Computer Architecture

Supplementary Materials

Zbigniew Szymański

January 2021

Institute of Computer Science

Nowowiejska 15 / 19,   00-665 Warszawa

Computer Architecture

supplementary materials

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

Table of Contents

1 MARS simulator

The Mars program is an integrated programming environment that allows you to create programs in assembly language of the MIPS processor. The program is written in Java, so it can be run on both Linux and Windows. You can download it from the website:

`http://courses.missouristate.edu/kenvollmar/mars/`

The use of the Mars program will be presented on the example of the program presented in Listing 1.1. It is available for download at:
`http://galera.ii.pw.edu.pl/~zsz/ecoar/mips.asm`

Listing 1.1. Code for the sample program in MIPS assembly language.

```
#-------------------------------------------------------------------------------
#author        : Zbigniew Szymanski
#date          : 2016.04.04
#description   : Read and display the input string
#-------------------------------------------------------------------------------
       .data
input: .space 80
prompt:.asciiz "\nInput string      > "
msg1:  .asciiz "\nConversion results> "
       .text
main:
#display the input prompt
        li $v0, 4          #system call for print_string
        la $a0, prompt     #address of string
        syscall
#read the input string
        li $v0, 8          #system call for read_string
        #...               #remember to set max length of the string!!!
        la $a0, input      #address of buffer
        syscall
#display the output prompt and the string
        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
```

The purpose of the program is to display a message **`Input string >`**, reading a string from the keyboard, displaying the message **`Conversion result>`** and displaying the string which was read from keyboard. One mistake was made intentionally in the program to show the code debugging process.

1.1 Description of the example program

The program has two sections beginning in the source code with `.data` and `.text` directives. The first section contains static data, the second - program code. Labels (starting at the beginning of the line and ending with :) are used to name the memory location following the label. The example program contains following labels:

- `input:`, `prompt:`, `msg1:` - associated with memory areas that store static variables;
- `main:`, `exit:` - associated with instruction addresses. The `main` label is particularly important because it is the entry point into the program. The instruction associated with it begins its execution.

The programmer can use human-readable names that will be converted into addresses at the assembly stage of the program.

The `.space` directive is used to declare data without giving it an initial value. In the sample program, 80 bytes are reserved for the read data buffer. The `.asciiz` directive is used to declare a character string (terminated with a zero code) that is given an initial value. In the example, two character strings containing messages displayed in the console are declared.

The three instructions following the main label are used to display the message associated with the prompt string in the console. The call to the *print_string* system function implemented by the Mars simulator is used. The table of all system functions is available after selecting the *Help* menu, then the *Help* menu item and the *Syscalls* tab in the help window. System functions are called using the *syscall* instruction (this is a no argument trap instruction). Before using it, put the number of the called system function in the `$v0` register, and the arguments of this function in the `$a_` registers.

The **`li $v0, 4`** instruction loads the immediate constant 4 (whose value is in the program code) into the `$v0` register. The name of the `li` statement is short for load immediate.

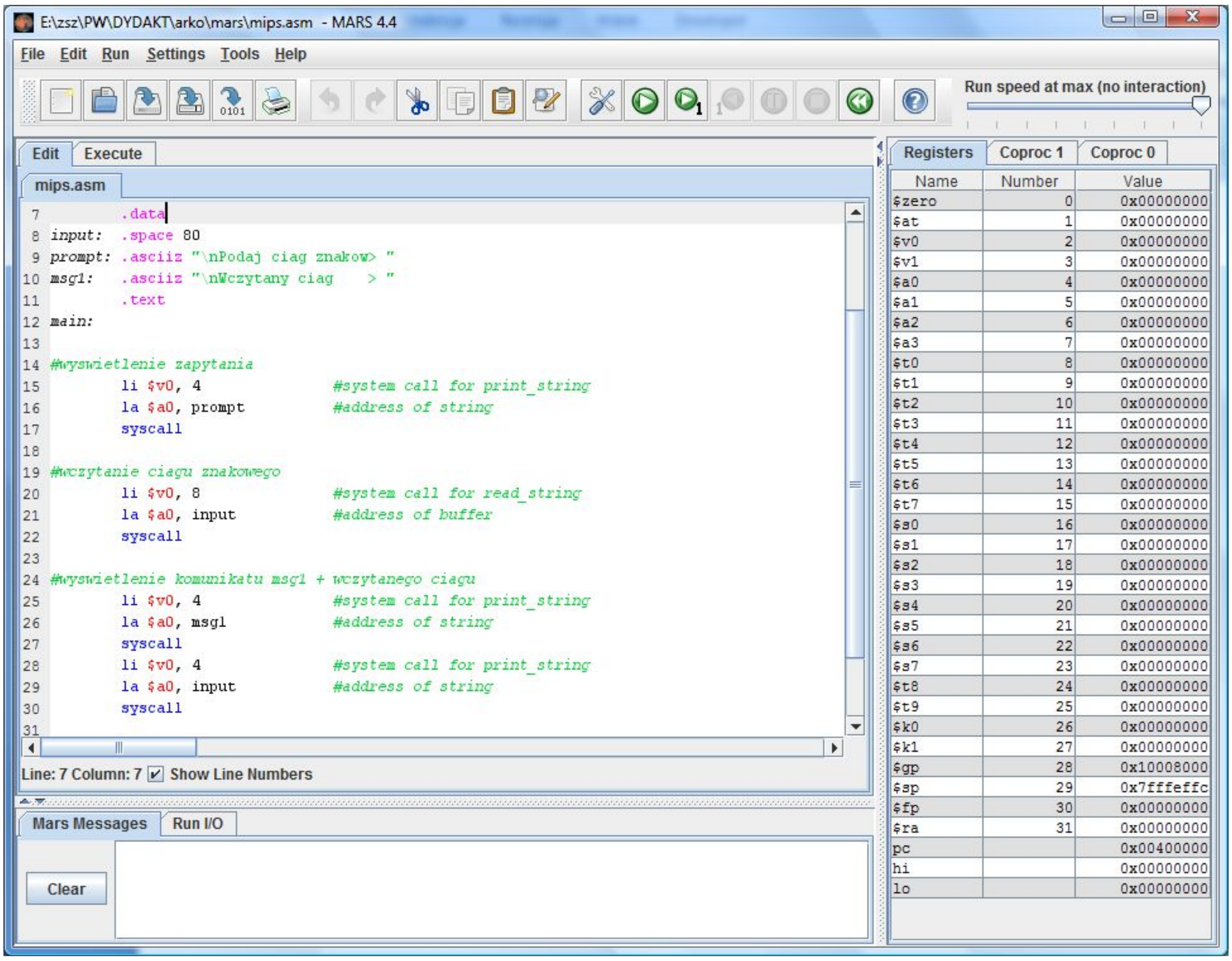
The **`la $a0, prompt`** instruction, loads the address of the given data (associated with the prompt label) into the register `$a0`. The name of the `la` statement is short for load address. It is worth noting that loading and storing data in memory can only be performed by instructions from the load and store instruction groups.

The # sign indicates the start of the comment - the text following it until the end of the line is ignored.

The next three instructions are used to call the *read_string* system function for reading the string from the keyboard. The *print_string* system function is then called twice – the first time the message associated with the *msg1* label is displayed, and the second time the content of the buffer associated with the *input* label is displayed. The program ends with a call to the system *exit* function.

1.2 Debugging

Fig. 1.1 shows the Mars program window after loading the sample program (*File* | *Open* menu). The central part of the window is occupied by the assembler program code (*Edit* tab). At the bottom of the program window, Mars simulator messages appear in the *Messages* tab e.g. regarding the success or failure of code assembly. In the right part of the window there is a preview of the processor register values used for debugging the program.


E:\zsz\PW\DYDAKT\arko\mars\mips.asm - MARS 4.4
File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
mips.asm
Line: 7 Column: 7 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0

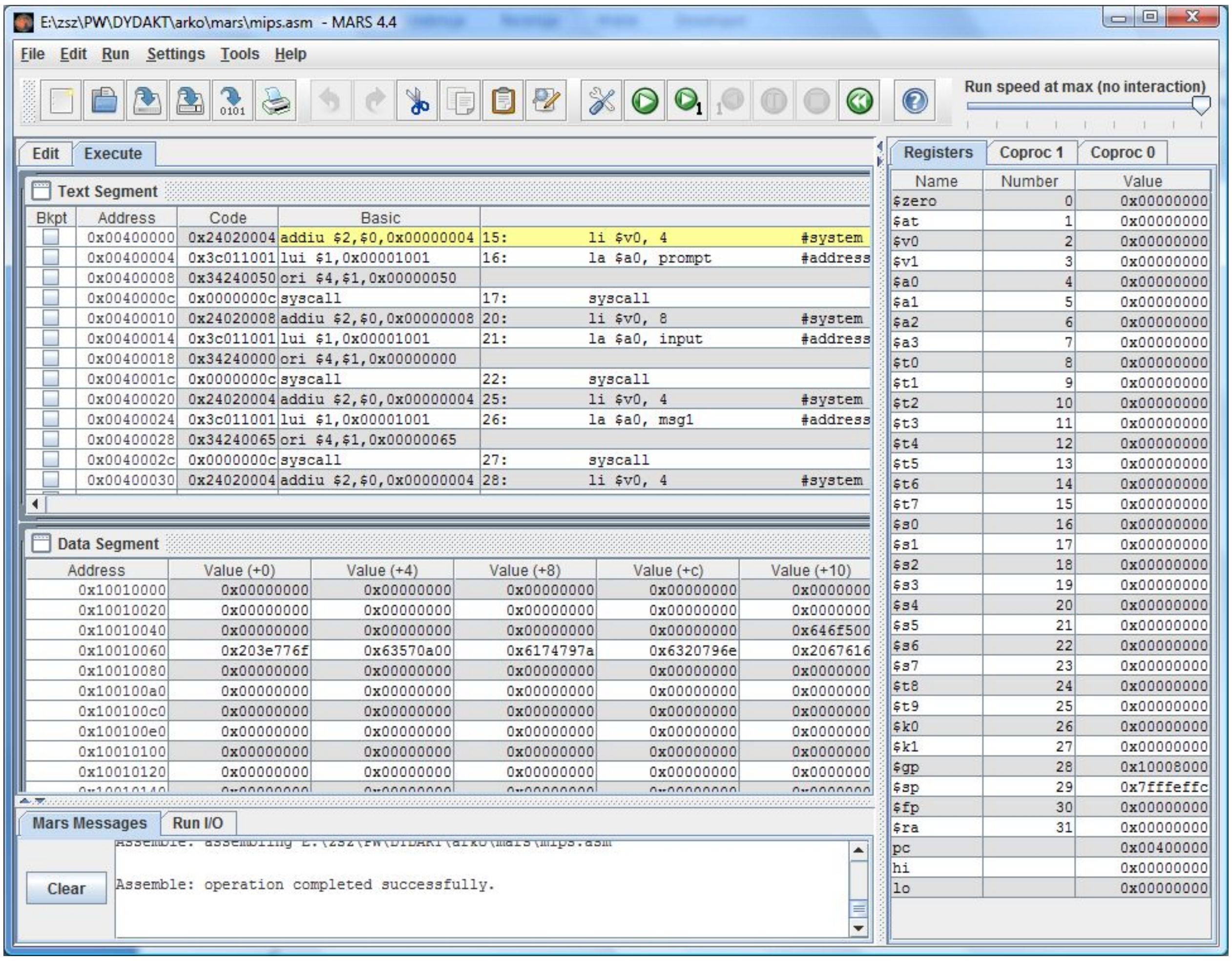
```
        .data
input:  .space 80
prompt: .asciiz "\nPodaj ciag znakow> "
msg1:   .asciiz "\nWczytany ciag    > "
        .text
main:

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

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

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

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Fig. 1.1 Mars environment - assembly code editing view

Before starting the program, you must perform code assembling (*Run* | *Assemble*, or *F3* key). A successful operation indicates „Assemble: operation completed successfully." message in *Mars Messages* tab (fig. 1.2). The program can be started by selecting the command from the *Run* | *Go* menu or by pressing *F5*. „Input string>" message will appear in the *Run I/O* tab. The program will allow you to enter only one character and will stop its operation, which is not consistent with the assumptions.

To run the program again, select *Run* | *reset* (or *F12* key). Please note (look at the *Registers* tab), that after selecting this command the PC register changes its value from 0x00400048 to 0x00400000, which means that the program will be executed from the beginning.


E:\zsz\PW\DYDAKT\arko\mars\mips.asm - MARS 4.4
File Edit Run Settings Tools Help
0101
Run speed at max (no interaction)
Edit
Execute
Text Segment
Data Segment
Registers
Coproc 1
Coproc 0
Mars Messages
Run I/O
Clear

Bkpt	Address	Code	Basic			
☐	0x00400000	0x24020004	addiu $2,$0,0x00000004	15:	li $v0, 4	#system
☐	0x00400004	0x3c011001	lui $1,0x00001001	16:	la $a0, prompt	#address
☐	0x00400008	0x34240050	ori $4,$1,0x00000050			
☐	0x0040000c	0x0000000c	syscall	17:	syscall	
☐	0x00400010	0x24020008	addiu $2,$0,0x00000008	20:	li $v0, 8	#system
☐	0x00400014	0x3c011001	lui $1,0x00001001	21:	la $a0, input	#address
☐	0x00400018	0x34240000	ori $4,$1,0x00000000			
☐	0x0040001c	0x0000000c	syscall	22:	syscall	
☐	0x00400020	0x24020004	addiu $2,$0,0x00000004	25:	li $v0, 4	#system
☐	0x00400024	0x3c011001	lui $1,0x00001001	26:	la $a0, msg1	#address
☐	0x00400028	0x34240065	ori $4,$1,0x00000065			
☐	0x0040002c	0x0000000c	syscall	27:	syscall	
☐	0x00400030	0x24020004	addiu $2,$0,0x00000004	28:	li $v0, 4	#system

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x646f500
0x10010060	0x203e776f	0x63570a00	0x6174797a	0x6320796e	0x2067616
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000
0x10010120	0x00000000	0x00000000	0x00000000	0x00000000	0x0000000

```
Assemble: assembling E:\zsz\PW\DYDAKT\arko\mars\mips.asm

Assemble: operation completed successfully.
```

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Fig. 1.2 Mars simulator – program debugging

In order to look how the program works just before calling the system function for reading the string from the keyboard, a breakpoint will be set at address 0x00400010. To do this, select the check box in the *Execute tab* in the *Bkpt* column in the line corresponding to the mentioned address and start the program (*Run* | *Go*). Program execution will be stopped before instruction at the address 0x00400010 is executed.

After selecting *Run* | *Step* (*F7* key) only one instruction will be executed in the step mode. Please note that the content of the $v0 register has changed (in the *Registers* tab) and is now equal to 8$_{dec}$. Step execution of the next two instructions will load the buffer address (in which the data read from keyboard are to be placed) into the $a0 register. The program is now stopped on the syscall instruction at address 0x0040001c.

By selecting *Help* | *Help* (*F1* key) you can check (by selecting the *Syscalls* tab at the bottom of the help window) the documentation for the *read string* function in the „Table of Available Services". The "arguments" column lists registers, whose value must be set for the function to work properly. These are the $a0 register containing the data buffer address and $a1 register containing the size of this buffer. It turns out that the registry value $a1 has not been set in the analyzed program. The program should be modified in the Edit tab as shown in Listing 1.2 (the added fragment is marked in gray).

Listing 1.2. The modified part of the sample program.

```
#read the input string
        li $v0, 8           #system call for read_string
        li $a1, 80          #remember to set max length of the string!!!
        la $a0, input       #address of buffer
        syscall
        ...
```

Then assemble and run the program once again. The modified program loads the string correctly and then displays it.

When debugging programs that operate on text data, it may be useful to save the content of the data segment to a text file. Selecting *File* | *Dump Memory* (CTRL d shortcut) opens a window titled „Dump Memory To File". From the *Memory Segment* list, select the name of the memory segment whose content is to be saved to the file, and from the *Dump Format* list, the method of saving. If "*Ascii text*" is selected, each line of the file will contain the content of one 32-bit memory word.

1.3 Program modification

The program from Listing 1.1 will now be enhanced - the length of the character string loaded from the keyboard will be counted. In case of coding in assembly language it is worth to write (pseudo) code in C language first. And then based on the C code to write assembly instructions. Listing 1.3 presents the program code for determining the length of the string. The reader should analyze this code on its own.

Listing 1.3. C language code for determining the length of the string

```
char *string;
int count;

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

Assembly code based on the C program is shown in Listing 1.4. The instructions shown should be placed in the sample program just before the *exit* label.

Listing 1.4. Assembler code for determining the length of the string

```
        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
```

The following design decisions were made before writing the code from Listing 1.4:

- The number of characters in the string (variable *count*) will be stored in the `$t1` register;

- The address of the current character (variable *string*) will be stored in the `$t2` register;
- The value of the current character (loaded from memory) will be stored in the `$t3` register.

The **`li $t1,0`** instruction initializes the *count* variable – it loads the immediate value 0 into the `$t1` register. The **`la $t2,input`** instruction, initializes the *string* variable - loads the address of the buffer associated with the `input` label into the `$t2` register.

The **`lbu $t3,($t2)`** instruction is the first instruction of the while loop. It loads one byte (one character from the string) from memory at the address contained in the `$t2` register and places it in the `$t3` register. The parentheses around the `$t2` register indicate the use of indirect register addressing mode – in this case the operand's argument is in memory and its address in the register.

The **`beqz $t3,loop_exit`** instruction is a conditional jump instruction (the name is an abbreviation of *branch if equal zero*). A jump is made to the `loop_exit` label if register `$t3` is zero (this means that the end of the string has been reached). Otherwise, the program proceeds to the next instruction.

The **`addi`** instruction adds the value of the third (immediate) argument to the value of the second argument (located in the register), and places the result in the register that is the first argument. The program uses the **`addi`** instruction to increment the character counter and the address of the current character.

The **`j loop`** statement is an unconditional jump instruction to the `loop` label, where the while loop condition is examined.

A call to the *print_integer* system function is placed after the loop. It’s task is to display the length of the string. The value 1 (system function number) is loaded into the `$v0` register. The move instruction copies the value from the **`$t1`** register (containing the length of the string) to the **`$a0`** register (which stores the argument of the *print_integer* function). The `syscall` instruction executes the system function.

After running the program, you can see that the displayed length of the string is longer than expected. This is because the *read_string* system call places additional byte, that represents the end of the line in the ASCII code, at the end of the string. The reader should modify the program by him/her self.

2 Short MIPS programs

Write a program in the MIPS assembly language. The input string should be read from the keyboard, converted and displayed on the screen.
You can use a template located at **`galera.ii.pw.edu.pl/~zsz/ecoar/mips.asm`**

1a	Convert all lower case letters to *. `Input string      > Wind On The Hill` `Conversion results> W*** O* T** H***`
1b	Convert all upper case letters to *. `Input string      > Wind On The Hill` `Conversion results> *ind *n *he *ill`
1c	Convert all digits to *. `Input string      > tel. 12-34-55` `Conversion results> tel. **-**-**`
1d	Convert all non letter characters to *. `Input string      > Wind On The Hill.` `Conversion results> Wind*On*The*Hill*`
2a	Swap the position of characters in consecutive pairs. `Input string      > Wind On The Hill` `Conversion results> iWdnO  nhT eiHll`
2b	Reverse the order of characters in the string. `Input string      > Wind On The Hill` `Conversion results> lliH ehT nO dniW`
2c	At the beginning of the output string put the characters from the odd positions, next the even. `Input string      > Wind On The Hill` `Conversion results> Wn nTeHlidO h il`
3a	Replace each character belonging to a word by the number of upper case characters in this word (mod 10). `Input string      > Wind ON The HiLL` `Conversion results> 1111 22 111 3333`
3b	Replace each character belonging to a word by the number of lower case characters in this word (mod 10). `Input string      > Wind ON The HiLL` `Conversion results> 3333 00 222 1111`
3c	Replace each character belonging to a word by the length of the word (mod 10). `Input string      > Wind ON The HiLL` `Conversion results> 4444 22 333 4444`

4a	The first and the second character in the string represent the (begin and the end) markers, which define a substring. Your task is to replace all characters between the first occurrence of begin marker and first occurcence of the end marker with * character. If there is no begin or end marker in the input string (the string after the **:** character), then nothing should be changed. Replace the first three characters of the string with spaces.
	`Input string      > oi:wind on the hill`
	`Conversion results>    wind *********ll`

4b	The first and the second character in the string represent the (begin and the end) markers, which define a substring. Your task is to replace all characters before the first occurrence of begin marker and first occurcence of the end marker with * character. If there is no begin or end marker in the input string (the string after the **:** character), then nothing should be changed. Replace the first three characters of the string with spaces.
	`Input string      > oi:wind on the hill`
	`Conversion results>    *****on the hi**`

4c	The first and the second character define number of characters wich should be left unchanged at beginning and at the end of the input string (the string after the **:** character). Your task is to replace all other characters with * character. If the sum of the two digits is larger then the length of the input string, then nothing should be changed. Replace the first three characters of the input string with spaces.
	`Input string      > 34:wind on the hill`
	`Conversion results>    win*********hill`

4d	The first and the second character define number of characters wich should be changed at beginning and at the end of the input string (the string after the **:** character). Your task is to replace required characters with * character. If the sum of the two digits is larger then the length of the input string, then nothing should be changed. Replace the first three characters of the input string with spaces.
	`Input string      > 34:wind on the hill`
	`Conversion results>    ***d on the ****`

5a	Write function `remove` which removes from the source string every small letter. `remove` returns the length of the resulting string. Source> `Computer Architecture Lab` Result> `C A L` `Return value: 5`
5b	Write function `remove` which removes from the source string every capital letter. `remove` returns the length of the resulting string. Source> `Computer Architecture Lab` Result> `omputer rchitecture ab` `Return value: 22`
5b	Write function `remove` which removes from the source string every digit. `remove` returns the length of the resulting string. Source> `7 plus 8 is 15` Result> ` plus  is ` `Return value: 10`
5c	Write function `remove` which removes from the source string every character that IS NOT a capital letter. `remove` returns the length of the resulting string. Source> `Implemented in ALGOL` Result> `IALGOL` `Return value: 6`
5d	Write function `remove` which removes from the source string every character that IS NOT a small letter. `remove` returns the length of the resulting string. Source> `Implemented in ALGOL!` Result> `mplementedin` `Return value: 12`
5e	Write function `remove` which removes from the source string every character that IS NOT a digit. `remove` returns the length of the resulting string. Source> `7*8=56` Result> `7856` `Return value: 4`
5f	Write function `remove` which removes from the source string every character before the first occurrence of left square bracket ([) and after the first following it occurrence of right square bracket (]). `remove` returns the length of the resulting string. Source> `Il ][ barbiere ][ di Siviglia` Result> `[ barbiere ]` `Return value: 12`
5g	Write function `remove` which removes from the source string every character between the first occurrence of left square bracket ([) and the first following it occurrence of right square bracket (]). `remove` returns the length of the resulting string. Source> `Il ][ barbiere ][ di Siviglia` Result> `Il ][][ di Siviglia` `Return value: 19`

6a	Write function `replace` which replaces in the source string every small letter with an asterisk (*). `replace` returns the length of the resulting string. Source> `Computer Architecture Lab` Result> `C******* A*********** L**` `Return value: 25`
6b	Write function `replace` which replaces in the source string every capital letter with an asterisk (*). `replace` returns the length of the resulting string. Source> `Computer Architecture Lab` Result> `*omputer *rchitecture *ab` `Return value: 25`
6c	Write function `replace` which replaces in the source string every digit with an asterisk (*). `replace` returns the length of the resulting string. Source> `7 plus 8 is 15` Result> `* plus * is **` `Return value: 14`
6d	Write function `replace` which replaces in the source string every character that IS NOT a capital letter with an asterisk (*). `replace` returns the length of the resulting string. Source> `Implemented in LISP` Result> `I**************LISP` `Return value: 19`
6e	Write function `replace` which replaces in the source string every character that IS NOT a small letter with an asterisk (*). `replace` returns the length of the resulting string. Source> `Implemented in LISP!` Result> `*mplemented*in******` `Return value: 20`
6f	Write function `replace` which replaces in the source string every character that IS NOT a digit with a question mark (?). `replace` returns the length of the resulting string. Source> `7*8=56` Result> `7?8?56` `Return value: 6`
6g	Write function `replace` which replaces in the source string every character before the first occurrence of left square bracket ([) and after the first following it occurrence of right square bracket (]) with an asterisk (*). `replace` returns the length of the resulting string. Source> `Eine ][ kleine ][ Nachtmusik` Result> `******[ kleine ]************` `Return value: 28`
6h	Write function `replace` which replaces in the source string every character between the first occurrence of left square bracket ([) and the first following it occurrence of right square bracket (]) with an asterisk (*). `replace` returns the length of the resulting string. Source> `Eine ][ kleine ][ Nachtmusik` Result> `Eine ][********][ Nachtmusik` `Return value: 28`

3 MIPS projects

Each project should be delivered as an *.asm file and a set of related test cases. The test cases should confirm that the program works correctly. Each project should contain a directory named *tests*. The sub directories of the directory *tests* correspond to individual tests. They should contain:

- input file(s),
- output file(s),
- a text file (named *description.txt*) containing short description what was being tested and the values of the input parameters (if any).

3.1 Color replacement

Replace color of pixels in an image by corresponding sepia tones. The replacement takes place when inequality (1) is satisfied

$$\boldsymbol{dist} \geq \sqrt{(\boldsymbol{R}-\boldsymbol{R_{sel}})^2+(\boldsymbol{G}-\boldsymbol{G_{sel}})^2+(\boldsymbol{B}-\boldsymbol{B_{sel}})^2} \qquad (1)$$

where (R,G,B) are the values of red, green, blue color components of the pixel, (R_{sel},G_{sel},B_{sel}) are the values of red, green, blue components of selected color (one for the whole image) and *dist* is the size of the color neighbourhood.

The formula for calculating of sepia tone [1] of a pixel:

```
outputRed   = (inputRed * .393) + (inputGreen *.769) + (inputBlue * .189)
outputGreen = (inputRed * .349) + (inputGreen *.686) + (inputBlue * .168)
outputBlue  = (inputRed * .272) + (inputGreen *.534) + (inputBlue * .131)
```

If any of these output values is greater than 255, you set it to 255.

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”
- (R_{sel},G_{sel},B_{sel}) – the values of red, green, blue components (0-255) of the replaced color (input from keyboard)
- *dist* - the size (0-442) of the color neighbourhood (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”

References:

[1] Zach Smith, “**How do I... Convert images to grayscale and sepia tone using C#?**” http://www.techrepublic.com/blog/how-do-i/how-do-i-convert-images-to-grayscale-and-sepia-tone-using-c/
[2] BMP file format – see section 4.2

3.2 Shading

Perform smooth shading of a triangle. Shading is the process of altering the color of pixels of a polygon. The colors of the vertices of the triangle are given. To find the R,G,B components of the color of other pixels linear interpolation may be used:

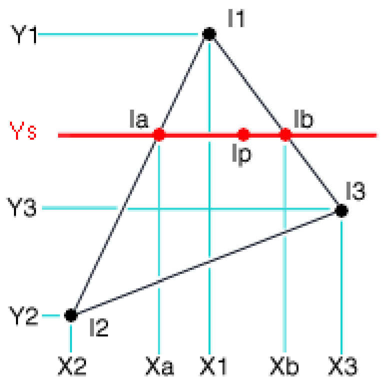


Fig.1 Smooth (interpolation) shading [1].

```
Ia = (Ys - Y2) / (Y1 - Y2) * I1 + (Y1 - Ys) / (Y1 - Y2) * I2
Ib = (Ys - Y3) / (Y1 - Y3) * I1 + (Y1 - Ys) / (Y1 - Y3) * I3
Ip = (Xb - Xp) / (Xb - Xa) * Ia + (Xp - Xa) / (Xb - Xa) * Ib
```

We assume that the location of the vertices of the triangle is fixed.

Input

- (R_1,G_1,B_1), (R_2,G_2,B_2), (R_3,G_3,B_3),– the values of red, green, blue components (0-255) of the color (input from keyboard) of three vertices

Output

- BMP file containing shaded triangle:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: “shading.bmp”

References:

[1] „**Lighting and Shading**”, http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/LightingAndShading.html
[2] BMP file format – see section 4.2

3.3 Binary image

Convert rectangular part of the source BMP image (24 bit RGB) to binary image (black white) using thresholding. The color of output pixel is white when inequality (1) is satisfied. Otherwise the pixel is black. The image outside the rectangular region is unchanged.

$$\boldsymbol{thres \geq 0.21R + 0.72G + 0.07B} \quad (1)$$

where R,G,B are the components of the color of the pixel.

Input

- BMP file containing the source image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: "source.bmp"
- (x1,y1) – top left corner of rectangular region (input from keyboard)
- (x2, y2) – bottom right corner of rectangular region (input from keyboard)
- thres – threshold value (input from keyboard)

Output

- BMP file containing modified image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: "dest.bmp"

References:

[1] BMP file format – see section 4.2

3.4 Puzzle

Divide the source image from a BMP file into n by m pieces and put them in random order.

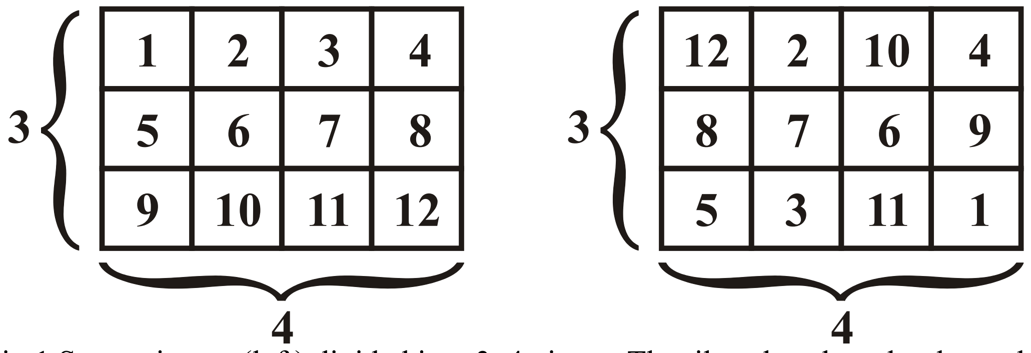


Fig.1 Source image (left) divided into 3x4 pieces. The tiles placed randomly on the destination image (right).

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: "source.bmp"
- n – number of vertical divisions (input from keyboard)
- m – number of horizontal divisions (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: "dest.bmp"

References:

[1] BMP file format – see section 4.2

3.5 Puzzle 2

Divide the source image from a BMP file into 3 by 4 pieces and put them in new order. Location of each piece in source image is described by two digits representing row number and column number. The new order of pieces in the destination image is described row-wise (from left to right) by numbers representing original locations of the pieces (fig. 2).

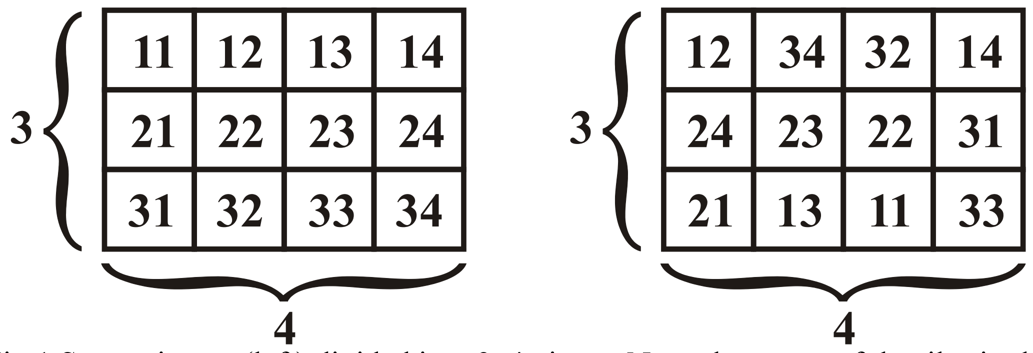


Fig.1 Source image (left) divided into 3x4 pieces. New placement of the tiles in the destination image (right).

`12,34,32,14,24,23,22,31,21,13,11,33`

Fig.2 Description of the placement of the tiles in the destination image (fig. 1)

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 318x240px,
  - file name: “source.bmp”
- s – description of the placement of the tiles in the destination image (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “dest.bmp”

References:

[1] BMP file format – see section 4.2

3.6 Maximum intensity

Mark (with a red frame) the region of the source BMP image (24 bit RGB) where the average intensity is maximum. The intensity of the pixel should be calculated according to formula (1). The average intensity should be calculated for a window of size m by n. If there is more than one region of the maximum intensity then the first detected should be marked.

$$\boldsymbol{I = 0.21R + 0.72G + 0.07B} \tag{1}$$

Input

- BMP file containing the source image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: “source.bmp”
- m – horizontal size of the scanning window (input from keyboard)
- n – vertical size of the scanning window (input from keyboard)

Output

- BMP file containing modified image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: “dest.bmp”

References:

[1] BMP file format – see section 4.2

3.7 Adding text

Add a single line of text (containing numbers and dots) to BMP image. The characters should be defined by a matrix of size 8 x 8 pixels eg. number 1:

```
. . . * * . . .
. * * * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . * * * * . .
```

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”
- the text string containing only numbers and dots (input from keyboard)
- (*x,y*) – the position of the text in the image (input from keyboard)

Output

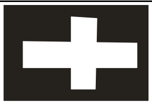
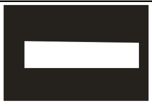
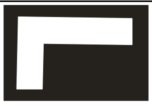
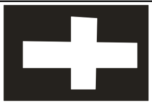
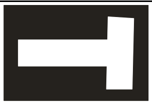
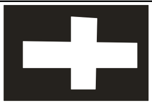
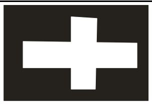
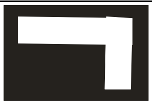
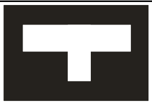
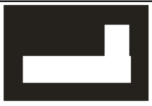
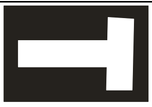
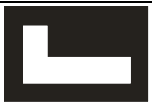
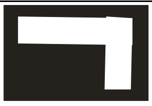
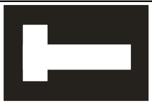
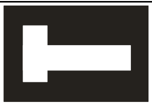
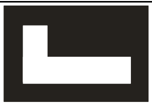
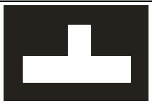
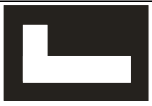
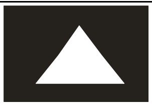
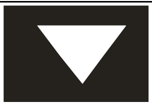
- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “dest.bmp”

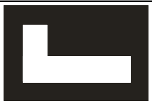
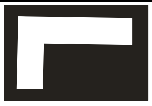
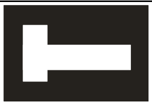
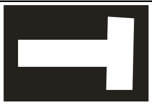
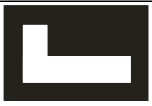
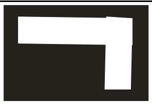
References:

[1] BMP file format – see section 4.2

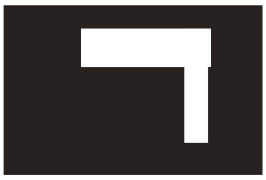
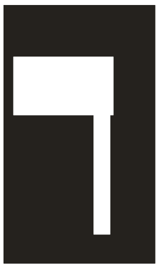
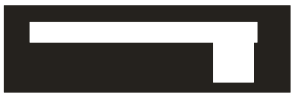
3.8 Shape detection

The input BMP image (24 bit RGB) contains one of two symbols from the given set. Your task is to recognize the shape and print in the console window its name ("Shape 1" or "Shape 2").

Set no.	Shape 1	Shape 2
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

11		
12		
13		

Take into account that shapes in the BMP file can be scaled. Images below show different versions of the same shape:

Input

- BMP file containing the source image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240 px,
    - file name: “source.bmp”

Output

- Console window – plain text

References:

[1] BMP file format – see section 4.2

3.9 Code 128 - barcode decoding

Write a program in the MIPS assembly language which decodes selected subset of Code 128 barcode.

Code 128 description

Code 128 encodes 128 symbols from one of three sets (Code Set A, Code Set B, Code Set C). The start symbol determines which subset is used. There are four widths of bars and spaces. Each symbol consists of three bars and three spaces (except stop symbol). Stop symbol consists of four bars and three spaces. The widths of bars and spaces are integer (1/2/3/4) multiples of the width of narrowest bar or space.

Code 128 consists of:

- quiet zone (empty space) – ten times the width of narrowest space or bar,
- start symbol,
- encoded data,
- check symbol,
- stop symbol,
- quiet zone.

Check symbol

The check symbol is calculated according to the formula:

$$check\_symbol\_value = \left(start\_symbol\_value + \sum_{i=1}^{data\ count} i * data\_value[i] \right) mod\ 103$$

Character encoding

The three Code 128 sets (Code Set A, Code Set B, Code Set C) are presented in table 1. Last column contains relative widths of bars (B) and spaces (S).

Table 1. Character codes

Value	Code set A	Code set B	Code set C	B S B S B S
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	I	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

Input

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “source.bmp”

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be displayed on standard output.

Project versions:

1. Code Set A decoding
2. Code Set B decoding
3. Code Set C decoding
4. Decoding of any set (A, B or C).

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. The program can scan only one line of the image (e.g. middle line).

References:

[1] BMP file format – see section 4.2
[2] “**Code 128**”, https://en.wikipedia.org/wiki/Code_128
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.10 Code 128 - barcode generation

Write a program in the MIPS assembly language which encodes a text using selected subset of Code 128 barcode.

Code 128 description

Code 128 encodes 128 symbols from one of three sets (Code Set A, Code Set B, Code Set C). The start symbol determines which subset is used. There are four widths of bars and spaces. Each symbol consists of three bars and three spaces (except stop symbol). Stop symbol consists of four bars and three spaces. The widths of bars and spaces are integer (1/2/3/4) multiples of the width of narrowest bar or space.

Code 128 consists of:

- quiet zone (empty space) – ten times the width of narrowest space or bar,
- start symbol,
- encoded data,
- check symbol,
- stop symbol,
- quiet zone.

Check symbol

The check symbol is calculated according to the formula:

$$check\_symbol\_value = \left(start\_symbol\_value + \sum_{i=1}^{data\ count} i * data\_value[i] \right) mod\ 103$$

Character encoding

The three Code 128 sets (Code Set A, Code Set B, Code Set C) are presented in table 1. Last column contains relative widths of bars (B) and spaces (S).

Table 1. Character codes

Value	Code set A	Code set B	Code set C	B S B S B S
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	I	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

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
    - Colors: bars – black, background – white.
- file name: “output.bmp”

Project versions:

1. Code Set A encoding
2. Code Set B encoding
3. Code Set C encoding
4. Encoding of any set (A, B or C).

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.

References:

[1] BMP file format – see section 4.2
[2] “**Code 128**”, https://en.wikipedia.org/wiki/Code_128
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.11 Code 39 - barcode decoding

Write a program in the MIPS assembly language which decodes Code 39 barcode.

Code 39 description

Code 39 encodes 43 symbols. The start and stop symbol is *. It can not be used in encoded data. There are two widths of bars and spaces. The thickness ratio of wide and thin bars and spaces may vary from 2,2:1 to 3:1 (in a given barcode it is constant). Each symbol consists of five bars and four spaces. The size of the space between two characters is not defined – usually it equals the width of thin bar or space.

Code 39 consists of:

- start symbol *,
- encoded data,
- check symbol,
- stop symbol.

Check symbol

The check symbol is calculated according to the formula:

$$check\_symbol\_value = \left(\sum_{i=1}^{data\ count} data\_value[i] \right) mod\ 43$$

Character encoding

The encoding of characters is presented in table 1. Last column contains relative widths of bars (B) and spaces (S). Digit 2 represents wide bar or space, digit 1 – thin bar or space.

Table 1. Character codes

Value	Char	B S B S B S B S B
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

Input

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: "source.bmp"

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be desplayed on standard output.

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. The program can scan only one line of the image (e.g. middle line).

References:

[1] BMP file format – see section 4.2
[2] "**Code 39**", https://en.wikipedia.org/wiki/Code_39
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.12 Code 39 - barcode generation

Write a program in the MIPS assembly language which encodes a text using Code 39 barcode.

Code 39 description

Code 39 encodes 43 symbols. The start and stop symbol is *. It can not be used in encoded data. There are two widths of bars and spaces. The thickness ratio of wide and thin bars and spaces may vary from 2,2:1 to 3:1 (in a given barcode it is constant). Each symbol consists of five bars and four spaces. The size of the space between two characters is not defined – usually it equals the width of thin bar or space.

Code 39 consists of:

- start symbol *,
- encoded data,
- check symbol,
- stop symbol.

Check symbol

The check symbol is calculated according to the formula:

$$check\_symbol\_value = \left(\sum_{i=1}^{data\ count} data\_value[i] \right) mod\ 43$$

Character encoding

The encoding of characters is presented in table 1. Last column contains relative widths of bars (B) and spaces (S). Digit 2 represents wide bar or space, digit 1 – thin bar or space.

Table 1. Character codes

Value	Char	B S B S B S B S B
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

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
    - Colors: bars – black, background – white.
- file name: “output.bmp”

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.

References:

[1] BMP file format – see section 4.2
[2] “**Code 39**”, https://en.wikipedia.org/wiki/Code_39
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.13 RM4SCC - barcode decoding

Write a program in the MIPS assembly language which decodes RM4SCC barcode [2].

RM4SCC code description

Each encoded character consists of top and bottom part. Each part is made up of 4 bars. Two of them are extended upward (in top part), and two are extended downward (in bottom part). The start symbol is a single bar extending upward. Stop symbol is a single bar extending upward and downward. RM4SCC code consists of:

- start symbol,
- encoded data,
- check symbol,
- stop symbol.

Check symbol

The check symbol is calculated according to the formula:

$$top\_check\_symbol\_value = \left(\sum_{i=1}^{data\ count} top\_value[i]\right) mod\ 6$$

$$bottom\_check\_symbol\_value = \left(\sum_{i=1}^{data\ count} bottom\_value[i]\right) mod\ 6$$

Look up the check symbol in table 1 in row corresponding to *top_check_symbol_value* and column corresponding to *bottom_check_symbol_value*.

Character encoding

The encoding of characters is presented in table 1 - 0 represents short bar, 1 represents long (extended) bar.

Table 1. RM4SCC barcode symbols [2]

		Bottom pattern					
		0011	**0101**	**0110**	**1001**	**1010**	**1100**
Top pattern	**Value**	**1**	**2**	**3**	**4**	**5**	**0**
0011	**1**	0	1	2	3	4	5
0101	**2**	6	7	8	9	A	B
0110	**3**	C	D	E	F	G	H
1001	**4**	I	J	K	L	M	N
1010	**5**	O	P	Q	R	S	T
1100	**0**	U	V	W	X	Y	Z

Input

- BMP file containing the barcode image:

    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “source.bmp”

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be desplayed on standard output.

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. The program can scan only three selected lines of the image (e.g. middle, top and bottom lines).

References:

[1] BMP file format – see section 4.2
[2] “**RM4SCC**”, https://en.wikipedia.org/wiki/RM4SCC
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.14 RM4SCC - barcode generation

Write a program in the MIPS assembly language which encodes a text using RM4SCC barcode.

RM4SCC code description

Each encoded character consists of top and bottom part. Each part is made up of 4 bars. Two of them are extended upward (in top part), and two are extended downward (in bottom part). The start symbol is a single bar extending upward. Stop symbol is a single bar extending upward and downward. RM4SCC code consists of:

- start symbol,
- encoded data,
- check symbol,
- stop symbol.

Check symbol

The check symbol is calculated according to the formula:

$$top\_check\_symbol\_value = \left(\sum_{i=1}^{data\ count} top\_value[i]\right) mod\ 6$$

$$bottom\_check\_symbol\_value = \left(\sum_{i=1}^{data\ count} bottom\_value[i]\right) mod\ 6$$

Look up the check symbol in table 1 in row corresponding to *top_check_symbol_value* and column corresponding to *bottom_check_symbol_value*.

Character encoding

The encoding of characters is presented in table 1 - 0 represents short bar, 1 represents long (extended) bar.

Table 1. RM4SCC barcode symbols [2]

		Bottom pattern					
		0011	**0101**	**0110**	**1001**	**1010**	**1100**
Top pattern	**Value**	**1**	**2**	**3**	**4**	**5**	**0**
0011	**1**	0	1	2	3	4	5
0101	**2**	6	7	8	9	A	B
0110	**3**	C	D	E	F	G	H
1001	**4**	I	J	K	L	M	N
1010	**5**	O	P	Q	R	S	T
1100	**0**	U	V	W	X	Y	Z

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
    - Colors: bars – black, background – white.
- file name: “output.bmp”

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.

References:

[1] BMP file format – see section 4.2
[2] “**RM4SCC**”, https://en.wikipedia.org/wiki/RM4SCC
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes
[4] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.15 Binary turtle graphics – version 1

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 15-14). The point (0,0) is located in the bottom left corner of the image. The second word contains the X (bits x9-x0) and Y (bits y5-y0) coordinates of the new position.

Table 1. The first word of the *set position* command.

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

Table 2. The second word of the *set position* command.

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

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

Table 3. The *set direction* command.

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

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	right
01	up
10	left
11	down

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can't leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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

Set pen state command

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits r3-r0 are the most significant bits of the 8-bits red component of the color (remaining bits are set to zero). Bits g3-g0 are the most significant bits of the 8-bits green component of the color (remaining bits are set to zero). Bits b3-b0 are the most significant bits of the 8-bits blue component of the color (remaining bits are set to zero).

Table 6. The *pen state* command.

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

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen raised (up)
1	pen lowered (down)

Input

- binary file containing 16/32-bits turtle commands
- file name: “input.bin”

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “output.bmp”

References:

[1] BMP file format – see section 4.2
[2] “**Turtle graphics**”, https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
    http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.16 Binary turtle graphics – version 2

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 1-0) and Y (bits y5-y0) coordinate of the new position. The second word contains the X (bits x9-x0) coordinate of the new position. The point (0,0) is located in the bottom left corner of the image.

Table 1. The first word of the *set position* command.

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**

Table 2. The second word of the *set position* command.

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

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

Table 3. The *set direction* command.

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

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	Right
01	Up
10	Left
11	Down

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can't leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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**

Set pen state command

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits r3-r0 are the most significant bits of the 8-bits red component of the color (remaining bits are set to zero). Bits g3-g0 are the most significant bits of the 8-bits green component of the color (remaining bits are set to zero). Bits b3-b0 are the most significant bits of the 8-bits blue component of the color (remaining bits are set to zero).

Table 6. The *pen state* command.

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**

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen raised (up)
1	pen lowered (down)

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: "output.bmp"

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
    http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.17 Binary turtle graphics – version 3

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 15-14) and Y (bits y5-y0) coordinate of the new position. The second word contains the X (bits x9-x0) coordinate. The point (0,0) is located in the bottom left corner of the image.

Table 1. The first word of the *set position* command.

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	-	-

Table 2. The second word of the *set position* command.

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	-	-	-	-	-	-

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

Table 3. The *set direction* command.

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

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	right
01	up
10	left
11	down

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can't leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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	-	-	-	-

Set pen state command

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits c2-c0 select one of the predefined colors from the color table.

Table 6. The *pen state* command.

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

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen raised (up)
1	pen lowered (down)

Table 8. Color table.

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

Input

- binary file containing 16/32-bits turtle commands
- file name: “input.bin”

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “output.bmp”

References:

[1] BMP file format – see section 4.2
[2] “**Turtle graphics**”, https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.18 Binary turtle graphics – version 4

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 1-0) and X (bits x9-x0) coordinate of the new position. The second word contains the Y (bits y5-y0) coordinate of the new position. The point (0,0) is located in the bottom left corner of the image.

Table 1. The first word of the *set position* command.

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**

Table 2. The second word of the *set position* command.

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
-	-	-	-	-	-	-	-

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

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**

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	Right
01	Up
10	Left
11	Down

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can't leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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**

Set pen state command

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits c2-c0 select one of the predefined colors from the color table.

Table 6. The *pen state* command.

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**

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen raised (up)
1	pen lowered (down)

Table 8. Color table.

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

Input

- binary file containing 16/32-bits turtle commands
- file name: “input.bin”

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “output.bmp”

References:

[1] BMP file format – see section 4.2
[2] “**Turtle graphics**”, https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.19 Binary turtle graphics – version 5

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 15-14). The point (0,0) is located in the bottom left corner of the image. The second word contains the X (bits x9-x0) and Y (bits y5-y0) coordinates of the new position.

Table 1. The first word of the *set position* command.

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

Table 2. The second word of the *set position* command.

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

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

Table 3. The *set direction* command.

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

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	up
01	left
10	down
11	right

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can't leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits r3-r0 are the most significant bits of the 8-bits red component of the color (remaining bits are set to zero). Bits g3-g0 are the most significant bits of the 8-bits green component of the color (remaining bits are set to zero). Bits b3-b0 are the most significant bits of the 8-bits blue component of the color (remaining bits are set to zero).

Table 6. The *pen state* command.

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

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen lowered (down)
1	pen raised (up)

Input

- binary file containing 16/32-bits turtle commands
- file name: “input.bin”

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: “output.bmp”

References:

[1] BMP file format – see section 4.2
[2] “**Turtle graphics**”, https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.20 Binary turtle graphics – version 6

In computer graphics, turtle graphics are vector graphics using a relative cursor (the "turtle") upon a Cartesian plane. The turtle has three attributes: a location, an orientation (or direction), and a pen. The pen, too, has attributes: color, on/off (or up/down) state [2].

The turtle moves with commands that are relative to its own position, such as "move forward 10 spaces" and "turn left 90 degrees". The pen carried by the turtle can also be controlled, by enabling it or setting its color.

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file [1].

Turtle commands

The length of all turtle commands is 16 or 32 bits. The first two bits define one of four commands (set position, set direction, move, set state). Unused bits in all commands are marked by the – character. They should not be taken into account when the command is decoded.

Set position command

The *set position* command sets the new coordinates of the turtle. It consists of two words. The first word defines the command (bits 15-14) and Y (bits y5-y0) coordinate of the new position. The second word contains the X (bits x9-x0) coordinate. The point (0,0) is located in the bottom left corner of the image.

Table 1. The first word of the *set position* command.

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
-	-	-	-	-	-	-	-

Table 2. The second word of the *set position* command.

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	-	-	-	-	-	-

Set direction command

The *set direction* command sets the direction in which the turtle will move, when a move command is issued. The direction is defined by the d1, d0 bits.

Table 3. The *set direction* command.

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

Table 4. The description of the d1,d0 bits.

Bits d1,d0	Turtle direction
00	up
01	left
10	down
11	right

Move command

The *move* command moves the turtle in direction specified by the d1-d0 bits. The movement distance is defined by the m9-m0 bits. If the destination point is located beyond the drawing area the turtle should stop at the edge of the drawing. It can’t leave the drawing area. The turtle leaves a visible trail when the pen is lowered (bit ud). The color of the trail is defined by the r3-r0, g3-g0, b3-b0 bits.

Table 5. The *move* command.

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	-	-	-	-

Set pen state command

The *pen state* command defines whether the pen is raised or lowered (bit ud) and the color of the trail. Bits c2-c0 select one of the predefined colors from the color table.

Table 6. The *pen state* command.

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
-	-	-	-	-	-	-	-

Table 7. The description of the ud bit.

ud bit	Pen state
0	pen lowered (down)
1	pen raised (up)

Table 8. Color table.

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

Input

- binary file containing 16/32-bits turtle commands
- file name: “input.bin”

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “output.bmp”

References:

[1] BMP file format – see section 4.2
[2] “**Turtle graphics**”, https://en.wikipedia.org/wiki/Turtle_graphics
[3] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip

3.21 Food processor

The input BMP image [1] (24-bit RGB) contains one of three food types from the given set (Table 1). Your task is to recognize the food and print in the console window its three-character code given in Table 1.

The recognition should be performed in three steps:

1. Preparation of the histogram of given color component (table 1). The histogram contains number of pixels for each value of given color component.
2. Based on the histogram the mode[4] should be calculated. Mode of a set of data values is the value that appears most often.
3. Mode ranges of images belonging to different food types are different. So, you can make decision which food type the image should be assigned to.

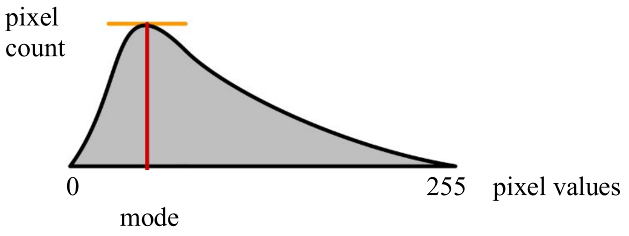


Fig 1. Histogram and mode [4]

Input

- BMP file containing the source image:
  - sub format: 24 bit RGB – no compression,
  - size: width 200px, height up to 200 px,
  - file name: “source.bmp”

Output

- Console window – plain text

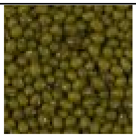
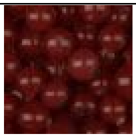
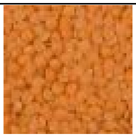
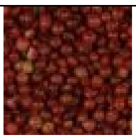
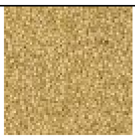
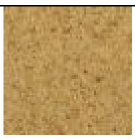
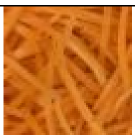
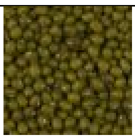
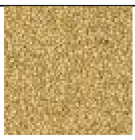
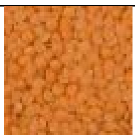
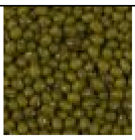
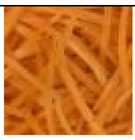
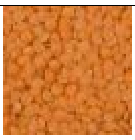
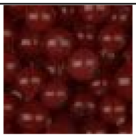
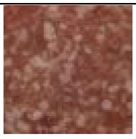
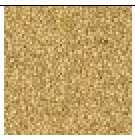
Remarks:

1. Please pay attention to the efficiency of the program (getPixel function is not very efficient)
2. Check the input data and signal errors (e.g. wrong file format)

References:

[1] BMP file format – see section 4.2
[2] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip
[3] RawFooT DB: Raw Food Texture Database,
http://www.ivl.disco.unimib.it/minisites/rawfoot/textures.php
[4] Mode (statistics), https://en.wikipedia.org/wiki/Mode_(statistics)
[5] Hexadecimal file editor, https://hexed.it/

Table 1. Data set description - example images

Set no.	**Food #1**	**Food #2**	**Food #3**	**Recognition based on the histogram of**
1	acb	 adz	 cur	Red component
2	acb	 len	sal	Red component
3	oat	 pep	 qui	Red component
4	 bre	 car	lin	Red component
5	cor	spe	ste	Blue component
6	 adz	pea	 qui	Green component
7	 len	lin	oat	Blue component
8	 adz	chi	pep	Green component
9	 car	 len	pep	Green component
10	bre	cor	spe	Blue component
11	 cur	 sal	ste	Green component
12	lin	pea	 qui	Blue component

3.22 Find image markers

The input BMP image [1] (24-bit RGB) contains markers shown in fig. 1. The image may contain other elements. Your task is to detect all markers of given type.

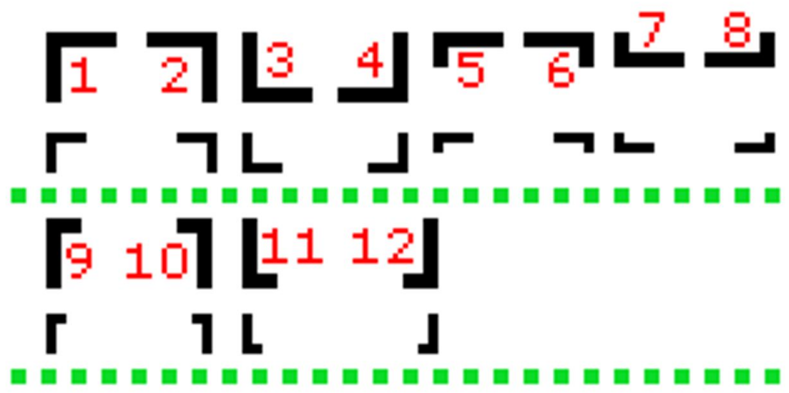
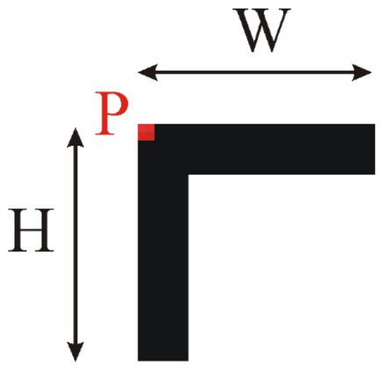


Fig.1. Example markers - red numbers indicate the type of marker.



Rys.2. The marker.

Characteristics of the markers
The markers are black and consist of two *arms* (fig. 2). For a given marker type, the ratio of width W to height H is constant (see table 1). Markers of a given type can be of various sizes if the proportions of the dimensions of the arms are kept. The marker position is determined by the point where the arms intersect (marked by P in Fig. 2).

Table 1. Width to height ratio of different marker types.

Marker type	W/H
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	2
9	½
10	½
11	½
12	½

Input
- BMP file containing the source image:
  - sub format: 24 bit RGB – no compression,
  - size: width 320px, height 240 px,
- file name: “source.bmp”

Output

- Console window – plain text - subsequent lines contain the coordinates of the detected markers, eg. 10, 15. The point (0,0) is in the upper left corner of the image.

Remarks:

1. Check the input data and signal errors (e.g. wrong file format)
2. We assume that the drawing may contain 50 markers (at most) of a given type.
3. A sample file containing markers is available at: http://galera.ii.pw.edu.pl/~zsz/ecoar/images/find_markers/example_markers.bmp

References:

[1] BMP file format – see section 4.2
[2] Example program for bmp reading/writing,
http://galera.ii.pw.edu.pl/~zsz/ecoar/bmp/bmp_mips.zip
[3] Hexadecimal file editor, https://hexed.it/

4 Supplementary materials

This section contains some supplementary materials, which might be useful for your projects.

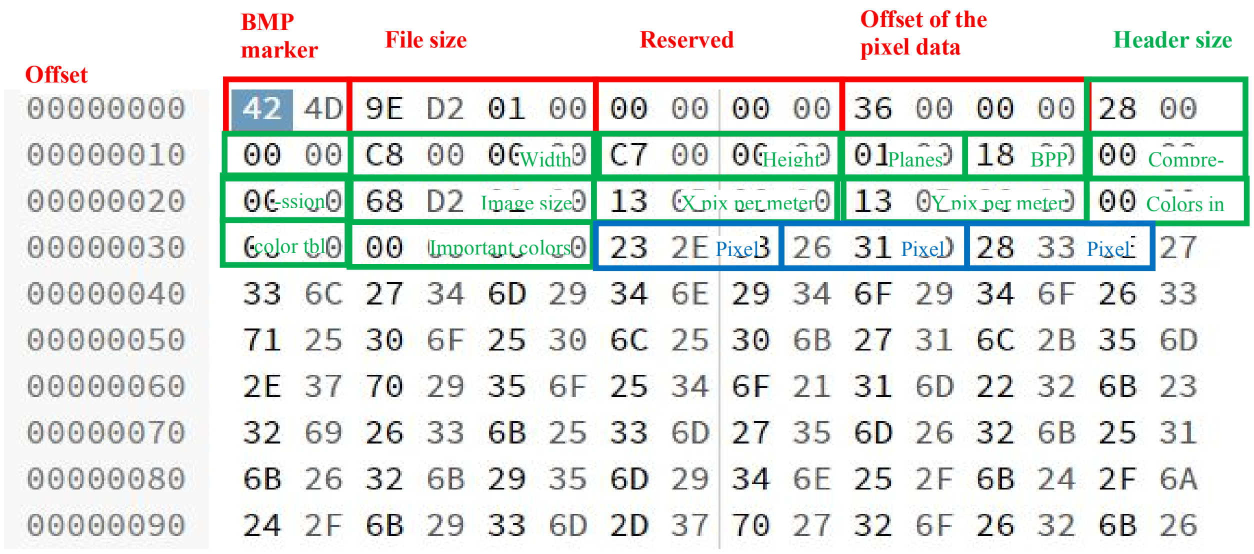
4.1 Reading data from file in Mars

The code below shows how to read data from a file [http://en.wikipedia.org/wiki/SPIM]:

```
#open the file
      li $v0, 13        #system call for file_open
      la $a0, filen     #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

#save the file descriptor
      move $s1, $v0

#check if file was opened correctly (file_descriptor<>-1)
      ...

#read data from file
      li $v0, 14        #system call for file_read
      move $a0, $s1     #move file descr from s1 to a0
      la $a1, buf       #address of data buffer
      li $a2, 4048      #amount to read (bytes)
      syscall

#check how much data was actually read
      beq $zero,$v0, fclose   #branch if no data is read
      ...

#close the file
fclose:
      li $v0, 16              #system call for file_close
      move $a0, $s1           #move file descr from s1 to a0
      syscall
```

4.2 BMP file header

The contents the header of a 24 bits RGB BMP file is shown in fig. 4.1. A hexadecimal file editor (see remark 4 below) may be used to analyze the contents of binary files. The width and height of the image is stored in 4 byte fields at 0x12 and 0x16 offsets. The offset (address) of the first pixel is stored in 32 bits *Offset of the pixel data* field at 0x0A offset. In example below pixel data starts at 0x36.



```
00000000  42 4D 9E D2 01 00 00 00 00 00 36 00 00 00 28 00
00000010  00 00 C8 00 00 00 C7 00 00 00 01 00 18 00 00 00
00000020  00 00 68 D2 01 00 13 0B 00 00 13 0B 00 00 00 00
00000030  00 00 00 00 00 00 23 2E 33 26 31 30 28 33 31 27
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
```

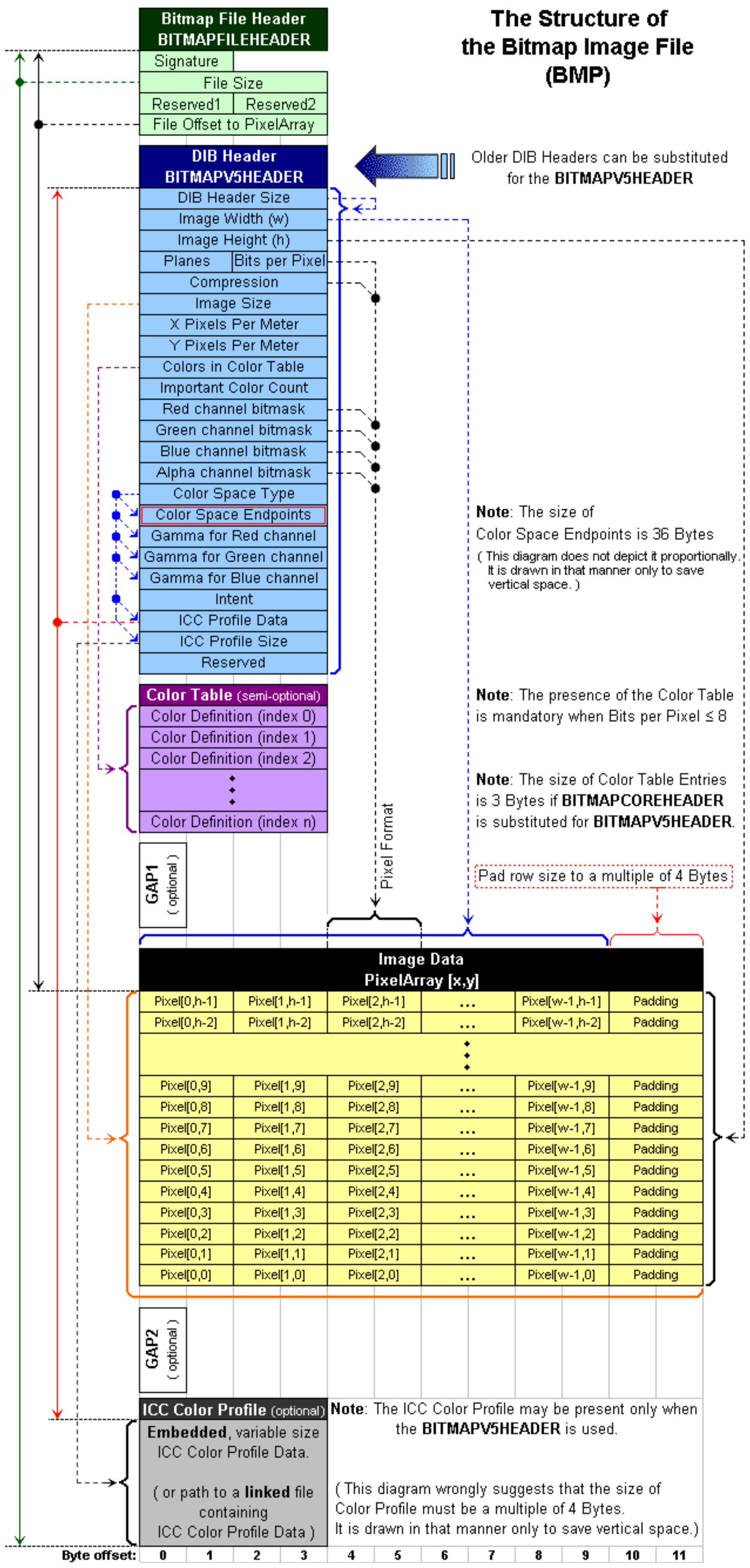
Figure 4.1. Contents of an example BMP file (all numbers in hexadecimal notation).

Remarks:

1. Little endian byte order – the first byte in a field is the least significant one.
2. Order of color components of a pixel: first byte - blue, second byte - green, third byte -red.
3. *Offset of the pixel data* field contains the offset(adress) of the area where the colors of the pixels are stored.
4. Hexadecimal file editor is available at **`https://hexed.it/`**

References:

1. "**file-format-bmp**", https://en.wikipedia.org/wiki/BMP_file_format



Bitmap File Header
BITMAPFILEHEADER
Signature
File Size
Reserved1
Reserved2
File Offset to PixelArray
DIB Header
BITMAPV5HEADER
DIB Header Size
Image Width (w)
Image Height (h)
Planes
Bits per Pixel
Compression
Image Size
X Pixels Per Meter
Y Pixels Per Meter
Colors in Color Table
Important Color Count
Red channel bitmask
Green channel bitmask
Blue channel bitmask
Alpha channel bitmask
Color Space Type
Color Space Endpoints
Gamma for Red channel
Gamma for Green channel
Gamma for Blue channel
Intent
ICC Profile Data
ICC Profile Size
Reserved
Color Table (semi-optional)
Color Definition (index 0)
Color Definition (index 1)
Color Definition (index 2)
Color Definition (index n)
GAP1 (optional)
Pixel Format
Image Data
PixelArray [x,y]
Pixel[0,h-1] Pixel[1,h-1] Pixel[2,h-1] ... Pixel[w-1,h-1] Padding
Pixel[0,h-2] Pixel[1,h-2] Pixel[2,h-2] ... Pixel[w-1,h-2] Padding
Pixel[0,9] Pixel[1,9] Pixel[2,9] ... Pixel[w-1,9] Padding
Pixel[0,8] Pixel[1,8] Pixel[2,8] ... Pixel[w-1,8] Padding
Pixel[0,7] Pixel[1,7] Pixel[2,7] ... Pixel[w-1,7] Padding
Pixel[0,6] Pixel[1,6] Pixel[2,6] ... Pixel[w-1,6] Padding
Pixel[0,5] Pixel[1,5] Pixel[2,5] ... Pixel[w-1,5] Padding
Pixel[0,4] Pixel[1,4] Pixel[2,4] ... Pixel[w-1,4] Padding
Pixel[0,3] Pixel[1,3] Pixel[2,3] ... Pixel[w-1,3] Padding
Pixel[0,2] Pixel[1,2] Pixel[2,2] ... Pixel[w-1,2] Padding
Pixel[0,1] Pixel[1,1] Pixel[2,1] ... Pixel[w-1,1] Padding
Pixel[0,0] Pixel[1,0] Pixel[2,0] ... Pixel[w-1,0] Padding
GAP2 (optional)
ICC Color Profile (optional)
Embedded, variable size ICC Color Profile Data.
(or path to a linked file containing ICC Color Profile Data)
Byte offset: 0 1 2 3 4 5 6 7 8 9 10 11
The Structure of the Bitmap Image File (BMP)
Older DIB Headers can be substituted for the BITMAPV5HEADER
Note: The size of Color Space Endpoints is 36 Bytes
(This diagram does not depict it proportionally. It is drawn in that manner only to save vertical space.)
Note: The presence of the Color Table is mandatory when Bits per Pixel ≤ 8
Note: The size of Color Table Entries is 3 Bytes if BITMAPCOREHEADER is substituted for BITMAPV5HEADER.
Pad row size to a multiple of 4 Bytes
Note: The ICC Color Profile may be present only when the BITMAPV5HEADER is used.
(This diagram wrongly suggests that the size of Color Profile must be a multiple of 4 Bytes. It is drawn in that manner only to save vertical space.)

Fig. 4.2. BMP file structure [https://en.wikipedia.org/wiki/BMP_file_format]

5 Linux, NASM, gcc

This chapter describes the environment for creating and running Intel x86 processor programs using the Linux server located in Institute of Information Technology named galera.ii.pw.edu.pl. The presented configuration of the working environment assumes that the user uses Windows locally on his computer, where the code editor is running, but the project is compiled and started on a remote machine with the Linux system.

5.1 Windows software

Recommended programs to install on Windows are the *PuTTY* terminal, the *WinSCP* file transfer program, and the *Notepad* ++ source code editor.

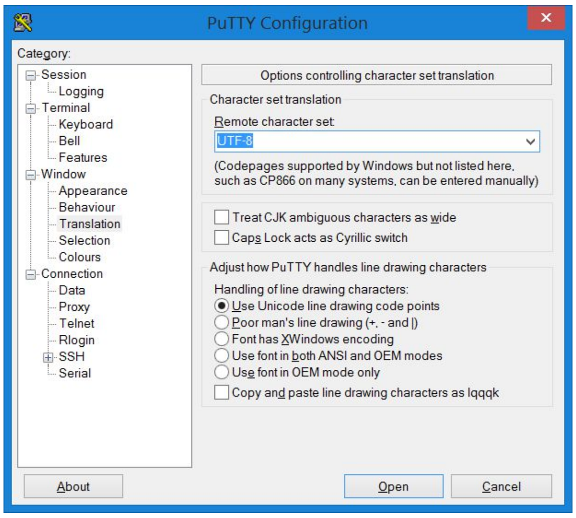
Putty
PuTTY is a terminal program that enables remote operation in command line mode on computers running Unix / Linux. Data transmission between computers is encrypted - SSH (secure shell) protocol is used. Putty installation files can be downloaded from the website:
`https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`

After starting the program, please make sure that UTF-8 character encoding is selected. To do this, in the *Category* panel on the left side of the window, select *Translation* (fig. 5.1). *UTF-8* should be selected in the *Remote character set* drop-down list.

To connect to a remote system, select the *Session* item in the *Category* panel (fig. 5.2). In the *Host Name* field, enter the name / address of the computer you want to connect to, i.e. **`galera.ii.pw.edu.pl`**. Please make sure that the *SSH* radio button belonging to the *Connection type* group is selected. The session settings listed above can be saved by entering the name for the session in the *Saved Sessions* text box e.g. *galera.ii.pw.edu.pl* and by clicking on the *Save* button. Connection is initiated after clicking the *Open* button.

The first time you connect to a computer, the *PuTTY* security alert window appears (Figure 5.3). You should click on the *Yes* button, confirming that the *PuTTY* program should trust the computer to which you are connecting.

When the *login as:* message appears, enter the username and press *enter*. When the *password:* message appears, enter the password (please note that no characters will appear on the screen as you enter the password). When the connection is established, the terminal window should look like the one in figure 5.4.



Options controlling character set translation

Character set translation

Remote character set: UTF-8

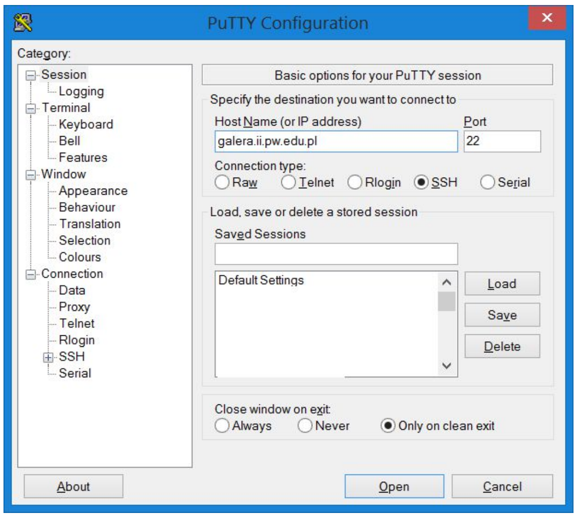
(Codepages supported by Windows but not listed here, such as CP866 on many systems, can be entered manually)

- [ ] Treat CJK ambiguous characters as wide
- [ ] Caps Lock acts as Cyrillic switch

Adjust how PuTTY handles line drawing characters

Handling of line drawing characters:
- ● Use Unicode line drawing code points
- ○ Poor man's line drawing (+, - and |)
- ○ Font has XWindows encoding
- ○ Use font in both ANSI and OEM modes
- ○ Use font in OEM mode only
- [ ] Copy and paste line drawing characters as lqqqk

Fig. 5.1. Putty - character encoding selection.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): galera.ii.pw.edu.pl Port: 22

Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

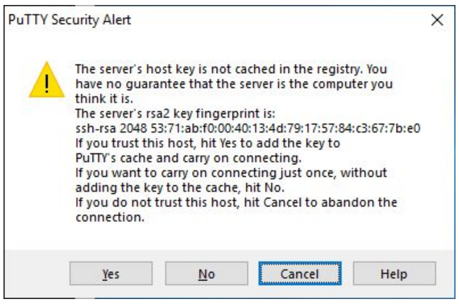
Load, save or delete a stored session

Saved Sessions

Default Settings

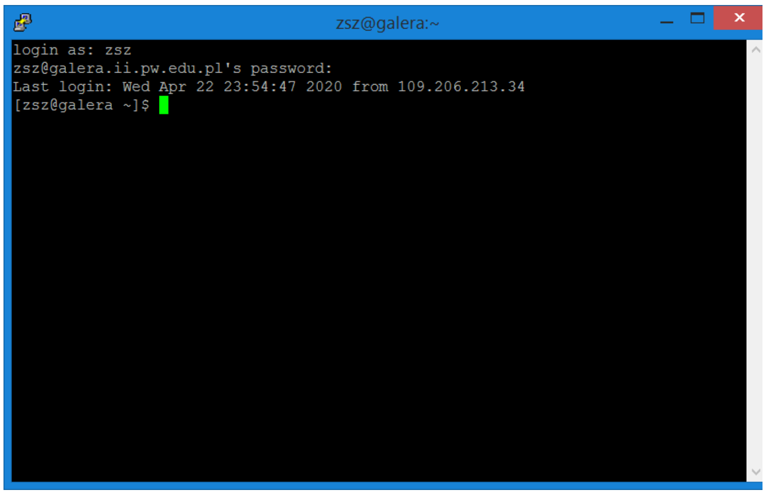
Close window on exit: ○ Always ○ Never ● Only on clean exit

Fig. 5.2. Putty – session parameters.



The server's host key is not cached in the registry. You have no guarantee that the server is the computer you think it is.
The server's rsa2 key fingerprint is:
ssh-rsa 2048 53:71:ab:f0:00:40:13:4d:79:17:57:84:c3:67:7b:e0
If you trust this host, hit Yes to add the key to PuTTY's cache and carry on connecting.
If you want to carry on connecting just once, without adding the key to the cache, hit No.
If you do not trust this host, hit Cancel to abandon the connection.

Fig. 5.3. PuTTY Security Alert window



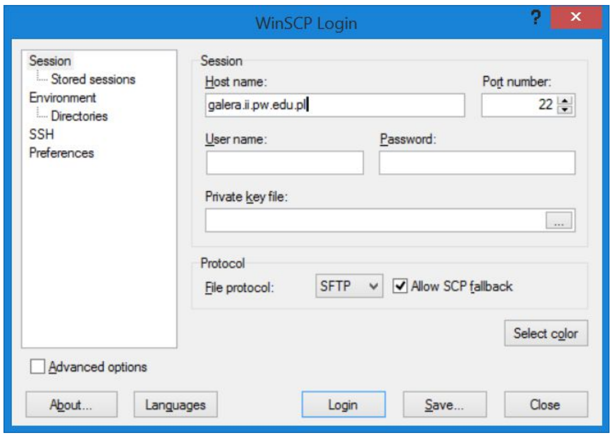
```
login as: zsz
zsz@galera.ii.pw.edu.pl's password:
Last login: Wed Apr 22 23:54:47 2020 from 109.206.213.34
[zsz@galera ~]$
```

Fig. 5.4. Terminal window after successful connection.

Using the Linux command line will be briefly discussed in the following sections.

WinScp

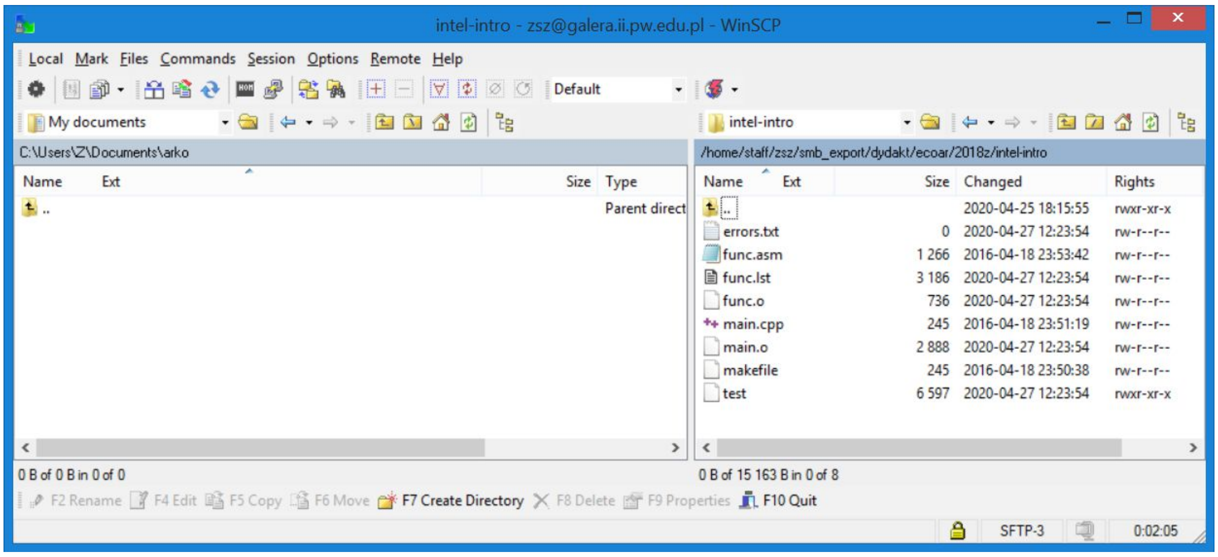
WinScp enables encrypted file transfers between a Windows computer and a Unix / Linux computer. The program installation files can be downloaded from:
`https://winscp.net/eng/download.php`
After starting the program, select *Session* in the panel on the left side of the window. Then enter the name / address of the computer you want to connect to in the *Host name* field. In our case it is **`galera.ii.pw.edu.pl`**.
User name and *password* fields may remain empty. The user will then be asked to enter this data when establishing a connection. Session settings can be saved by clicking the *Save…* button and then entering the name for the defined session e.g. *galera.ii.pw.edu.pl*. Connection is initiated after clicking the *Login* button.



Session
Host name: galera.ii.pw.edu.pl
Port number: 22
User name:
Password:
Private key file:
Protocol
File protocol: SFTP
[✓] Allow SCP fallback

Fig. 5.5. WinSCP – session parameters.

ter establishing the connection, the WinScp window should look similar to the one in figure 5.6. It consists of two panels. The left one displays the contents of the folder from the local computer, and the right one from the remote computer to which you have connected. Files and folders can be transferred by dragging between WinSCP panels or by dragging between the windows of the system File Explorer and the remote computer panel.


intel-intro - zsz@galera.ii.pw.edu.pl - WinSCP
Local Mark Files Commands Session Options Remote Help
Default
My documents
intel-intro
F2 Rename F4 Edit F5 Copy F6 Move F7 Create Directory F8 Delete F9 Properties F10 Quit
SFTP-3
0:02:05

C:\Users\Z\Documents\arko

Name	Ext	Size	Type
..			Parent direct

0 B of 0 B in 0 of 0

/home/staff/zsz/smb_export/dydakt/ecoar/2018z/intel-intro

Name	Ext	Size	Changed	Rights
..			2020-04-25 18:15:55	rwxr-xr-x
errors.txt		0	2020-04-27 12:23:54	rw-r--r--
func.asm		1 266	2016-04-18 23:53:42	rw-r--r--
func.lst		3 186	2020-04-27 12:23:54	rw-r--r--
func.o		736	2020-04-27 12:23:54	rw-r--r--
main.cpp		245	2016-04-18 23:51:19	rw-r--r--
main.o		2 888	2020-04-27 12:23:54	rw-r--r--
makefile		245	2016-04-18 23:50:38	rw-r--r--
test		6 597	2020-04-27 12:23:54	rwxr-xr-x

0 B of 15 163 B in 0 of 8

Fig. 5.6. WinScp window.

Notepad++ and the NppFTP plugin
Notepad++ is a source code editor that supports syntax highlighting in many programming languages. The *NppFTP* plugin allows you to edit files located on remote machines running Unix / Linux systems without having to download them first e.g. using the *WinScp*. The program can be downloaded from the:
`https://notepad-plus-plus.org/downloads/`
Using the *NppFTP* plugin is possible after configuring connection parameters. You must first open the plugin window by selecting *NppFTP* from the *Plugins* menu, and then *Show NppFTP Window* (fig. 5.7). The *Profile settings* command (fig. 5.8) opens the window for entering the name of the computer to which the connection should be made, as well as the username and password. In this window, click the *Add new* button. The *Adding profile* window will appear, in which you must enter the profile name (e.g. *galera.ii.pw.edu.pl*) and press *OK*. Then in the *Profile settings* window (Fig. 5.9) enter in the *hostname* field the name of the computer to which you want to connect (**`galera.ii.pw.edu.pl`**). From the *Connection type* list, select *SFTP*, and enter the username in the *username* field. Checking the

ask for password box will cause the program to prompt for a password when establishing a connection. The settings window will close after pressing the *Close* button.

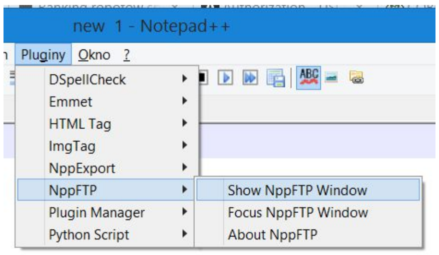


Fig. 5.7. Enabling the NppFTP plugin window.

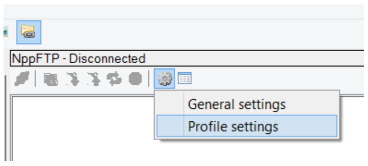
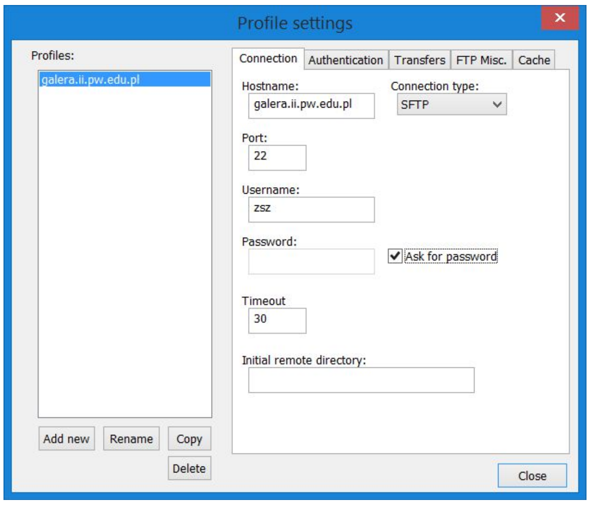


Fig. 5.8. NppFTP plugin settings.



Profile settings

Profiles: galera.ii.pw.edu.pl

Connection | Authentication | Transfers | FTP Misc. | Cache

Hostname: galera.ii.pw.edu.pl

Connection type: SFTP

Port: 22

Username: zsz

Password:

☑ Ask for password

Timeout: 30

Initial remote directory:

Add new | Rename | Copy | Delete | Close

Fig. 5.9. Connection profile configuration.

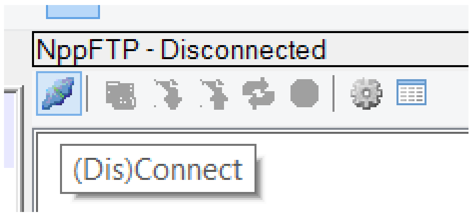
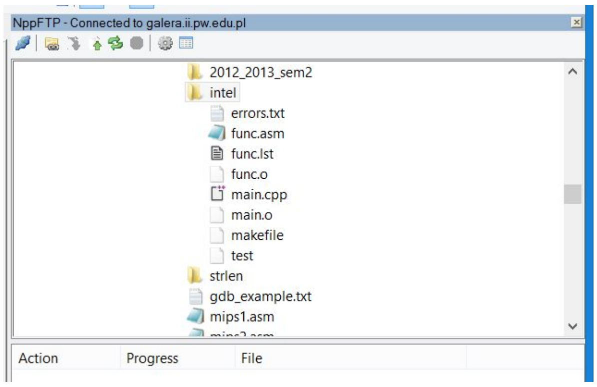


Fig. 5.10. (Dis)Connect button.

The connection is established by clicking on the *(Dis)Connect* button (Fig. 5.10) and selecting the name of the connection profile (in the example above it is galera.ii.pw.edu.pl). A list of files on the remote computer will appear in the plug-in window (Fig. 5.11). Double-clicking on the file name will open the file in the editing window. Further work with the file is the same as with the file saved locally on the computer.



NppFTP - Connected to galera.ii.pw.edu.pl

```
2012_2013_sem2
intel
    errors.txt
    func.asm
    func.lst
    func.o
    main.cpp
    main.o
    makefile
    test
strlen
gdb_example.txt
mips1.asm
```

Action | Progress | File

Fig. 5.11. File list of the remote computer.

5.2 Basic Linux commands

When working remotely with Linux using *Putty* (or another SSH client), a command line interface is used in which commands are entered in the form of text from the keyboard. Each command is confirmed by pressing the ENTER key.

Linux file names are case-sensitive. A common error for novice users is creating files (e.g. document.txt) and then referring to these files e.g. from their own programs using a different case (e.g. Document.txt).

Basic concepts:

- current directory – each file name, which is not preceded by information about the directory in which the file is located, refers to the file in the so-called current directory. In other words, the current directory is the reference point when locating files on a Linux file system. Each terminal session has its current directory. The current directory is marked by a period „`.`".
- parent directory – it is a directory one level higher in the file system hierarchy relative to the given directory. The parent directory is marked with two dots „`..`".
- home directory - it is a directory designated by the system administrator to store the user's files. The home directory is marked with a tilde sign „`~`".
- the absolute (full) path to the file (or directory) uniquely identifies the location of the file or directory in the file system. The path includes the names of subsequent directories (separated by the `/` character), starting from the root directory, which you must enter to locate the file. The **`/home/users/xgucio/prog.cpp`** path means that the root directory `/` has a *home* subdirectory, and the *users* subdirectory in it. *xgucio* subdirectory is located in *users* directory and contains the *prog.cpp* file.

Figure 5.12 shows the components of the command line. From the left, these are the username, the name of the computer the user is connected to (important information for remote work) and the last component of the full path to the current directory.

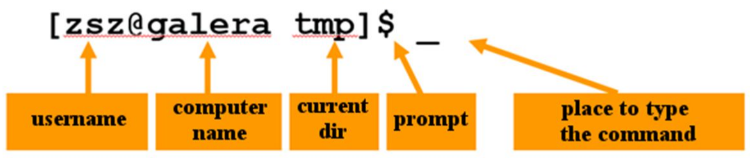

```
[zsz@galera tmp]$ _
```

Figure 5.12. Components of the command line.

This section describes some of the basic Linux commands. It should be noted that these commands have significantly more options than discussed below. A full description of the commands is available in the system help run by the command: **`man`** ***`command_name`***

5.2.1 passwd

The passwd command allows the user to change the password. The initial password is difficult to remember because it was machine generated - it is worth changing it. Please invent a new password - not shorter than 6 characters, containing at least one capital letter or number or special character. The password cannot contain the account name (login) or the name of the owner.

Passwords for Linux systems of computer lab of Institute of Computer Science are stored on the *galera.ii.pw.edu.pl* server, so to change the password, you need to start a remote ssh session to this server, e.g. using the Putty program (see section 5.1). After logging in, run the passwd command and enter the new password twice. In the terminal window, issue the commands shown in Fig. 4.13 as bold text (NOTE: replace the example jkowalsk login with your own login).

```
[jkowalsk@galera ~]$ passwd    { Linux command to change password }
New UNIX password:             { enter a new password - nothing appears, it's normal }
Retype new UNIX password:      { repeat the new password }
Changing password for user jkowalsk.
passwd: all authentication tokens updated successfully.
[jkowalsk@galera ~]$ exit      { end of remote ssh session }
```

Fig. 5.13. Changing the password on Linux system.

The changed password will be automatically propagated from the galley server to all Linux PCs in the laboratory (max. delay: 3 minutes).

5.2.2 ls

The **`ls`** command lists the files and subdirectories of the current directory. The **`ls -al`** command (with the -al parameter added, the parameters are separated from the command name by a space) displays lines similar to one shown in Figure 5.14.

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

Fig. 5.14. Sample output of the ls command

The first column containing **`-rw-r--r`** describes the access rights to the file whose name is given in the last column (in the example from the figure, this is **`sensortagcollector.py`**). User and group the file belongs to are described by **`pi pi`** (user **`pi`** group **`pi`**). The file size is **`11531`** bytes and the last modification date was **`May 22 2018`**.

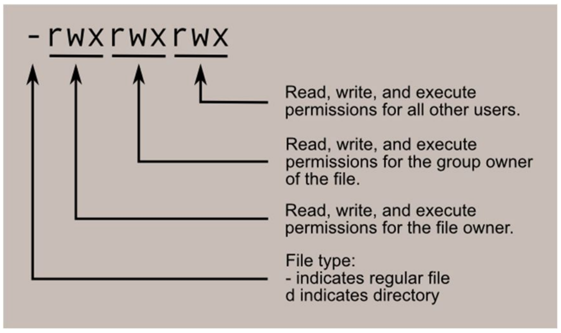

- rwx rwx rwx
Read, write, and execute permissions for all other users.
Read, write, and execute permissions for the group owner of the file.
Read, write, and execute permissions for the file owner.
File type:
- indicates regular file
d indicates directory

Fig. 5.15. File access rights on Unix systems [http://linuxcommand.org/lc3_lts0090.php].

File permissions on Unix / Linux systems are described in Figure 4.15. There are separate permissions for the file owner, for users belonging to one indicated group and for all other system users. A detailed description of the permissions can be found in [http://linuxcommand.org/lc3_lts0090.php].

5.2.3 pwd

The **`pwd`** (print working directory) command displays the full absolute path to the current directory. Just after logging in and running the pwd command, the displayed message should be like one in figure 5.16 (change xgucio to your user name) - this is the path to the user's home directory.

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

Fig. 5.16. Sample output of the *pwd* command.

5.2.4 cd

The **`cd`** command changes the current working directory. This command will set the current directory to the user's home directory if it is issued without any parameters. If a directory path is given as the parameter after the command name, it will become the current directory. E.g. **`cd /usr/bin`** will change the current directory to /usr/bin, **`cd ..`** will change the current directory to the parent directory of the current directory.

5.2.5 ps

The term *process* refers to an instance of a running program. The process includes program instructions, open files, input and output streams [https://www.tecmint.com/linux-process-management/]. There are two types of processes on Linux.

Foreground processes (also known as interactive processes) - they are initiated and controlled through a terminal session. In other words, a user must be connected to the system to start such processes; they did not start automatically as part of system functions / services.

Background processes (also called non-interactive / automated processes) - are processes not connected to the terminal; they do not expect user interaction.

There are many Linux tools for viewing / displaying running processes in the system, two traditional and well-known commands are **`ps`** and **`top`**. The **`ps -ef`** command displays all processes running in the system - Figure 5.17 shows two sample lines from the command execution.

```
...
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 ...
...
```

Fig. 5.17. Output of the *ps -ef* command.

The first column contains the username (owner) of the process. Linux is a multi-user system - this means that different users can run different programs on the system. Thus, each running instance of the program must be uniquely identified by the kernel. Each process is identified by its process identifier (PID), which is shown in the second column. The end of each line is the command that was used to start the process.

5.2.6 kill

Kill is a computer program in Unix and Linux systems that allows you to send signals to processes running in the operating system. By default, kill sends a signal to the process instructing it to exit. Contrary to the name suggesting immediate termination of work, the command closes the process correctly and preserves all internal data.

An example of using the kill command to end the first of the processes shown in Figure 5.17 (the process is identified by the PID number, which in this case is 2150): **`kill 2150`**

If the process does not respond to the standard use of the kill command, you can use the kill command with the -9 parameter. This will unconditionally end the process. In this case, however, data may be lost. An example of using the kill command to unconditionally terminate a process: `kill -9 2150`

5.2.7 mkdir

The mkdir command is used to create a new directory. The parameter is the name of the newly created directory. An example of using the command to create a directory named *project1* in the current directory: `mkdir project1`

5.2.8 rm

The **`rm`** command is used to delete files and directories. When deleting, you will be asked to confirm the deletion. To avoid confirmation questions, use the **`-f`** parameter. Example of using the **`rm`** command to remove the *main.cpp* file in the current directory:

```
rm -f main.cpp
```

An example of using the **`rm`** command to remove the project1 directory (including the content) in the current directory: `rm -rf project1`

5.2.9 cp

The **`cp`** command is used to copy files and directories. The command can be used in several different ways. The general usage can be represented as follows:

```
cp source destination
```

If the source is a file name and the destination is the name of an existing directory, the file will be copied to the specified directory. If the source is a file name and the destination is not a directory name, then the contents of the source file will be copied to the file specified as the destination.

If the directory and its contents are to be copied, use the cp command with the **`-r`** parameter representing recursive copying. The following example copies the directory named *project1* to the *archive* directory located in the parent directory:

```
cp -r project1 ../archive
```

5.2.10 mv

The **`mv`** command is used to move or rename files and directories. The general method of calling can be represented as follows:

```
mv source destination
```

If the source is the name of a file or directory, and the target is the name of an existing directory, then the source will be moved to the indicated directory. If the source is a file name and the destination is not a directory name, then the source file will be moved to the file specified as the destination.

5.2.11 tar

The **`tar`** program is a Unix program for placing a set of files in one uncompressed file (so-called archive). It can then be compressed, e.g. with the **`gzip`** program automatically launched by the tar program. Compressed archives usually have the extension *.tar.gz* or *.tgz*.

The **`tar`** program syntax for creating an archive is as follows:

```
tar -zcf archive_file.tar.gz source1 source2...
```

The *-z* parameter is responsible for compressing the archive with **`gzip`**. The *-c* parameter indicates the archive creation mode, followed by the name of the resulting file. *Source1*, *source2* etc. are the names of the files or directories to be archived.

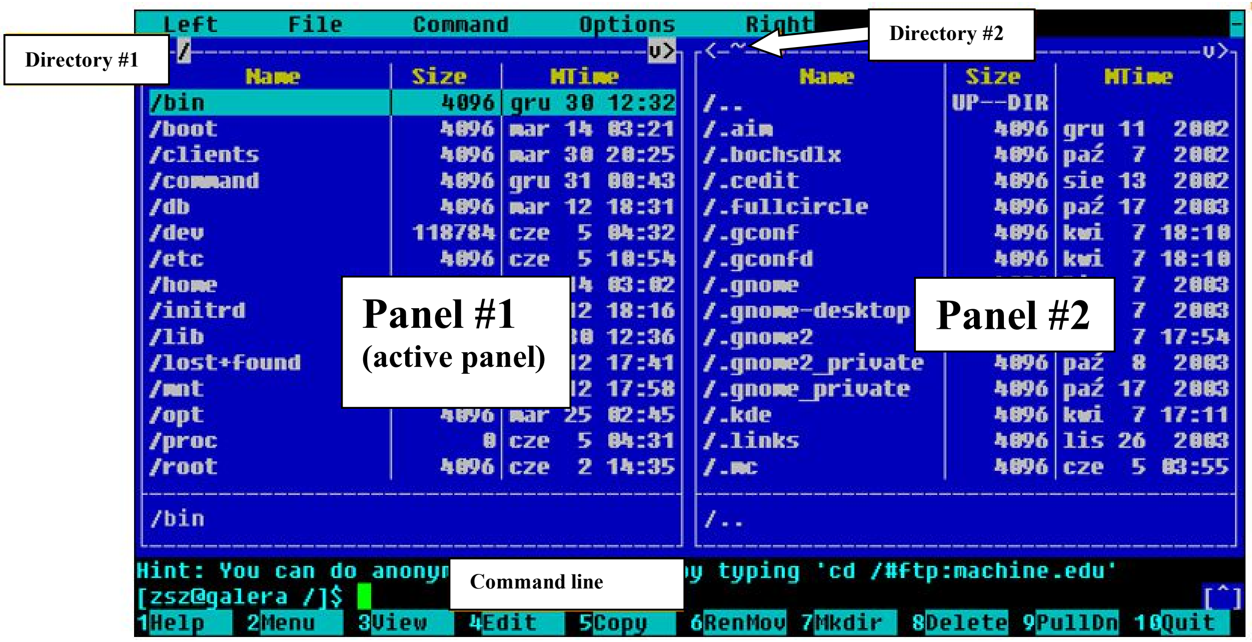
The **`tar`** program syntax for unpacking the archive is as follows:

```
tar -zxf plik_archiwum.tar.gz
```

The *-x* parameter indicates the archive unpacking mode. The name of the file to unpack is given after the *-f* parameter. The archive is unpacked into the current directory.

5.2.12 mc

File operations done from the command line can be time consuming and non-intuitive for a novice user. The Midnight Commander program provides a semigraphic user interface in text mode (Figure 5.18).



```
  Left       File       Command      Options      Right
 /-----------------------------------v> <-~----------------------------v>
        Name         Size     MTime            Name        Size     MTime
/bin                 4096 gru 30 12:32   /..              UP--DIR
/boot                4096 mar 14 03:21   /.aim               4096 gru 11  2002
/clients             4096 mar 30 20:25   /.bochsdlx          4096 paź  7  2002
/command             4096 gru 31 00:43   /.cedit             4096 sie 13  2002
/db                  4096 mar 12 18:31   /.fullcircle        4096 paź 17  2003
/dev               118784 cze  5 04:32   /.gconf             4096 kwi  7 10:10
/etc                 4096 cze  5 10:54   /.gconfd            4096 kwi  7 10:10
/home                          14 03:02   /.gnome                        7  2003
/initrd                        12 18:16   /.gnome-desktop                7  2003
/lib                           30 12:36   /.gnome2                       7 17:54
/lost+found                    12 17:41   /.gnome2_private    4096 paź  8  2003
/mnt                           12 17:58   /.gnome_private     4096 paź 17  2003
/opt                 4096 mar 25 02:45   /.kde               4096 kwi  7 17:11
/proc                   0 cze  5 04:31   /.links             4096 lis 26  2003
/root                4096 cze  2 14:35   /.mc                4096 cze  5 03:55

/bin                                     /..
Hint: You can do anony                   y typing 'cd /#ftp:machine.edu'
[zsz@galera /]$                                                          [^]
1Help  2Menu  3View  4Edit  5Copy  6RenMov 7Mkdir  8Delete 9PullDn 10Quit
```

Fig. 5.18. User interface of the Midnight Commander (mc).

The program displays on the screen the contents of two directories in two panels. Directory access paths are displayed on the upper edge of the frame surrounding the panels. One panel is an active panel - in it you can browse the list of files using the up and down arrows on the keyboard and select files and directories by pressing the *insert* button (selected items are yellow).
The directory can be entered by setting the blue highlight on its name and pressing the *enter* key (you can go to the parent directory by selecting ..). The *tab* key allows you to change the active panel. The highlighted files can be copied or moved from the directory displayed in the active panel to the directory whose content is shown in the second panel by pressing *F5* or *F6* respectively. Pressing the *F7* key creates a new directory. Deleting the highlighted element or group of elements marked yellow is done by pressing the *F8* key. The upper menu is activated by pressing *F9* and the *F10* key exits the program.

Midnight Commander also provides a command line where the user can enter any command. If the running command displays messages on the screen, they will be hidden by panels after the operation is completed. To see them again, press *CTRL+O*. The user interface elements of the mc program will then be completely hidden (but the program is still running in the background). They will be displayed again after pressing *CTRL+O* for the second time.

F3	Text file viewer	**F6**	Move file(s)	**F9**	Upper menu
F4	Text file editor	**F7**	Create directory	**F10**	Exit program
F5	Copy file(s)	**F8**	Delete file(s)	**Insert**	Select file
CTRL O	Show/hide panels	**TAB**	Switch panel		

Fig. 5.19. Keyboard shortcuts of Midnight Commander (mc).

The program also has a browser and a text file editor that is activated respectively by the *F3* and *F4* keys (after placing the highlight on the file name). Figure 5.19 shows the most commonly used keyboard shortcuts of the mc program.

5.3 Example program

The Intel assembly programming will be presented on the example of the program shown in Listings 5.1, 5.2. It is available for download at:

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

The file should be downloaded to the home directory on the Galera server, e.g. by downloading the file with a browser to your own computer and sending to the home directory with the WinScp program or in the command line in the terminal with the command:

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

Then, unpack the compressed .tar.gz archive with the command

```
tar zxf intel-intro.tar.gz
```

After unpacking, an *intel-intro* directory will be created containing three files:

- main.cpp – a file containing the *main* function in C ++ from which the *func* assembler function is called;
- func.asm – assembly language function called from the main program in C ++;
- makefile – a file to automate the compilation process.

To compile and run the program, change the current directory to *intel-intro*:

```
cd intel-intro
```

Then compile the project using the *make* command which uses the configuration saved in the *makefile*:

```
make
```

The last step is to run the program. The executable file is named test, so the program starts with the command:

```
./test
```

5.3.1 Discussion of the sample program

The *main.cpp* file contains a call to the external *func* function defined in the *func.asm* file. Declaration `extern "C" int func(char *a);` causes the function name to be linked according to C language conventions (this means, among other things, that the compiler does not modify the function name during the compilation process).

The assembler file contains only one section - it is the *text* section in which the program code is located. The `global func` directive makes the *func* symbol visible to external modules and can be used by the linker.

Comments in the assembler file start with a semicolon *;*. The content of the lines from the semicolon to the end of the line is ignored by the compiler.

Addressing stack elements is done using the stack frame pointer located in the *ebp* register. After entering the function, one should preserve the contents of the stack frame pointer of the caller, i.e. save the *ebp* register on the stack. The `push ebp` instruction is responsible for this. When exiting the function, you must restore the old contents of the ebp register.

Then a new value of the stack frame pointer is initiated in the ebp register - it is the current value of the stack pointer (indicating the address of the element at the top of the stack). The `mov ebp, esp` instruction copies the contents of the *esp* register to the *ebp* register. For assembler instructions having two parameters, the first parameter is the target parameter and the second is the source parameter.

The discussed function has one parameter – a pointer to the beginning of the character string. It is located at *EBP+8*. A schematic drawing of the contents of the stack is placed at the end of Listing 5.2. It shows that the stack grows towards smaller addresses and when a function is called then first the function parameters are placed (in this case, *char *a*), the return address, and then in the called function the value of stack frame pointer of the calling function is stored, as described in previous paragraph.

Listing 5.1. Source code of the *main.cpp* file.

```
#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 5.2. Source code of the *func.asm* file.

```
…
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
;
;
;============================================
```

The addresses are 32 bits, hence the shift of 8 bytes when accessing the second element relative to the stack frame pointer. The `mov eax, DWORD [ebp+8]` instruction copies the double word data (*DWORD* - 4 bytes) stored in the memory at *ebp+8* to the *eax* register. Square brackets indicate that the address is in the register and the data is being copied in memory. After executing the instruction, the *eax* register contains the address of the character string given as the function parameter.

In the next line of the program `mov BYTE [eax],'w'` the ASCII code of lowercase letter *w* is inserted at the first position of the string. The *BYTE* size specifier is responsible for generating the correct processor instruction, which transfers only one byte to memory.

The value returned by the function should be placed in the *eax* register. The example function always returns the value zero: `mov eax, 0`. Note that a better way to zero a register is to use the *xor* instruction.

The `pop ebp` instruction removes the value from the stack and places it in the *ebp* register. This restores the contents of the caller's stack frame pointer. The `ret` instruction picks the return address from the stack. Execution of the program will continue from the instructions at this address (the next instruction after the *call* instruction).

5.3.2 Description of the *makefile*

Listing 5.3. The *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
```

Listing 5.3 shows the *makefile* responsible for configuring the compilation and linking process. It consists of three steps. First, assembly is carried out using the *nasm* program. The parameters of the program are:

- `-o` defines the resulting name of the object file after assembly (*func.o*);
- `-f` defines the format of the .o file (*elf*);
- `-g` forces the generation of information for the debugger;
- `-l` enables generation of listing (in the *func.lst* file) containing machine code in hexadecimal form;
- the last parameter is the name of the file being assembled (*func.asm*).

The *main.cpp* file is compiled using the *g++* program. The meaning of theparameters:

- `-m32` causes 32 bit code to be generated;
- `-c` causes the file specified as the last parameter to be compiled;
- `-g` forces the generation of information for the debugger;
- `-O0` disable optimization when compiling code;

Compiler messages and diagnostic information are redirected to a file called *errors.txt* (they will not appear on the screen).

Linking is done by calling *g++*. The *-o* parameter is responsible for the name of the generated executable file (*test*). At the end of the command there is a list of object files to be linked by the linker into one executable program.

5.3.3 Enhancement of the sample program

The sample program will be modified in such a way that the function *func* converts all lowercase letters *a* in the string (given as the parameter of the function) to the asterisk *. The value returned by the function should be the number of substitutions made.

As written in chapter 1.3, before implementing the assembler code, it is worth writing (pseudo) code in C language and to encode assembler instructions on this basis. Listing 5.4 shows the C code for the modified function. The reader should analyse it by him/her/self. Assembler code created on the basis of C function is shown in Listing 5.5.

Listing 5.4. 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;
}
```

Before writing the code from Listing 5.5, following design decisions were made:

- The address of the current character will be stored in the *eax* register;
- The number of changed characters will be stored in the *ecx* register;
- The value of the current character (loaded from memory) will be stored in the *bl* register.

Listing 5.5. Modified *func* function.

```
…
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
                                 ;{
         cmp bl,'a'              ;if (*a=='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
```

The **`xor ecx, ecx`** statement initializes the *replace_count* variable – it zeros the value in the *ecx* register. The **`mov bl,[eax]`** instruction is the first statement of the while loop. It loads one byte (one character) from memory at the address contained in the *eax* register and it stores it in the *bl* register. Square brackets around the *eax* register indicate the use of indirect register addressing mode – it means that the argument is in memory and its address in the register.

The **`cmp bl,0`** instruction is a comparison instruction that sets the appropriate processor flags. The conditional jump **`je replace_exit`** is performed (based on the flag values) to the *replace_exit* label, if the register *bl* is zero (this means that the end of the string has been reached). Otherwise, the program goes to the next instruction.

The **`cmp bl,'a'`** comparison instruction examines whether the ASCII code of the character in the *bl* register is equal to the code of the lowercase letter *a*. If not, then a jump is performed to the *next_char* label. Otherwise, the current character is modified to * - the **`mov BYTE [eax],'*'`** instruction saves the ASCII code of the * character at the address contained in the eax register. The **`inc ecx`** instruction increments the counter of changed characters.

The **`inc`** instruction next to the *next_char* label increments the address of current character. The **`jmp replace_loop`** instruction is an unconditional jump to the *replace_loop* label, where the while loop condition is examined.

After exiting the loop, the number of changed characters stored in the *ecx* register is copied to the *eax* register, where the value returned by the function should be placed.

5.4 Example debugging session using *gdb*

You can use the *gdb* debugger to detect errors and run programs step by step. It is a tool with a text interface. This chapter shows an example debug session. Commands entered by the user are highlighted in blue, messages of the analyzed program are highlighted in green.

The parameter of the *gdb* program is the name of the executable file whose operation will be analyzed. The executable file is obtained in the linking phase (see section 5.3.2). The *gdb* program displays the initial messages shown in Figure 5.20.

```
[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".
```

Fig. 5.20. Example use of gdb – part 1.

(gdb) is the debugger prompt, after which you can enter commands. The first command shown in Fig. 5.21 is the command to set the convention in which assembly instructions will be displayed. You can choose from *att* and *intel* convention. In this example *intel* convention was selected.

The *break* command sets a breakpoint on the instruction associated with the func *label*. The debugger confirms setting the breakpoint by displaying its number and address of the instruction on which the breakpoint was set. The program starts after issuing the *run* command. When the program execution has reached the breakpoint, a message is displayed with the breakpoint number and the address, and then the debugger prompt appears.

```
(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 ()
```

Fig. 5.21. Example use of gdb – part 2.

The *print / d* command shown in Figure 5.22 displays the contents of the *eax* and *ecx* registers as signed integers. Available are (among others) the following display formats: */x* - hexadecimal, */d* - decimal (signed integer), */u* - decimal (unsigned integer), */t* - binary, */a* -address.

The *info registers* command displays the contents of registers.

```
(gdb) print/d $eax
$1 = -4719557
(gdb) print/d $ecx
$2 = 0
(gdb) info registers
eax            0xffb7fc3b       -4719557
ecx            0x0      0
edx            0x5e20b0 6168752
ebx            0x5e0ff4 6164468
esp            0xffb7fc18       0xffb7fc18
ebp            0xffb7fc18       0xffb7fc18
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
```

Fig. 5.22. Example use of gdb – part 3.

The program can be executed in the step mode using the *stepi* (fig. 5.23) and *nexti* instructions. The first of them, after encountering a *call* instruction, enters the called function. Second - executes all instructions of the function and stops at the first instruction after the *call* instruction.

```
(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.
```

Fig. 5.23. Example use of gdb – part 4.

The *disassemble* command is used to disassemble the machine code in computer memory into human-readable mnemonics. The command parameter can be, among others function name, label, address range.

The *cont* command shown in Figure 5.24 causes the program to run in continuous mode until it encounters another breakpoint (or program end).

```
(gdb) break loop_exit
Breakpoint 2 at 0x80484e6
(gdb) cont
Continuing.
Breakpoint 2, 0x080484e6 in loop_exit ()
```

Fig. 5.24. Example use of gdb – part 5.

The *disassemble* command (without any parameters) disassembles the code starting from the nearest label. The *quit* command shown in Figure 5.25 is used to quit the *gdb* program.

```
(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
```

Fig. 5.25. Example use of gdb – part 6.

6 Intel projects

The goal of Intel project is to write a function in Intel x86 32 bits assembly language and to call this function from a C/C++ program.

The project should be delivered as a *.zip file containing all source files (cpp, c, asm), make file and a set of related test cases. The test cases should confirm that the program works correctly. The project should contain a directory named *tests*. The sub directories of the directory *tests* correspond to individual tests. They should contain:

- input file(s),
- output file(s),
- a text file (named *description.txt*) containing short description what was being tested and the values of the input parameters (if any).

Grading:

- base version - 32bits program – max 6 points,
- additional version - 64bits – max 2 points.

6.1 Color replacement

Replace color of pixels in an image by corresponding sepia tones. The replacement takes place when inequality (1) is satisfied

$$\boldsymbol{dist} \geq \sqrt{(\boldsymbol{R}-\boldsymbol{R_{sel}})^2+(\boldsymbol{G}-\boldsymbol{G_{sel}})^2+(\boldsymbol{B}-\boldsymbol{B_{sel}})^2} \qquad (1)$$

where (R,G,B) are the values of red, green, blue color components of the pixel, (R_{sel},G_{sel},B_{sel}) are the values of red, green, blue components of selected color (one for the whole image) and *dist* is the size of the color neighbourhood.

The formula for calculating of sepia tone [1] of a pixel:

```
outputRed   = (inputRed * .393) + (inputGreen *.769) + (inputBlue * .189)
outputGreen = (inputRed * .349) + (inputGreen *.686) + (inputBlue * .168)
outputBlue  = (inputRed * .272) + (inputGreen *.534) + (inputBlue * .131)
```

If any of these output values is greater than 255, you set it to 255.

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”
- (R_{sel},G_{sel},B_{sel}) – the values of red, green, blue components (0-255) of the replaced color (input from keyboard)
- *dist* - the size (0-442) of the color neighbourhood (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”

References:

[1] Zach Smith, “**How do I... Convert images to grayscale and sepia tone using C#?**” http://www.techrepublic.com/blog/how-do-i/how-do-i-convert-images-to-grayscale-and-sepia-tone-using-c/
[2] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.2 Shading

Perform smooth shading of a triangle. Shading is the process of altering the color of pixels of a polygon. The colors of the vertices of the triangle are given. To find the R,G,B components of the color of other pixels linear interpolation may be used:

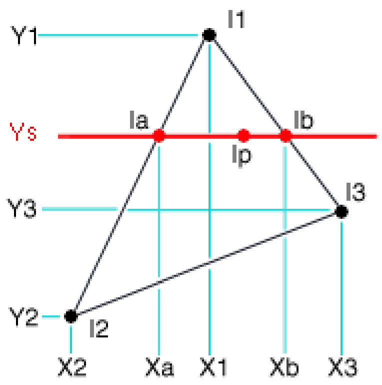


Fig.1 Smooth (interpolation) shading [1].

```
Ia = (Ys - Y2) / (Y1 - Y2) * I1 + (Y1 - Ys) / (Y1 - Y2) * I2
Ib = (Ys - Y3) / (Y1 - Y3) * I1 + (Y1 - Ys) / (Y1 - Y3) * I3
Ip = (Xb - Xp) / (Xb - Xa) * Ia + (Xp - Xa) / (Xb - Xa) * Ib
```

We assume that the location of the vertices of the triangle is fixed.

Input

- (R_1,G_1,B_1), (R_2,G_2,B_2), (R_3,G_3,B_3),– the values of red, green, blue components (0-255) of the color (input from keyboard) of three vertices

Output

- BMP file containing shaded triangle:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “shading.bmp”

References:

[1] „**Lighting and Shading**”, http://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/LightingAndShading.html
[2] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.3 Binary image

Convert rectangular part of the source BMP image (24 bit RGB) to binary image (black white) using thresholding. The color of output pixel is white when inequality (1) is satisfied. Otherwise the pixel is black. The image outside the rectangular region is unchanged.

$$\boldsymbol{thres \geq 0.21R + 0.72G + 0.07B} \tag{1}$$

where R,G,B are the components of the color of the pixel.

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”
- (x1,y1) – top left corner of rectangular region (input from keyboard)
- (x2, y2) – bottom right corner of rectangular region (input from keyboard)
- thres – threshold value (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “dest.bmp”

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.4 Puzzle

Divide the source image from a BMP file into n by m pieces and put them in random order.

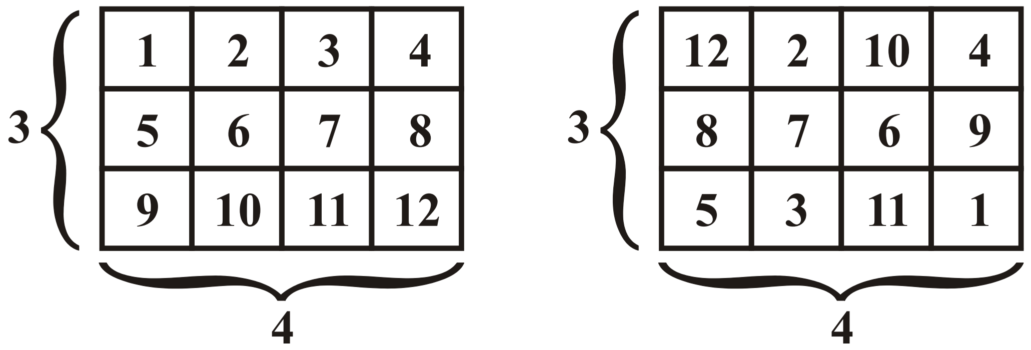


Fig.1 Source image (left) divided into 3x4 pieces. The tiles placed randomly on the destination image (right).

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: "source.bmp"
- n – number of vertical divisions (input from keyboard)
- m – number of horizontal divisions (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: "dest.bmp"

References:

[1] "**file-format-bmp**", https://www.daubnet.com/en/file-format-bmp

6.5 Puzzle 2

Divide the source image from a BMP file into 3 by 4 pieces and put them in new order. Location of each piece in source image is described by two digits representing row number and column number. The new order of pieces in the destination image is described row-wise (from left to right) by numbers representing original locations of the pieces (fig. 2).

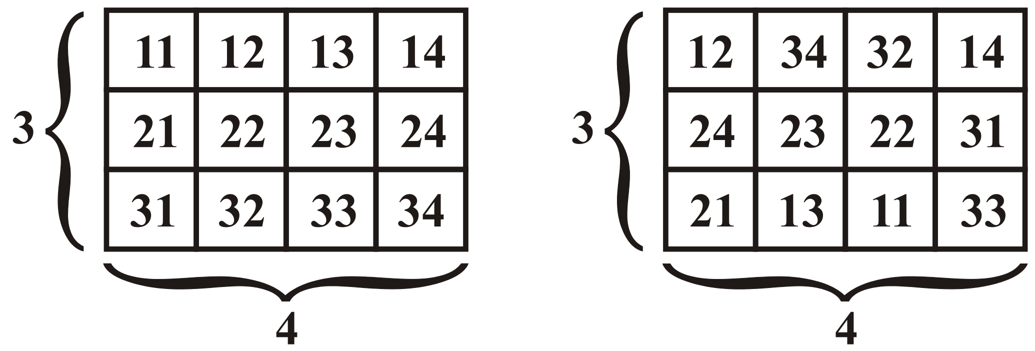


Fig.1 Source image (left) divided into 3x4 pieces. New placement of the tiles in the destination image (right).

```
12,34,32,14,24,23,22,31,21,13,11,33
```

Fig.2 Description of the placement of the tiles in the destination image (fig. 1)

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 318x240px,
  - file name: “source.bmp”
- s – description of the placement of the tiles in the destination image (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “dest.bmp”

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.6 Maximum intensity

Mark (with a red frame) the region of the source BMP image (24 bit RGB) where the average intensity is maximum. The intensity of the pixel should be calculated according to formula (1). The average intensity should be calculated for a window of size m by n. If there is more than one region of the maximum intensity then the first detected should be marked.

$$\boldsymbol{I = 0.21R + 0.72G + 0.07B} \tag{1}$$

Input

- BMP file containing the source image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: "source.bmp"
- m – horizontal size of the scanning window (input from keyboard)
- n – vertical size of the scanning window (input from keyboard)

Output

- BMP file containing modified image:
    - Sub format: 24 bit RGB – no compression,
    - 320x240px,
    - file name: "dest.bmp"

References:

[1] "**file-format-bmp**", https://www.daubnet.com/en/file-format-bmp

6.7 Adding text

Add a single line of text (containing numbers and dots) to BMP image. The characters should be defined by a matrix of size 8 x 8 pixels eg. number 1:

```
. . . * * . . .
. * * * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . . * * . . .
. . * * * * . .
```

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “source.bmp”
- the text string containing only numbers and dots (input from keyboard)
- (*x,y*) – the position of the text in the image (input from keyboard)

Output

- BMP file containing modified image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240px,
  - file name: “dest.bmp”

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.8 Shape detection

Write a function which recognizes selected shapes (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

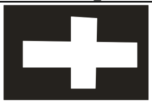
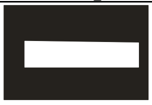
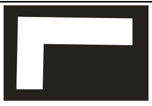
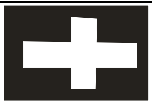
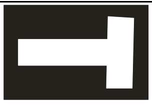
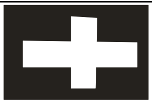
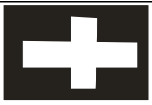
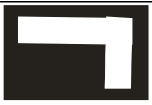
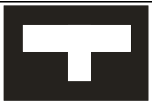
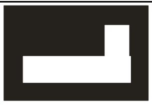
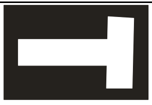
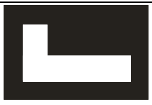
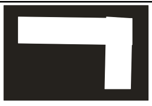
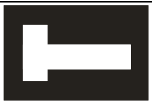
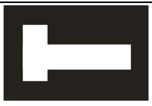
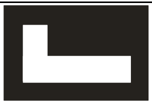
```
int shape(unsigned char *source_bitmap)
```

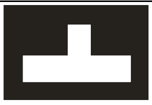
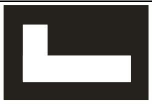
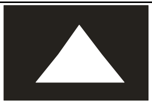
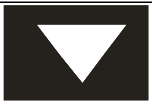
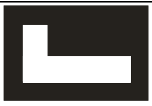
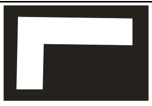
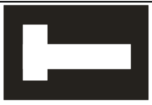
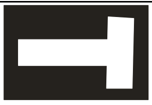
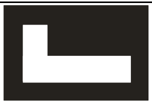
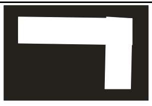
where:

- *source_bitmap* is a pointer to source image (containing header and pixel data)

The function should return:

- 1 or 2 – detected shape number,
- 0 – empty image
- negative number indicating an error (e.g. wrong image format).

Set no.	Shape 1	Shape 2
1		
2		
3		
4		
5		
6		
7		
8		

9		
10		
11		
12		
13		

Consider that shapes in the BMP file can be scaled.

Input

- BMP file containing the source image:
  - Sub format: 24 bit RGB – no compression,
  - 320x240 px,
  - file name: “source.bmp”

Output

- Console window – plain text

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp

6.9 Code 128 - barcode decoding

Write a function which decodes selected subset of Code 128 barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int decode128(unsigned char *source_bitmap,
            int scan_line_no,
            char *text)
```

where:

- *source_bitmap* is a pointer to source image (header and pixel data)
- *scan_line_no* – line number used for scanning,
- *text* – decoded text.

The function should return a 0 (OK) or a positive number indicating an error (e.g. wrong image size or format, wrong check symbol).

Input

- BMP file containing the barcode image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "source.bmp"

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be displayed on standard output.

Project versions:

1. Code Set A decoding
2. Code Set B decoding
3. Code Set C decoding
4. Decoding of any set (A, B or C).

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] "**file-format-bmp**", https://www.daubnet.com/en/file-format-bmp
[2] "**Code 128**", https://en.wikipedia.org/wiki/Code_128
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.10 Code 128 - barcode generation

Write a function which encodes a text using selected subset of Code 128 barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int encode128(unsigned char *dest_bitmap,
              int bar_width,
              char *text)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *bar_width* - the width in pixels of narrowest bar,
- *text* – text to be encoded.

The function should return a 0 (OK) or a positive number indicating an error (e.g. barcode does not fit in bitmap of given size, text contains characters which can not be encoded).

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
  - Colors: bars – black, background – white.
- file name: "output.bmp"

Project versions:

1. Code Set A encoding
2. Code Set B encoding
3. Code Set C encoding
4. Encoding of any set (A, B or C).

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] "**file-format-bmp**", https://www.daubnet.com/en/file-format-bmp
[2] "**Code 128**", https://en.wikipedia.org/wiki/Code_128
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.11 Code 39 - barcode decoding

Write a function which decodes Code 39 barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int decode39(unsigned char *source_bitmap,
             int scan_line_no,
             char *text)
```

where:

- *source_bitmap* is a pointer to source image (header and pixel data)
- *scan_line_no* – line number used for scanning,
- *text* – decoded text.

The function should return a 0 (OK) or a positive number indicating an error (e.g. wrong image size or format, wrong check symbol).

Input

- BMP file containing the barcode image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: “source.bmp”

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be displayed on standard output.

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp
[2] “**Code 39**”, https://en.wikipedia.org/wiki/Code_39
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.12 Code 39 - barcode generation

Write a function which encodes a text using Code 39 barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int encode39(unsigned char *dest_bitmap,
             int bar_width,
             char *text)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *bar_width* - the width in pixels of narrowest bar,
- *text* – text to be encoded.

The function should return a 0 (OK) or a positive number indicating an error (e.g. barcode does not fit in bitmap of given size, text contains characters which can not be encoded).

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
  - Colors: bars – black, background – white.
- file name: “output.bmp”

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp
[2] “**Code 39**”, https://en.wikipedia.org/wiki/Code_39
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.13 RM4SCC - barcode decoding

Write a function which decodes RM4SCC barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int decodeRM4SCC(unsigned char *source_bitmap,
            int scan_line_no,
            char *text)
```

where:

- *source_bitmap* is a pointer to source image (header and pixel data)
- *scan_line_no* – line number used for scanning,
- *text* – decoded text.

The function should return a 0 (OK) or a positive number indicating an error (e.g. wrong image size or format, wrong check symbol).

Input

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: “source.bmp”

The bars are black and are paralel to vertical edge of the image. The background is white. There are no distortions in the image.

Output

The decoded text should be displayed on standard output.

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp
[2] “**RM4SCC**”, https://en.wikipedia.org/wiki/RM4SCC
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.14 RM4SCC - barcode generation

Write a function which encodes a text using RM4SCC barcode (the same task as for MIPS project). The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

```
int encodeRM4SCC(unsigned char *dest_bitmap,
              int bar_width,
              char *text)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *bar_width* - the width in pixels of narrowest bar,
- *text* – text to be encoded.

The function should return a 0 (OK) or a positive number indicating an error (e.g. barcode does not fit in bitmap of given size, text contains characters which can not be encoded).

Input

- the width in pixels of narrowest bar,
- text to be encoded.

Output

- BMP file containing the barcode image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
    - Colors: bars – black, background – white.
- file name: “output.bmp”

Remarks:

1. Do not store bars and spaces patterns in coding table as character strings.
2. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] “**file-format-bmp**”, https://www.daubnet.com/en/file-format-bmp
[2] “**RM4SCC**”, https://en.wikipedia.org/wiki/RM4SCC
[3] Example images, http://galera.ii.pw.edu.pl/~zsz/ecoar/images/barcodes

6.15 Binary turtle graphics – version 1

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32 bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
    - Sub format: 24 bits RGB – no compression,
    - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.16 Binary turtle graphics – version 2

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32 bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```
where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.17 Binary turtle graphics – version 3

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.18 Binary turtle graphics – version 4

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32 bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.19 Binary turtle graphics – version 5

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32 bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.20 Binary turtle graphics – version 6

Your task is to write a program, which translates binary encoded turtle commands to a raster image in a BMP file (the same task as for MIPS project). The function should be written in Intel x86 32bits assembly language. The C/C++ declaration of the function should be

```
int turtle(unsigned char *dest_bitmap,
           unsigned char *commands,
           unsigned int commands_size)
```

where:

- *dest_bitmap* is a pointer to generated image (header and pixel data),
- *commands* – pointer to the binary encoded turtle commands,
- *commands_size* – size of the command block (in bytes).

The function should return a 0 (OK) or a positive number indicating an error.

Input

- binary file containing 16/32-bits turtle commands
- file name: "input.bin"

Output

- BMP file containing the generated image:
  - Sub format: 24 bits RGB – no compression,
  - Image size: 600x50 px,
- file name: "output.bmp"

Remarks:

1. File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] "**Turtle graphics**", https://en.wikipedia.org/wiki/Turtle_graphics

6.21 Food processor

The input BMP image [1] (24-bit RGB) contains one of three food types from the given set (Table 1). Your task is to recognize the food and print in the console window its three-character code (the same task as for MIPS project).

The function should be written in Intel x86 (32 and 64 bit) assembly language. The C/C++ declaration of the function should be

`int food(unsigned char *input_bitmap)`

where:

- *input_bitmap* is a pointer to the food image (header and pixel data),

The function should return a 1, 2, 3 (class numbers) or a negative number indicating an error.

Input

- BMP file containing the source image:
  - sub format: 24 bit RGB – no compression,
  - size: width 200px, height up to 200 px,
  - file name: "source.bmp"

Output

- Console window – plain text

Remarks:

1. Please pay attention to the efficiency of the program (getPixel function is not very efficient)
2. Check the input data and signal errors (e.g. wrong file format)

Remarks:

File I/O, memory allocation, displaying of messages should be done in C/C++ program.

References:

[1] BMP file format – see section 4.2
[2] RawFooT DB: Raw Food Texture Database,
http://www.ivl.disco.unimib.it/minisites/rawfoot/textures.php
[3] Mode (statistics), https://en.wikipedia.org/wiki/Mode_(statistics)
[4] Hexadecimal file editor, https://hexed.it/

6.22 Find image markers

The task is to write a program that detects one of the twelve types of markers in the image stored in the BMP format [1]. The shape of the markers and their characteristics are discussed in chapter 3.22.

The program should consist of a part written in C/C++, from which a function written in assembler of Intel x86 processor (32 and 64 bits) is called. When writing a program, you should implement an assembler function with the following declaration in the C language:

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

Description:
The function detects markers in the image according to the specification from task 3.22.

Return value:

- Value greater or equal 0 – number of detected markers,
- Negative numbers indicate an error.

Function parameters:

- `bitmap` – pointer to the buffer containing the image (including the BMP header),
- `x_pos` – an array containing the x coordinates of the detected markers,
- `y_pos` – an array containing the y coordinates of the detected markers.

Input

- BMP file containing the source image:
  - sub format: 24 bit RGB – no compression,
  - size: **any size**,
- File name provided as parameter of the program

Output

- Console window – plain text - subsequent lines contain the coordinates of the detected markers, eg. 10, 15. The point (0,0) is in the upper left corner of the image.

Remarks:

1. Check the input data and signal errors (e.g. wrong file format)
2. We assume that the drawing may contain 50 markers (at most) of a given type.
3. Reading data from a file is performed at the C language level.
4. Buffers memory allocation is performed at the C language level.
5. The entire interpretation of the BMP file content should be done in an assembler function.
6. Please note that padding bytes may occur if the number of bytes in the image line is not divisible by 4.
7. A makefile should be provided with the project, which will be used to automatically generate an executable file called **find_markers**.
8. The project should be delivered as one .zip or .tar.gz file

References:

[1] BMP file format – see section 4.2
[2] Hexadecimal file editor, https://hexed.it/