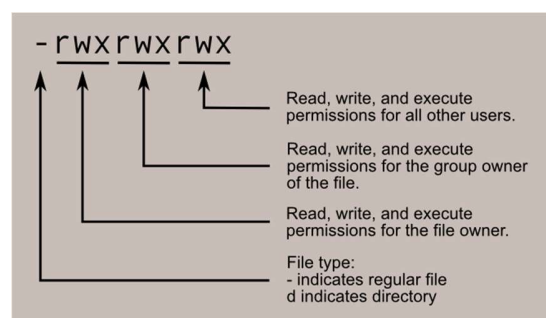
4.2.2 ls

Polecenie **ls** wyświetla listę plików i podkatalogów katalogu bieżącego. Komenda **ls -al** (z dodanym parametrem *-al*, parametry oddzielone są od nazwy polecenia spacją) wyświetla linie w formacie takim jak przykładowa linia na rys. 4.14 .

```
...
-rw-r--r--  1 pi    pi      11531 May 22  2018 sensortagcollector.py
...
```

Rys. 4.14. Przykładowe wyjście komendy *ls*.

Pierwsza kolumna zawierająca **-rw-r--r--** opisuje prawa dostępu do pliku, którego nazwa podana jest w ostatniej kolumnie (na przykładzie z rysunku jest to **sensortagcollector.py**). Użytkownik i grupa (ang. user, group), do których należy plik są opisani przez **pi pi** (użytkownik **pi** i grupa **pi**). Rozmiar pliku wynosi **11531** bajty, a data ostatniej modyfikacji to **May 22 2018**.



Rys. 4.15. Prawa dostępu do plików w systemach typu Unix [18].

Uprawnienia do plików w systemach typu Unix / Linux opisane są na rys. 4.15. Są oddzielne uprawnienia dla właściciela pliku, dla użytkowników należących do jednej wskazanej grupy oraz dla wszystkich pozostałych użytkowników systemu. Dokładny opis uprawnień znajduje się w [18].

4.2.3 pwd

Komenda **pwd** (ang. print working directory) wyświetla pełną ścieżkę bezwzględną do katalogu bieżącego. Tuż po zalogowaniu się i wykonaniu komendy **pwd** wyświetlony komunikat powinien być podobny do tego na rys. 4.16 z dokładnością do ostatniego członu będącego nazwą użytkownika – jest to ścieżka do katalogu domowego użytkownika.

```
/home/users/xgucio
```

Rys. 4.16. Wyjście komendy **pwd**.

4.2.4 cd

Komenda **cd** zmienia bieżący katalog roboczy. Polecenie to bez żadnych parametrów ustawi bieżący katalog na katalog domowy użytkownika. Jeśli po nazwie polecenia jako parametr zostanie podana ścieżka do katalogu to stanie się on bieżącym katalogiem. Np. **cd /usr/bin** zmieni bieżący katalog na /usr/bin, **cd ..** zmieni bieżący katalog na katalog nadrzędny do aktualnego.

4.2.5 ps

Pojęcie procesu odnosi się do instancji wykonywanego programu. W skład procesu wchodzi instrukcje programu, otwarte pliki, strumień danych wejściowych od użytkownika systemu [19]. Istnieją dwa typy procesów w systemie Linux.

Procesy pierwszego planu (zwane również procesami interaktywnymi) - są one inicjowane i kontrolowane za pomocą sesji terminalowej. Innymi słowy, do systemu musi być podłączony użytkownik, aby rozpocząć takie procesy; nie uruchomiły się one automatycznie w ramach funkcji / usług systemowych.

Procesy w tle (zwane również procesami nieinteraktywnymi / automatycznymi) - są procesami niepołączonymi z terminalem; nie oczekują interakcji z użytkownikiem.

Istnieje wiele narzędzi systemu Linux do przeglądania / wyświetlania uruchomionych procesów w systemie, dwa tradycyjne i dobrze znane polecenia to **ps** i **top**. Komenda **ps -ef** wyświetla wszystkie procesy uruchomione w systemie – na rys. 4.17 przedstawiono dwie przykładowe linie z wykonania komendy.

```
...
root  2150  1245  0 07:50 ? 00:00:00
                sudo python -u /home/pi/sensortagcollector.py -o -d
root  2154  2150  0 07:50 ? 00:00:07
                python -u /home/pi/sensortagcollector.py -o -d ...
...
```

Rys. 4.17. Wyjście komendy **ps -ef**.

Pierwsza kolumna zawiera nazwę użytkownika (właściciela) procesu. Linux jest systemem dla wielu użytkowników – oznacza to, że różni użytkownicy mogą uruchamiać różne programy w systemie. Zatem każde uruchomione wystąpienie programu musi być jednoznacznie zidentyfikowane przez jądro. Każdy proces jest identyfikowany przez jego identyfikator procesu (process identifier, PID), który jest pokazany w drugiej kolumnie. Koniec każdej linii to polecenie, które posłużyło do uruchomienia danego procesu.

4.2.6 kill

Kill jest to program komputerowy w systemach Unix i Linux pozwalający na przesyłanie sygnałów do procesów pracujących w systemie operacyjnym. Domyślnie program kill wysyła do procesu sygnał nakazujący mu zakończenie pracy. Wbrew nazwie sugerującej natychmiastowe przerwanie pracy, polecenie powoduje poprawne zamknięcie procesu i zachowanie wszelkich wewnętrznych danych [20].

Przykładowe użycie komendy kill do zakończenia pracy pierwszego z procesów pokazanych na rys. 4.17 (proces jest identyfikowany numerem PID, który w tym przypadku ma wartość 2150): **kill 2150**

W przypadku, gdy proces nie reaguje na standardowe użycie komendy kill można użyć komendę kill z parametrem -9. Spowoduje to bezwarunkowe zakończenie wykonywania procesu. W takim wypadku może jednak nastąpić utrata danych. Przykładowe użycie komendy kill do bezwarunkowego zakończenia procesu: **kill -9 2150**

4.2.7 mkdir

Polecenie **mkdir** służy do utworzenia nowego katalogu. Parametrem jest nazwa nowo tworzonego katalogu. Przykładowe użycie komendy, do utworzenia katalogu o nazwie *projekt1* w katalogu bieżącym: **mkdir projekt1**

4.2.8 rm

Polecenie **rm** służy do usuwania plików i katalogów. Przy kasowaniu pojawia się pytanie o potwierdzenie skasowania. Aby uniknąć pytań należy użyć parametru **-f**. Przykład użycia polecenia **rm** do usunięcia pliku *main.cpp* znajdującego się w katalogu bieżącym:

rm -f main.cpp

Przykład użycia polecenia **rm** do usunięcia katalogu *projekt1* (wraz z zawartością) znajdującego się w katalogu bieżącym: **rm -rf projekt1**

4.2.9 cp

Polecenie **cp** służy do kopiowania plików i katalogów. Polecenia można używać na kilka różnych sposobów. Ogólny sposób wywołania można przedstawić w następująco:

cp źródło cel

Jeśli źródło jest nazwą pliku, a cel nazwą istniejącego katalogu to plik zostanie skopiowany do wskazanego katalogu. Jeśli źródło jest nazwą pliku, a cel nie jest nazwą katalogu to zawartość pliku źródłowego zostanie skopiowana do pliku podanego jako cel.

W przypadku, gdy skopiowany ma być katalog wraz z zawartością należy użyć komendy **cp** z parametrem **-r** oznaczającym kopiowanie rekursywne. Następujący przykład powoduje skopiowanie katalogu o nazwie *projekt1* do katalogu *archiwum* znajdującego się w katalogu nadrzędnym do bieżącego:

cp -r projekt1 ../archiwum

4.2.10 mv

Polecenie **mv** służy do przenoszenia lub zmiany nazwy plików i katalogów. Ogólny sposób wywołania można przedstawić w następująco:

mv źródło cel

Jeśli źródło jest nazwą pliku lub katalogu, a cel nazwą istniejącego katalogu to źródło zostanie przeniesione do wskazanego katalogu. Jeśli źródło jest nazwą pliku, a cel nie jest nazwą katalogu to plik źródłowy zostanie przeniesiony do pliku podanego jako cel.

4.2.11 tar

Program **tar** jest uniksowym programem do umieszczania zbioru plików w jednym nieskompresowanym pliku (tzw. archiwum). Może on następnie zostać skompresowany np. programem **gzip** automatycznie uruchamianym przez program **tar**. Skompresowane archiwa mają zwyczajowo rozszerzenie *.tar.gz* lub *.tgz*.

Składnia programu tar przy tworzeniu archiwum jest następująca:

tar -zcf plik_archiwum.tar.gz źródło1 źródło2...

Parametr **-z** odpowiada za kompresję archiwum programem **gzip**. Parametr **-c** oznacza tryb tworzenia archiwum, zaś po parametrze **-f** podana jest nazwa wynikowego pliku. *Źródło1*, *źródło2* itd. są nazwami plików lub katalogów do umieszczenia w archiwum.

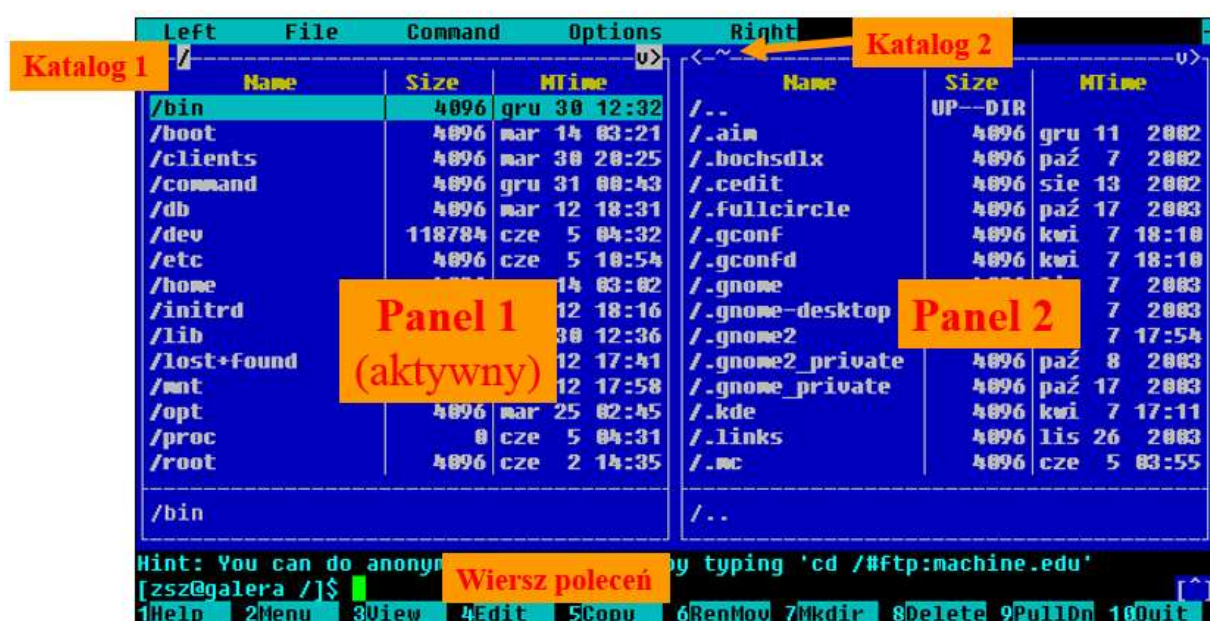
Składnia programu tar przy rozpakowaniu archiwum jest następująca:

tar -xzf plik_archiwum.tar.gz

Parametr **-x** oznacza tryb rozpakowania archiwum. Nazwa pliku do rozpakowania podana jest po parametrze **-f**. Archiwum jest rozpakowywane do bieżącego katalogu.

4.2.12 mc

Operacje plikowe przy pomocy komend linuxowych wydawanych z wiersza poleceń mogą być czasochłonne i nieinstuicyjne dla początkującego użytkownika. Program *Midnight Commander* zapewnia semigraficzny interfejs użytkownika w trybie tekstowym (rys. 4.18).



Rys. 4.18. Interfejs programu Midnight Commander (mc).

Program wyświetla na ekranie zawartość dwóch katalogów w dwóch panelach. Ścieżki dostępu do katalogów wyświetlają się na górnej krawędzi ramki okalającej panele. Jeden panel jest panelem aktywnym – w nim można przeglądać listę plików posługując się strzałkami w górę i w dół na klawiaturze oraz zaznaczać pliki i katalogi klawiszem *insert* (wtedy ich nazwa wyświetlana jest kolorem żółtym). Katalog można zmienić ustawiając podświetlenie na jego nazwie i naciskając klawisz *enter* (do katalogu nadrzędnego można przejść wybierając ..). Klawisz tabulacji pozwala zmienić aktywny panel. Podświetlone pliki można skopiować lub przenieść z katalogu wyświetlonego w aktywnym panelu do katalogu, którego zawartość pokazana jest w drugim panelu, naciskając odpowiednio klawisze *F5* lub *F6*. Wciśnięcie klawisza *F7* umożliwi utworzenie nowego katalogu. Kasowanie podświetlonego elementu lub grupy zaznaczonych na żółto elementów odbywa się po naciśnięciu klawisza *F8*. Uaktywnienie górnego menu następuje po naciśnięciu *F9*, zaś klawisz *F10* kończy działanie programu.

Program Midnight Commander udostępnia również wiersz poleceń, w którym użytkownik może wpisywać dowolne komendy. Jeśli uruchomione polecenie wyświetla komunikaty na ekranie to po zakończeniu działania zostaną one zasłonięte przez panele. Aby ponownie je zobaczyć należy wcisnąć kombinację klawiszy *CTRL+O*. Elementy interfejsu użytkownika programu *mc* zostaną wtedy całkowicie ukryte (program nadal działa). Ponowne ich wyświetlenie nastąpi po powtórnym naciśnięciu *CTRL+O*.

F3 - podgląd pliku	F6 - przenoszenie plików	F9 - menu
F4 - edycja pliku	F7 - utwórz katalog	F10 - zakończ
F5 - kopiowanie plików	F8 - kasowanie	insert - zaznaczanie
CTRL O - wł./wył. panele	TAB - zmiana aktywnego panela	

Rys. 4.19. Skróty klawiaturowe programu Midnight Commander (mc).

W programie dostępna jest też przeglądarka i edytor plików tekstowych uruchamiane odpowiednio klawiszami *F3* i *F4* (po umieszczeniu podświetlenia na nazwie pliku). Na rys. 4.19 umieszczono najczęściej używane skróty klawiaturowe programu *mc*.

4.3 Przykładowy program

Użycie programu w assemblerze procesora Intel zostanie zaprezentowane na przykładzie programu przedstawionego na listingach 4.1, 4.2. Jest dostępny do pobrania pod adresem: <http://galera.ii.pw.edu.pl/~zsz/arko/intel-intro.tar.gz>

Plik należy ściągnąć do katalogu domowego na serwerze Galera np. pobierając plik przeglądarką na własny komputer i przesyłając do katalogu domowego programem WinScp lub w wierszu poleceń w terminalu poleceniem:

```
wget http://galera.ii.pw.edu.pl/~zsz/arko/intel-intro.tar.gz
```

Następnie należy skompresowane archiwum *.tar.gz* rozpakować poleceniem

```
tar xzf intel-intro.tar.gz
```

Po rozpakowaniu powstanie katalog *intel-intro* zawierający trzy pliki:

- *main.cpp* – plik zawierający funkcję *main* w języku C++, z której wywoływana jest funkcja assemblerowa *func*;
- *func.asm* – funkcja assemblerowa wywoływana z programu głównego w języku C++;

- `makefile` – plik pozwalający zautomatyzować proces kompilacji.

Aby skompilować i uruchomić program należy zmienić katalog bieżący na *intel-intro*:

```
cd intel-intro
```

Następnie należy skompilować projekt przy pomocy polecenia *make* wykorzystującego konfigurację zapisaną w pliku *makefile*:

```
make
```

Ostatnim krokiem jest uruchomienie programu. Plik wykonywalny ma nazwę *test*, zatem uruchomienie programu następuje komendą:

```
./test
```

4.3.1 Omówienie działania przykładowego programu

W pliku *main.cpp* znajduje się wywołanie zewnętrznej funkcji *func* zdefiniowanej w pliku *func.asm*. Deklaracja `extern "C" int func(char *a);` sprawia, że nazwa funkcja będzie linkowana zgodnie z konwencjami języka C (oznacza to m.in., że kompilator nie modyfikuje nazwy funkcji podczas procesu kompilacji).

Plik assemblerowy zawiera tylko jedną sekcję – jest to sekcja *text*, w której znajduje się kod programu. Dyrektywa `global func` sprawia, że symbol *func* będzie widoczny dla zewnętrznych modułów i będzie mógł być wykorzystany przez linker.

Komentarze w pliku assemblerowym rozpoczynają się od znaku średnika `;`. Zawartość linii od znaku średnika do końca linii jest ignorowana przez kompilator.

Adresowanie elementów stosu odbywa się przy pomocy wskaźnika ramki stosu umieszczonego w rejestrze *ebp*. Po wejściu do funkcji, należy zachować zawartość wskaźnika ramki stosu funkcji wywołującej, tzn. zapisać rejestr *ebp* na stosie. Odpowiedzialna jest za to instrukcja `push ebp`. Przy wyjściu z funkcji będzie można odtworzyć starą zawartość rejestru *ebp*.

Następnie inicjowana jest nowa wartość wskaźnika ramki stosu w rejestrze *ebp* – jest nią aktualna wartość wskaźnika stosu (wskazującego adres elementu na szczycie stosu). Instrukcja `mov ebp, esp` kopiuje zawartość rejestru *esp* do rejestru *ebp*. W przypadku instrukcji assemblerowych z dwoma parametrami konwencja jest taka, że pierwszy parametr jest parametrem docelowym, a drugi źródłowym.

Omawiana funkcja ma jeden parametr – wskazanie na początek łańcucha znakowego. Znajduje się on pod adresem *EBP+8*. Schematyczny rysunek zawartości stosu umieszczono na końcu listingu 4.2. Widać na nim, że stos rośnie w kierunku mniejszych adresów i przy wywołaniu funkcji najpierw umieszczane są parametry funkcji (w tym wypadku *char *a*), adres powrotu, a następnie w wywołanej funkcji zachowywana jest wartość wskaźnika ramki stosu funkcji wywołującej, jak to opisano w poprzednim akapicie. Adresy są 32 bitowe, stąd przesunięcie o 8 bajtów przy dostępie do drugiego elementu względem wskaźnika ramki stosu. Instrukcja `mov eax, DWORD [ebp+8]` kopiuje daną o rozmiarze double word (*DWORD* – 4 bajty) umieszczoną w pamięci pod adresem *ebp+8* do rejestru *eax*. Nawiasy kwadratowe wskazują, że adres umieszczony jest w rejestrze, a kopiowana dana w pamięci. Po wykonaniu instrukcji rejestr *eax* zawiera adres ciągu znakowego podany jako parametr funkcji.

W kolejnej linii programu `mov BYTE [eax], 'w'` kod ASCII małej litery *w* wstawiany jest na pierwszej pozycji ciągu znakowego. Specyfikator rozmiaru *BYTE* odpowiada za wygenerowanie właściwej instrukcji procesora przesyłającej do pamięci tylko jeden bajt.

Wartość zwracana przez funkcję powinna być umieszczona w rejestrze *eax*. Omawiana funkcja zwraca zawsze wartość zero: `mov eax, 0`. Należy zwrócić uwagę, że lepszą metodą zerowania rejestru jest użycie instrukcji *xor*.

Instrukcja `pop ebp` zdejmuje ze stosu wartość i umieszcza ją w rejestrze *ebp*, co odtwarza zawartość wskaźnika ramki stosu funkcji wywołującej. Instrukcja `ret` zdejmuje ze stosu adres

powrotu. Wykonanie programu będzie kontynuowane od instrukcji pod tym adresem (następna instrukcja po instrukcji *call*).

Listing 4.1. Kod źródłowy pliku *main.cpp*.

```
#include <stdio.h>

extern "C" int func(char *a);

int main(void)
{
    char text[]="Wind On The Hill";
    int result;

    printf("Input string      > %s\n", text);
    result=func(text);
    printf("Conversion results> %s\n", text);

    return 0;
}
```

Listing 4.2. Kod źródłowy pliku *func.asm*.

```
...
section .text
global func

func:
    pushebp
    mov ebp, esp
    mov eax, DWORD [ebp+8]    ;address of *a to eax
    mov BYTE [eax], 'w'      ;a[0]='w'
    mov eax, 0                ;return 0
    pop ebp
    ret

;=====
; THE STACK
;=====
;
; larger addresses
;
; |                               |
; | ...                           |
; |-----|
; | function parameter - char *a | EBP+8
; |-----|
; | return address                | EBP+4
; |-----|
; | saved ebp                     | EBP, ESP
; |-----|
; | ... here local variables      | EBP-x
; |   when needed                 |
;
; \/\
; \/\ the stack grows in this    \/\
; \/\ direction                   \/\
;
; lower addresses
;
;
;=====
```

4.3.2 Omówienie pliku *makefile*

Listing 4.3. Plik *makefile*.

```
CC=g++
ASMBIN=nasm

all : asm cc link
asm :
    $(ASMBIN) -o func.o -f elf -g -l func.lst func.asm
cc :
    $(CC) -m32 -c -g -O0 main.cpp &> errors.txt
link :
    $(CC) -m32 -g -o test main.o func.o
clean :
    rm *.o
    rm test
    rm errors.txt
    rm func.lst
```

Na listingu 4.3 pokazano plik *makefile* odpowiedzialny za konfigurację procesu kompilacji i linkowania. Składa się on z trzech kroków. Asemlacja dokonywana jest programem *nasm*. Parametry wywołania programu to:

- `-o` definiuje nazwę wynikową pliku object po asemlacji (*func.o*);
- `-f` definiuje format pliku *.o* (*elf*);
- `-g` wymusza generowanie informacji dla debugera;
- `-l` włącza generowanie listingu (w pliku *func.lst*) zawierającego kod maszynowy w postaci szesnastkowej;
- ostatni parametr jest nazwą pliku poddawanego asemlacji (*func.asm*).

Kompilacja pliku *main.cpp* dokonywana jest programem *g++*. Znaczenie parametrów wywołania:

- `-m32` powoduje, że generowany jest kod 32 bitowy;
- `-c` powoduje kompilację pliku wskazanego jako ostatni parametr;
- `-g` wymusza generowanie informacji dla debugera;
- `-O0` wyłącza optymalizację przy kompilacji kodu;

Komunikaty i informacje diagnostyczne kompilatora przekierowane są do pliku o nazwie *errors.txt* (nie będą pojawiały się na ekranie).

Linkowanie wykonywane jest poprzez wywołanie programu *g++*. Parametr `-o` odpowiedzialny jest za nazwę wygenerowanego pliku wykonywalnego z programem (*test*). Na końcu polecenia znajduje się lista plików *object*, które mają zostać połączone przez linker w jeden program wykonywalny.

4.3.3 Rozbudowa przykładowego programu

Przykładowy program zostanie zmodyfikowany w taki sposób, aby funkcja *func* zamieniała wszystkie małe litery *a* w ciągu znakowym podanym jako parametr, na znaki gwiazdki ***. Wartość zwracana przez funkcję ma być liczbą wykonanych zamian.

Jak napisano w rozdz. 1.3 przed implementacją kodu assemblerowego warto napisać (pseudo)kod w języku C i dopiero na jego podstawie kodować instrukcje assemblerowe. Na listingu 4.4 przedstawiono kod omawianej funkcji. Czytelnik powinien samodzielnie dokonać jego analizy. Kod assemblerowy utworzony na podstawie funkcji w języku C pokazano na listingu 4.5.

Listing 4.4. Kod w języku C zmodyfikowanej funkcji *func*.

```

int func(char *a){
    replace_count=0;
    while(*a!='\0'){
        if (*a=='a'){
            *a='*';
            replace_count++;
        }
        a++;
    }
    return replace_count;
}

```

Przed napisaniem kodu z listingu 4.5 podjęto następujące decyzje projektowe:

- Adres bieżącego znaku będzie przechowywany w rejestrze *eax*;
- Liczba zmienionych znaków będzie przechowywana w rejestrze *ecx*;
- Wartość bieżącego znaku (wczytanego z pamięci) będzie przechowywana w rejestrze *bl*.

Listing 4.5. Zmodyfikowana funkcja *func*.

```

...
section .text
global func

func:
    pushebp
    mov ebp, esp
    mov eax, DWORD [ebp+8]    ;address of *a to eax

    xor ecx, ecx              ;replace_count=0
replace_loop:
    mov bl, [eax]             ;while(*a!='\0')
    cmp bl, 0
    je replace_exit           ;{
                                ;if (*a=='a')
    cmp bl, 'a'
    jne next_char
    mov BYTE [eax], '*'       ;{ *a='*'
    inc ecx                   ;replace_count++}

next_char:
    inc eax                   ;a++
    jmp replace_loop          ;}

replace_exit:
    mov eax, ecx              ;return replace_count
    pop ebp
    ret

```

Instrukcja `xor ecx, ecx` inicjuje zmienną `replace_count` – zeruje wartość w rejestrze *ecx*. Instrukcja `mov bl, [eax]` jest pierwszą instrukcją pętli `while`. Ładuje jeden bajt (jeden znak ciągu znakowego) z pamięci pod adresem zawartym w rejestrze *eax* i umieszcza go w rejestrze *bl*. Nawiasy kwadratowe wokół rejestru *eax* oznaczają użycie rejestrowego pośredniego trybu adresowania – argument operacji w takim przypadku znajduje się w pamięci, a w rejestrze jego adres.

Instrukcja `cmp bl, 0` jest instrukcją porównania, która ustawia odpowiednie flagi. Skok warunkowy `je replace_exit` jest wykonywany (na podstawie ustawionych flag) do etykiety

replace_exit jeśli rejestr *bl* ma wartość zero (oznacza to, że osiągnięto koniec łańcucha znakowego). W przeciwnym przypadku program przechodzi do kolejnej instrukcji.

Instrukcja porównania `cmp bl, 'a'` bada, czy kod ASCII znaku w rejestrze *bl* jest równy kodowi małej litery *a*. Jeśli nie to następuje skok do etykiety *next_char*. W przeciwnym przypadku modyfikowany jest bieżący znak na `*` - instrukcja `mov BYTE [eax], '*'` zapisuje do pamięci pod adres zawarty w rejestrze *eax* kod ASCII znaku `*`. Instrukcję `inc` inkrementuje licznik zmienionych znaków.

Instrukcja `inc` przy etykiecie *next_char* inkrementuje adres bieżącego znaku, zaś instrukcja `jmp replace_loop` jest instrukcją bezwarunkowego skoku do etykiety *replace_loop*, gdzie badany jest warunek pętli *while*.

Po wyjściu z pętli liczba zmienionych znaków przechowywana w rejestrze *ecx* jest kopiowana do rejestru *eax*, gdzie zgodnie z konwencją powinna być umieszczona wartość zwracana przez funkcję.

4.4 Przykładowa sesja uruchomieniowa gdb

Do wykrywania błędów i uruchamiania programów krok po kroku można wykorzystać debugger *gdb*. Jest to narzędzie z interfejsem tekstowym. W niniejszym rozdziale pokazano przykładową sesję debugową. Komendy wpisywane przez użytkownika wyróżnione są kolorem niebieskim, komunikaty analizowanego programu kolorem zielonym.

Parametrem wywołania programu *gdb* jest nazwa pliku wykonywalnego, którego działanie będzie analizowane, otrzymana w fazie linkowania (patrz pkt. 4.3.2). Program *gdb* wyświetla komunikaty początkowe pokazane na rys. 4.20.

```
[zsz@galera intel]$ gdb ./test
GNU gdb Red Hat Linux (6.5-15.fc6rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...Using host
libthread_db library
"/lib64/libthread_db.so.1".
```

Rys. 4.20. Przykładowe użycie programu *gdb* – część 1.

Znak zachęty debuggera ma postać napisu (*gdb*), po którym można wpisywać komendy. Pierwszą komendą pokazaną na rys. 4.21 jest komenda ustawiająca konwencję w jakiej będą wyświetlane instrukcje assemblerowe. Do wyboru jest tzw. konwencja *att* oraz *intel*, którą wybrano na przykładzie.

Polecenie *break* ustawia pułapkę na instrukcji związanej z etykietą *func*. Debugger potwierdza ustawienie pułapki wyświetlając jej numer i adres instrukcji, w której ustawiono pułapkę. Uruchomienie programu następuje po wydaniu polecenia *run*. W momencie, gdy

wykonanie programu osiągnęło pułapkę wyświetlany jest komunikat z numerem pułapki i jej adresem, a następnie pojawia się znak zachęty debuggera.

```
(gdb) set disassembly-flavor intel
(gdb) break func
Breakpoint 1 at 0x80484d3

(gdb) run
Starting program: /intel/test
warning: Lowest section in system-supplied DSO at 0xffffe000 is .hash at
ffffe0b4
Input string      > Wind

Breakpoint 1, 0x080484d3 in func ()
```

Rys. 4.21. Przykładowe użycie programu gdb – część 2.

Komenda *print/d* pokazana na rys. 4.22 wyświetla zawartość rejestrów *eax* i *ecx*, jako liczby całkowite ze znakiem. Dostępne są m.in. następujące formaty wyświetlania: */x* – szesnastkowy, */d* – dziesiętny (liczba całkowita ze znakiem), */u* – dziesiętny (liczba całkowita bez znaku), */t* – binarny, */a* - adres.

Komenda *info registers* wyświetla zawartość rejestrów.

```
(gdb) print/d $eax
$1 = -4719557
(gdb) print/d $ecx
$2 = 0
(gdb) info registers
eax          0xffffb7fc3b      -4719557
ecx          0x0              0
edx          0x5e20b0 6168752
ebx          0x5e0ff4 6164468
esp          0xffffb7fc18      0xffffb7fc18
ebp          0xffffb7fc18      0xffffb7fc18
esi          0x4a1ca0 4856992
edi          0x0              0
eip          0x80484d3          0x80484d3 <func+3>
eflags      0x286             [ PF SF IF ]
cs          0x23              35
ss          0x2b              43
ds          0x2b              43
es          0x2b              43
fs          0x0              0
gs          0x63              99
```

Rys. 4.22. Przykładowe użycie programu gdb – część 3.

Program może być wykonywany w trybie pracy krokowej przy użyciu instrukcji *stepi* (rys. 4.23) i *nexti*. Pierwsza z nich po napotkaniu instrukcji *call* przechodzi do wywoływanej funkcji. Druga – wykonuje wszystkie instrukcje funkcji i zatrzymuje się na pierwszej instrukcji za instrukcją *call*.


```

(gdb) stepi
0x080484d6 in func ()
(gdb) stepi
0x080484db in loop ()
(gdb) disassemble loop
Dump of assembler code for function loop:
0x080484db <loop+0>:    mov     bl,BYTE PTR [eax]
0x080484dd <loop+2>:    cmp     bl,0x0
0x080484e0 <loop+5>:    je      0x80484e6 <loop_exit>
0x080484e2 <loop+7>:    inc     eax
0x080484e3 <loop+8>:    inc     ecx
0x080484e4 <loop+9>:    jmp     0x80484db <loop>
End of assembler dump.

```

Rys. 4.23. Przykładowe użycie programu gdb – część 4.

Komenda *disassemble* służy do deasemblacji kodu maszynowego znajdującego się w pamięci komputera do postaci mnemoników czytelnych dla człowieka. Parametrem komendy może być m.in. nazwa funkcji, etykieta, zakres adresów.

Komenda *cont* pokazana na rys. 4.24 powoduje kontynuację wykonania programu w trybie pracy ciągłej, aż do napotkania kolejnej pułapki (lub zakończenia programu).

```

(gdb) break loop_exit
Breakpoint 2 at 0x80484e6
(gdb) cont
Continuing.
Breakpoint 2, 0x80484e6 in loop_exit ()

```

Rys. 4.24. Przykładowe użycie programu gdb – część 5.

Komenda *disassemble* bez dodatkowych parametrów przeprowadza deasemblację kodu począwszy od najbliższej etykiety. Komenda *quit* pokazane na rys. 4.25 służy do zakończenia pracy z programem *gdb*.

```

(gdb) disassemble
Dump of assembler code for function loop_exit:
0x080484e6 <loop_exit+0>:    mov     eax,ecx
0x080484e8 <loop_exit+2>:    pop     ebp
0x080484e9 <loop_exit+3>:    ret
0x080484ea <loop_exit+4>:    nop
...
0x080484ef <loop_exit+9>:    nop
End of assembler dump.
(gdb) cont
Continuing.
Length                > 4

Program exited normally.
(gdb) quit

```

Rys. 4.25. Przykładowe użycie programu gdb – część 6.

5 Małe programy dla procesora Intel

Korzystając z szablonu programu umieszczonego pod adresem **galera.ii.pw.edu.pl/~zsz/arko/intel-wstep.tar.gz** napisać program w assemblerze 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

6 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),

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). Na podstawie dostarczonych plików i opisu testowania osoba sprawdzająca program powinna być w stanie powtórzyć testy.

6.1 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

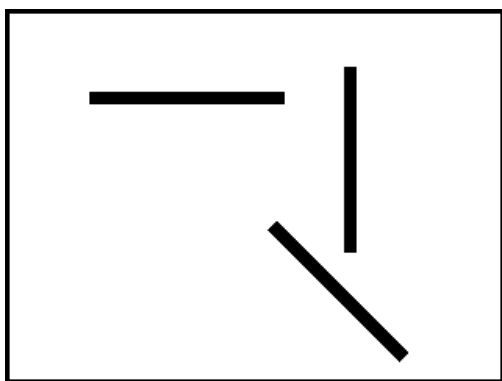
6.2 Co widzi robot

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 wiązki 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).

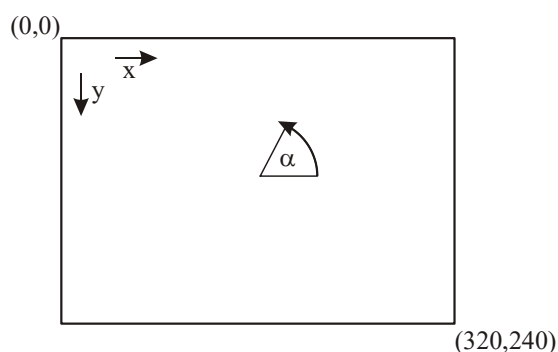
Wejście

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 $\langle 0, 240 \rangle$,
- α - 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,

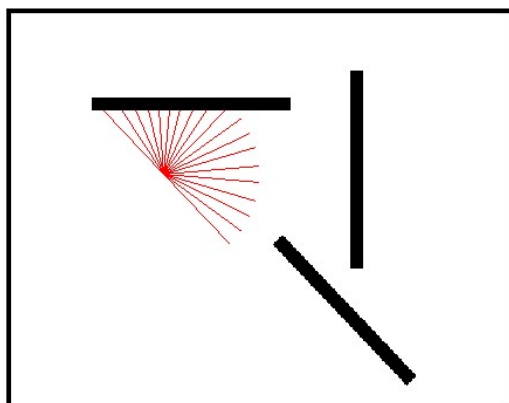
- liczba wiązek laserowych – 19. Wiązki są rozmieszczone równomiernie co 10° w zakresie $\langle 90^\circ, -90^\circ \rangle$ od kierunku, w którym skierowany jest robot.

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 położzeń 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° (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.

6.3 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.

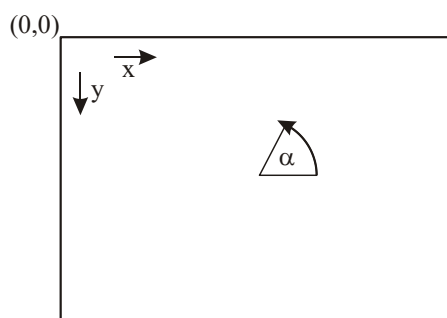
Wejście

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



Rys. 2. Zastosowany układ współzrędných.

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 bufory

```
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).

6.4 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
```

6.5 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_{dec}) jest nie większa niż 1024 znaki.
- tekst jawny zawiera znaki o kodach ASCII z przedziału <32_{dec}, 95_{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_{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_{dec}. 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.

6.6 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).

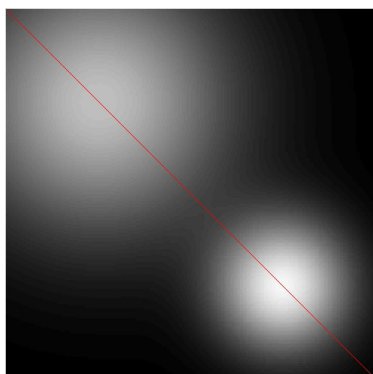
Wyjście

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

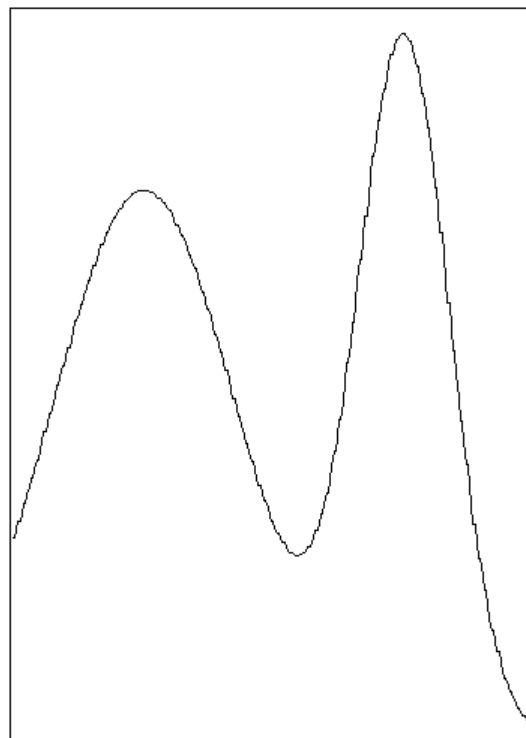
Plik mapy (rys. 1) zawiera obraz mapy wysokości w skali szarości. Obraz ma stałe rozmiary 201x201 pikseli. Wartość MIN z mapy wysokości jest reprezentowana przez kolor czarny. Wartości większe równe MAX reprezentowane są przez kolor biały. Pośrednie wartości reprezentowane są jako odcienie szarości.

Kolorem czerwonym zaznaczono linię wzdłuż której wykonany jest przekrój (linia powinna być ciągła, 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 asemblerze 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 nagłówków 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 nagłówków 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.

6.7 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 żółwiowej 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].

6.8 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 assemblerze 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 assemblerze 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 nagłówek BMP)
- **unsigned char *image2** – wskazanie na bufor zawierający drugi obraz (bez nagłówek 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 bufor realizowana jest na poziomie języka C.

6.9 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 bufory 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/>

6.10 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 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 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>

6.11 Binarna grafika żółwiowa – wersja 1

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend. Specyfikacja komend jest taka sama jak w zadaniu 3.11.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: **dowolny**,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.12 Binarna grafika żółwiowa – wersja 2

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend.

Specyfikacja komend jest taka sama jak w zadaniu 3.12.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.13 Binarna grafika żółwiowa – wersja 3

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend.

Specyfikacja komend jest taka sama jak w zadaniu 3.13.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.14 Binarna grafika żółwiowa – wersja 4

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend.

Specyfikacja komend jest taka sama jak w zadaniu 3.14.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.15 Binarna grafika żółwiowa – wersja 5

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend.

Specyfikacja komend jest taka sama jak w zadaniu 3.15.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.16 Binarna grafika żółwiowa – wersja 6

Zadaniem jest napisanie programu, który dokonuje translacji binarnie zakodowanych komend sterujących żółwiem na rastrowy plik w formacie BMP [5, 16]. Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w assemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję assemblerową o następującej deklaracji w języku C:

```
int exec_turtle_cmd(unsigned char *dest_bitmap,
                   unsigned char *command,
                   TurtleContextStruct *tc)
```

Opis:

Funkcja wykonuje jedną komendę grafiki żółwiowej znajdującą się w buforze komend.

Specyfikacja komend jest taka sama jak w zadaniu 3.16.

Wartość zwracana:

- 0 - w przypadku powodzenia
- liczba większa od 0 – numer błędu w przypadku, gdy wykonanie komendy nie powiodło się.

Parametry funkcji:

- **dest_bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **command** – wskazanie na bufor zawierający binarną komendę grafiki żółwiowej,
- **tc** – struktura przechowująca kontekst żółwia - służy do przekazania dodatkowych danych potrzebnych do prawidłowego wykonania sekwencji komend np.:

```
struct TurtleContextStruct
{
    int x_pos; //współrzędna x położenia żółwia
    int y_pos; //współrzędna y położenia żółwia
               //i ewentualnie inne potrzebne pola
}
```

Wejście

- Plik binarny zawierający 16/32-bitowe komendy sterujące żółwiem
- Nazwa pliku: “input.bin”
- Z pliku tekstowego “config.txt” należy pobrać dwie liczby całkowite (każda w oddzielnym wierszu) oznaczające rozmiar obrazu w pikselach.

Wyjście

- Plik BMP zawierający wygenerowany obraz:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
 - Nazwa pliku: “output.bmp”

Uwagi

1. Odczyt danych z pliku realizowany jest na poziomie języka C.
2. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
3. Nagłówek pliku BMP można wygenerować na poziomie języka C.
4. Nie wolno interpretować ani przetwarzać komend żółwia w języku C – jest to zadanie funkcji assemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik `makefile`, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **`turtle`**.
7. Projekt należy dostarczyć w formie jednego pliku **`.zip`** lub **`.tar.gz`**.

6.17 Znajdź znaczniki

Zadanie polega na napisaniu programu, który w obrazie w formacie BMP [5, 16] wykrywa jeden z dwunastu rodzajów znaczników. Wygląd znaczników oraz ich charakterystyka omówiona jest w zadaniu 3.17.

Program ma składać się z części napisanej w języku C/C++, z której wywoływana jest funkcja napisana w asemblerze procesora Intel x86 (32 bitowego). Pisząc program należy zaimplementować funkcję asemblerową o następującej deklaracji w języku C:

```
int find_markers (unsigned char *bitmap,
                 unsigned int *x_pos,
                 unsigned int *y_pos)
```

Opis:

Funkcja wykrywa znaczniki znajdujące się w obrazie zgodnie ze specyfikacją z zadania 3.17.

Wartość zwracana:

- wartość większa lub równa 0 - liczba wykrytych znaczników,
- liczba mniejsza od 0 – w przypadku wystąpienia błędu - jego numer.

Parametry funkcji:

- **bitmap** - wskazanie na bufor zawierający obraz (łącznie z nagłówkiem BMP),
- **x_pos** – tablica zawierająca współrzędne x wykrytych znaczników,
- **y_pos** – tablica zawierająca współrzędne y wykrytych znaczników.

Wejście

- Plik BMP zawierający obraz ze znacznikami:
 - Sub format: 24 bitowy RGB – bez kompresji,
 - Rozmiar obrazu: dowolny,
- Nazwa pliku pobierana jako parameter wywołania programu

Wyjście

- Terminal – w kolejnych wierszach wyświetlane są współrzędne wykrytych znaczników np. 10, 15. Punkt (0,0) znajduje się w górnym lewym rogu obrazu.

Uwagi:

1. Zakładamy, że na rysunku może wystąpić co najwyżej 50 znaczników danego rodzaju.
2. Odczyt danych z pliku realizowany jest na poziomie języka C.
3. Alokacja pamięci na bufory realizowana jest na poziomie języka C.
4. Cała interpretacja zawartości pliku BMP powinna być wykonana w funkcji asemblerowej.
5. Proszę zwrócić uwagę na możliwość wystąpienia bajtów dopełniających (ang. padding bytes) w przypadku, gdy liczba bajtów w wierszu obrazu nie jest podzielna przez 4.
6. Z projektem należy dostarczyć plik makefile, przy pomocy którego zostanie automatycznie wygenerowany plik wykonywalny o nazwie **find_markers**.
7. Projekt należy dostarczyć w formie jednego pliku **.zip** lub **.tar.gz**.

7 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
- [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
- [8] **„Lighting and Shading”**,
<http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/LightingAndShading.html>
- [9] **“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
- [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
- [16] **“file-format-bmp”**, https://en.wikipedia.org/wiki/BMP_file_format
- [17] **“Turtle graphics”**, https://en.wikipedia.org/wiki/Turtle_graphics
- [18] **Permissions**, http://linuxcommand.org/lc3_lts0090.php
- [19] Aaron Kili, **All You Need To Know About Processes in Linux [Comprehensive Guide]**, <https://www.tecmint.com/linux-process-management/>
- [20] **Polecenie kill**, [https://pl.wikipedia.org/wiki/Kill_\(polecenie\)](https://pl.wikipedia.org/wiki/Kill_(polecenie))

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

`http://galera.ii.pw.edu.pl/~zsz/arko`