Institute of Computer Science
Warsaw University of Technology

Computer Architecture Lab Tutorial

Grzegorz Mazur
Zbigniew Szymański

Draft version 29.02.2024

February 2024

The document describes the software environment used during lab classes of Computer Architecture course.

For correct display or printing, the document requires Adobe Source Serif 4, Source Sans 3 and Source Code Pro fonts to be installed in user's computer operating system. The font packages may be downloaded from https://github.com/adobe-fonts.

Table of Contents

1. Lab environment..... .4
1.1. RARS RISC-V computer simulator..... .4
1.2. x86 development environment..... .5
2. Galera server Linux environment..... .5
2.1. Account and password management..... .5
2.2. Logout..... .5
3. Working with Unix/Linux system..... .6
3.1. Basic Unix/Linux concepts..... .6
3.2. Common Linux commands..... .7
3.2.1. passwd – change password..... .7
3.2.2. ls – list directory content..... .7
3.2.3. pwd – print working directory..... .8
3.2.4. cd – change directory..... .8
3.2.5. ps – process status..... .8
3.2.6. kill – terminate process..... .9
3.2.7. mkdir – create directory..... .9
3.2.8. rm – remove files..... .9
3.2.9. cp – copy files..... .10
3.2.10. mv – move or rename files..... .10
3.2.11. tar – manage archives..... .10
3.2.12. mc – interactive file manager..... .11
4. Remote program development on Galera server using computer running Windows..... .12
4.1. Putty..... .12
4.2. WinScp..... .13
4.3. Notepad++ and the NppFTP plugin..... .15
5. x86 Linux programming environment on a personal computer..... .16
5.1. WSL Ubuntu setup..... .16
5.2. Programming tools setup..... .17
5.3. Using Visual Studio Code..... .17
5.3.1. Running VScode..... .17
5.3.2. Getting the examples..... .17
5.3.3. Creating a new project for VScode..... .18
6. gcc..... .18
6.1. Using gcc..... .18
7. Assembly programming..... .19
7.1. General rules and common techniques..... .19

7.2. RISC-V RV32I assembly..... .20
7.3. x86 assembly techniques..... .20
7.4. x86 32-bit hybrid programming..... .21
7.5. x86-64 64-bit hybrid programming..... .21
8. Make utility..... .22
8.1. Using make..... .22
8.2. Makefile example..... .23
9. Working with .BMP image files..... .23
10. Command line processing in C programs..... .24
11. Lab exercises..... .25
11.1. Simple RISC-V assembly program – exercise 1..... .25
11.2. Simple x86 hybrid program – exercise 2..... .25
11.3. RISC-V assembly project – exercise 3..... .27
11.4. x86 hybrid project – exercise 4..... .27
11.5. x64 version – exercise 5..... .28

1. Lab environment

The lab projects are to be implemented on Galera server of Institute of Computer Science, running Linux. While the RISC-V assembly programming projects are operating-system independent and may be developed on any OS platform (Linux/Windows/macOS), the x86 projects require Linux operating system.

The students are encouraged to use Galera server on-site or remotely for project development done at home. It is also possible to develop the programs on a home computer running Linux natively or in Ubuntu Linux session under Windows, using Microsoft WSL (Windows Subsystem for Linux) which may be easily installed from Microsoft Store. The final program, however, should be capable of running on the laboratory computers.

The subsequent chapters of this document contain the information on lab environment and setup instructions for developing the programs in various scenarios.

1.1. RARS RISC-V computer simulator

RARS simulator is used for RISC-V assembly programming projects. *RARS* is written in Java, and distributed as .jar Java archive, which makes it platform-independent. It may be run under any OS, including Linux, Windows and macOS. *RARS* simulator may be downloaded as operating system-independent Java archive .JAR file from

https://github.com/TheThirdOne/rars

To get the ready-to run version of the program, scroll down to the link labeled “releases page”.

https://github.com/TheThirdOne/rars/releases

The program may be executed under any operating system supporting Java by clicking on the program file name or icon or by issuing the command from the command line. To run *RARS*, Java environment must be installed. If it is not already present under Windows, it may be downloaded from java.com.

For convenience, *RARS* should be invoked from the folder in which the project source files are stored, to avoid traversing the directory tree while saving the source files. When using *RARS* for a single project under Windows, put the *rars* program file in the working folder and invoke it by clicking on its icon in the Explorer. While working under Linux, open the terminal, **`cd`** to the project directory, then invoke *rars* (spelled in all-lowercase letters):

```
cd my-rars-project
rars
```

The program developed under RARS will run in the same way when transferred between different operating systems. The projects may be prepared and tested on a home computer running Linux, Windows or macOS and transferred to Galera server.

1.2. x86 development environment

The x86 development environment used for lab projects consists of GNU C compiler, NASM assembler and (optional but highly recommended) *make* utility. The same environment may be used on galera server (locally or remotely) or a student's PC running Linux as the primary OS or in WSL Ubuntu (Ubuntu Linux running under Windows) – see chapter 5 for instructions .

The essential components of programming environment are:

- *gcc* – the GNU Compiler Collection,
- *gcc-mulitib* – 32-bit x86 support package for gcc (by default only 64-bit libraries are installed with gcc),
- *NASM* – the modern x86 assembler,

In addition to these, the following packages may be useful:

- *make* – for automating program compilation and linking,
- *git* – for cloning example projects from github.com/gbm-ii repository
- *Visual Studio Code* – may be used as editor or as a complete programming environment.

All of the above components are installed on Galera server.

2. Galera server Linux environment

The laboratory computers are configured for multiple boot. To use ECOAR lab environment, standard Ubuntu Linux environment should be loaded. If Windows login screen is displayed, select Restart and, while in UEFI boot menu, choose Ubuntu Linux as the operating system to boot. To log onto Galera, you must have Galera server account name and password, issued by Institute of Computer Science laboratory administrators, which is not the same as your University or Faculty account.

All the user files and settings are stored on Galera server. The files may be accessed from any workstation in the lab rooms 10 and 139.

2.1. Account and password management

Since the authentication is managed globally by Galera server, to change the password you must open a remote terminal session on Galera server. It is not possible to change the password locally, in the lab desktop computer only. See chapter 3.2.1 for details.

2.2. Logout

To finish your session you should logout from the system. In GUI, right-click on the desktop, then select LogOut from the popup menu. Do NOT turn off/shutdown the laboratory computers when leaving the lab unless explicitly asked for it by the laboratory staff.

3. Working with Unix/Linux system

3.1. Basic Unix/Linux concepts

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

Unlike in Windows, Linux file names are case-sensitive. A common error of novice users is to create a file (e.g. *document.txt*) and to try to refer to this file using a different letter case (e.g. *Document.txt*).

The important directories in Unix file system are:

- Current working 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 used to locate files in a Linux file system. Each terminal session has its current directory. The current directory is referred to by a period character ".".
- Parent directory – it is a directory one level higher in the file system hierarchy relative to the given directory. The parent directory is referred to by two dots "..".
- Home directory – it is a directory designated by the system administrator for storing the user's files. The home directory is referred to by a tilde character "~".
- Root directory – it is the origin of the whole file system. It is referred to with a slash character "/" placed at the start of the path string.

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 / contains a *home* subdirectory, and the *users* subdirectory in it. *xgucio* subdirectory is located in *users* directory and contains the *prog.cpp* file.

Figure 3.1 shows the components of the command line prompt. Starting from 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.

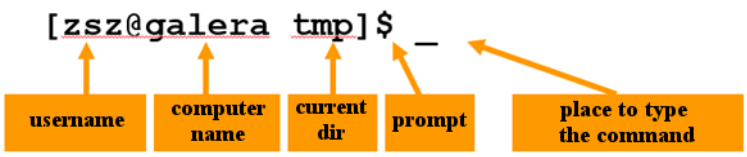

Figure 3.1: Components of Linux command line prompt.

3.2. Common Linux commands

This section describes some of the basic Linux commands. It should be noted that many of these commands have significantly more options than presented below. A full description of system commands is available in the system help which may be accessed using the command:

```
man command_name
```

3.2.1. passwd – change password

The **passwd** command allows the user to change the login password. The initial password is difficult to remember because it was machine generated – it may be convenient to change it. Please invent a new password – not shorter than 6 characters, containing at least one capital letter or digit 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 *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 chapter 4.1). After logging in, run the **passwd** command and enter the new password twice. In the terminal window, issue the commands shown below as bold text (NOTE: **jkowalsk** is an example login name, your login name will appear instead).

Terminal window:	Comment:
`[jkowalsk@galera ~]$ passwd`	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

The changed password will be automatically propagated from the Galera server to all Linux PCs in the laboratory (with maximum delay of 3 minutes).

3.2.2. ls – list directory content

The **ls** command lists the files and subdirectories of the current directory. The **ls -al** command (with the **-al** option; note that the options are separated from the command name by a space) displays lines similar to one shown below.

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

The first column containing **-rw-r--r** shows the access rights to the file which 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 is **May 22 2018**.

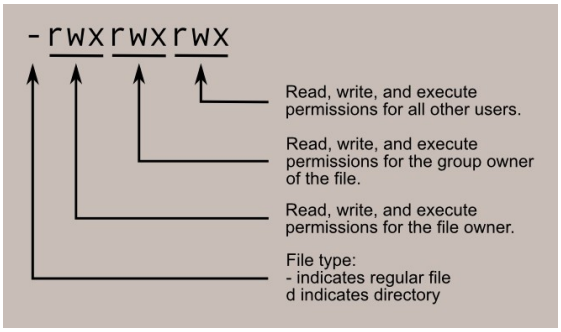

Figure 3.2: File access rights in Unix systems.

File permissions in Unix/Linux systems are described in Figure 3.2. There are three sets of permissions: for the file owner, for users belonging to the same group as the owner and for other users. A detailed description of the permissions can be found in [http://linuxcommand.org/lc3_lts0090.php].

3.2.3. pwd – print working directory

The **pwd** (print working directory) command displays the full absolute path of the current directory. Immediately after logging in, the output of **pwd** shows the home directory of the user, like **/home/users/gucio**

3.2.4. cd – change directory

The **cd** command changes the current working directory (folder). 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.

3.2.5. ps – process status

The term *process* refers to a running instance of a program. The process includes program instructions and data, 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), 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 or automated processes) are processes not connected to the terminal; they do not expect user interaction.

There are many Linux tools for 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 – the figure below 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 …
…
```

Figure 3.3. Output of ps 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.

3.2.6. kill – terminate process

`kill` is a Unix command which sends signals to processes running in the operating system. By default, **`kill`** sends a signal to the specified process instructing it to exit (terminate). 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 3.3 (the process is identified by the PID number, which may be discovered by running **`ps`** (in this example it 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`** option. 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
```

3.2.7. mkdir – create directory

The **`mkdir`** command is used to create a new directory (folder). 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
```

3.2.8. rm – remove files

The **`rm`** command is used to delete files and directories. When deleting, you will be asked to confirm the deletion. To avoid confirmation questions and warnings about nonexistent files, use the **`-f`** option. The **`-r`** option causes recursive operation – the **`rm -r`** will remove all the files and directories in the specified directory, so it should be used with caution.

To remove the *main.cpp* file from the current directory, use:

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

The use of **-f** above suppresses the warning message if the *main.cpp* file does not exist.

To remove the *project1* directory (including its content – files and subdirectories) in the current directory, use:

```
rm -rf project1
```

3.2.9. cp – copy files

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** option representing recursive copying. The following command copies the directory named project1 to the archive directory located in the parent directory:

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

3.2.10. mv – move or rename files

The **mv** command is used to move or rename files and directories. The general method of use is:

```
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.

3.2.11. tar – manage archives

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** command syntax for creating an archive is:

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

The **-z** option is responsible for compressing the archive with **gzip**. The **-c** option 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** command syntax for unpacking the archive is:

```
tar -zxf archive_file.tar.gz
```

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

3.2.12. mc – interactive file manager

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 3.4).

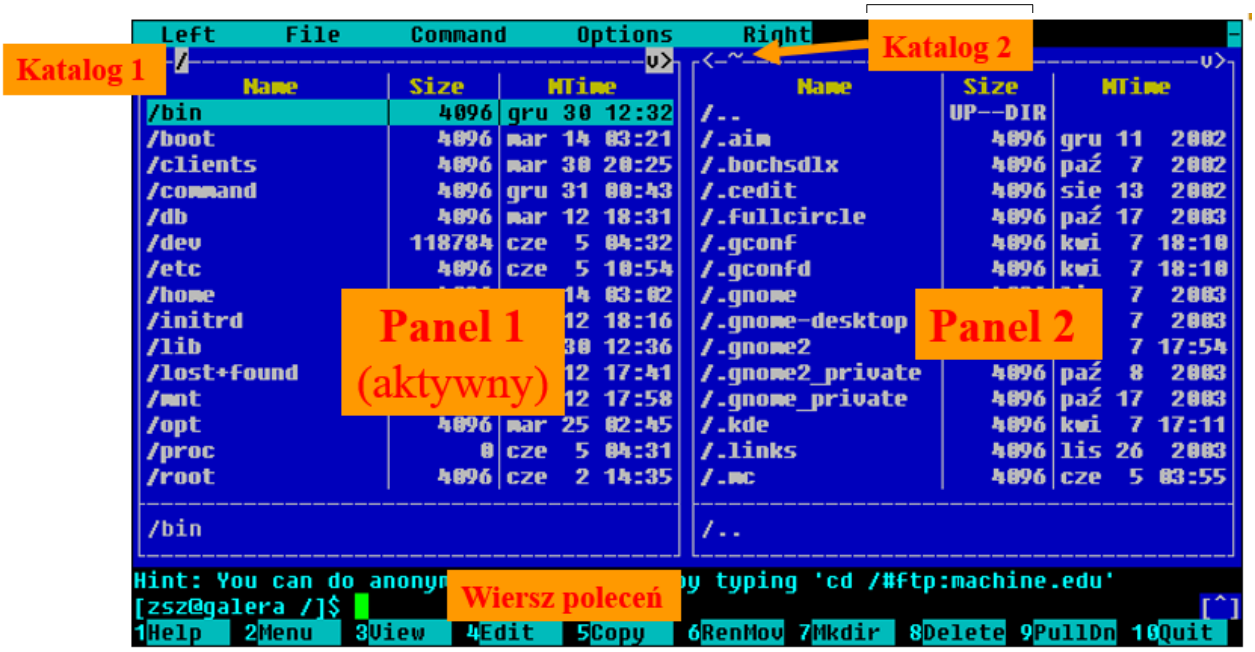

Figure 3.4: Midnight Commander user interface

The program displays the contents of two directories in two panels. Directory paths are displayed in the top edges of frames surrounding the panels. One of the panels is active – its file list may be scrolled by pressing the up and down arrows on the keyboard and the files and directories may be selected by pressing the Insert key (selected items are marked in 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 changes the active panel.
The highlighted files can be copied or moved from the directory displayed in the active panel to the directory which content is shown in the other panel by pressing F5 or F6 respectively. Pressing the F7 key creates a new directory. Deleting the highlighted element or group of elements marked in yellow is done by pressing the F8 key. The upper menu is activated by pressing F9 and pressing the F10 key exits the program.

Midnight Commander also provides a classic command line interface where the user can enter any system command. If the running command displays messages on the screen, they will be obscured by mc panels after the operation is completed. To show them, 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.

The program also contains a text file viewer and an editor which may be activated respectively by pressing the F3 and F4 keys (after placing the highlight on the file name).

Keyboard shortcuts of *Midnight Commander* are shown in the table below.

Key	Function
F3	Text file viewer
F4	Text file editor
F5	Copy file
F6	Move file

Key	Function
F7	Create directory
F8	Delete file
F9	Top menu
F10	Exit program

Key	Function
Tab	Switch panel
Insert	Select a file
Ctrl+O	Hide/Show panels

4. Remote program development on Galera server using computer running Windows

Galera server may be accessed remotely via SSH terminal session, allowing the Linux commands to be run remotely from a terminal program running on a home computer. It is possible to transfer the files between a home computer and Galera server and to edit the program source files and other text files stored on the server using a text editor running under Windows.

To develop programs under this scenario, it is recommended to use three utility programs:

- *PuTTY* terminal emulator,
- *WinSCP* secure file transfer client,
- *Notepad++* source code editor.

4.1. 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 may 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 (Figure 4.1). *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 (Figure 4.2). 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 is selected in the Connection type group. 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 by clicking the Open button.

The first time you connect to a computer, the *PuTTY* security alert window appears (Figure 4.3). You should click 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 4.4.

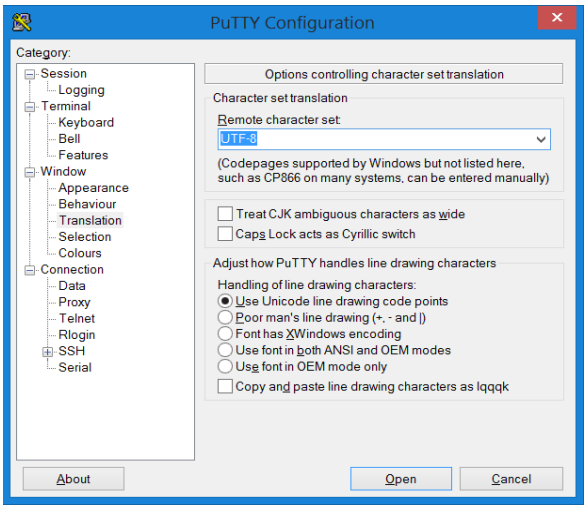

Figure 4.1: PuTTY character encoding selection.

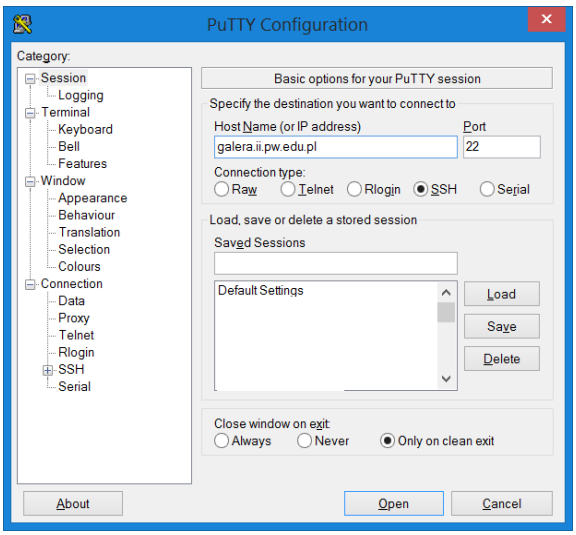

Figure 4.2: PuTTY session parameters.

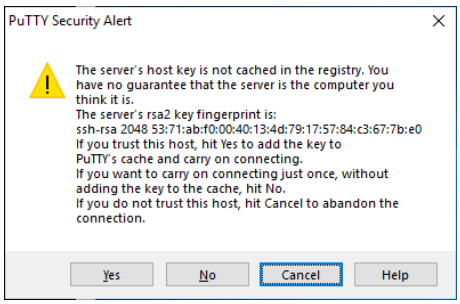

Figure 4.3: PuTTY Security Alert window

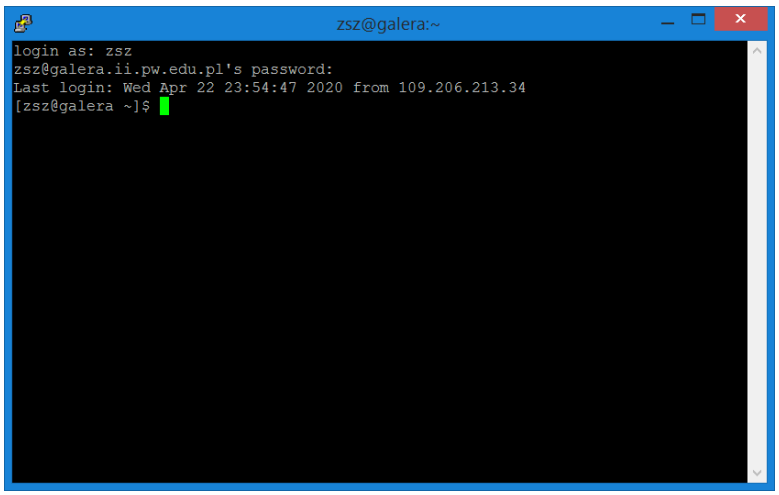

Figure 4.4: Terminal window after successful connection.

4.2. 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.

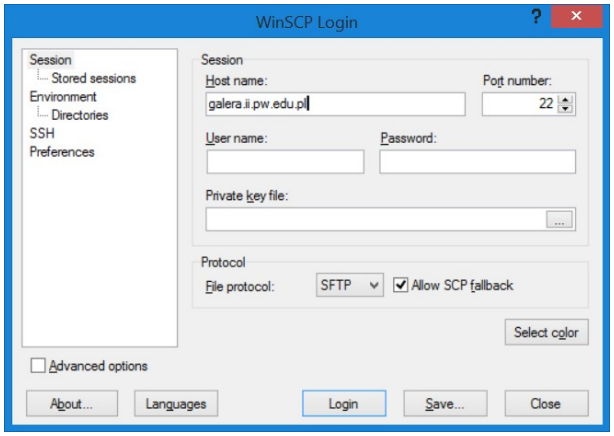

Figure 4.5: WinSCP session settings.

After establishing the connection, the *WinSCP* window should look similar to the one in Figure 4.6. 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 files between *WinSCP* panels or by dragging between the windows of the system File Explorer and the remote computer panel.

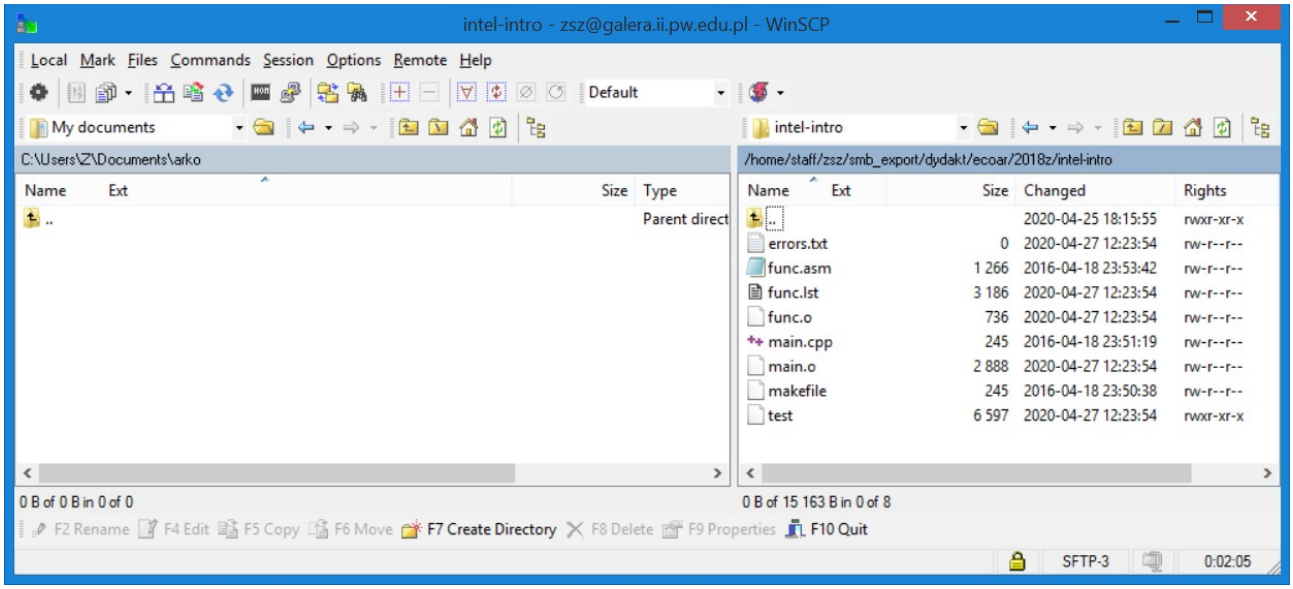

Figure 4.6: WinSCP main window.

4.3. Notepad++ and the NppFTP plugin

Notepad++ is a popular source code editor running under Windows, supporting syntax highlighting in many programming languages. The *NppFTP* plugin for *Notepad++* allows for editing files located on remote machines running Unix / Linux systems without having to download them first e.g. using the *WinSCP*. The Notepad++ program can be downloaded from the link:

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 select Show NppFTP Window (Figure 4.7). The Profile settings command (Figure 4.8) opens the dialog 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 (Figure 4.9), in the Hostname field enter 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.

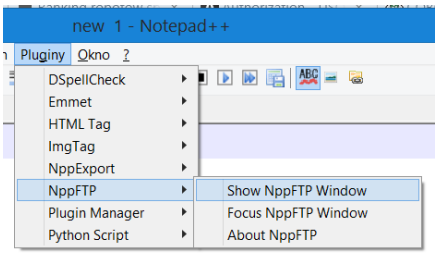

Figure 4.7: Enabling the NppFTP plugin window.

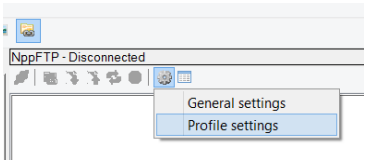

Figure 4.8: NppFTP plugin settings.

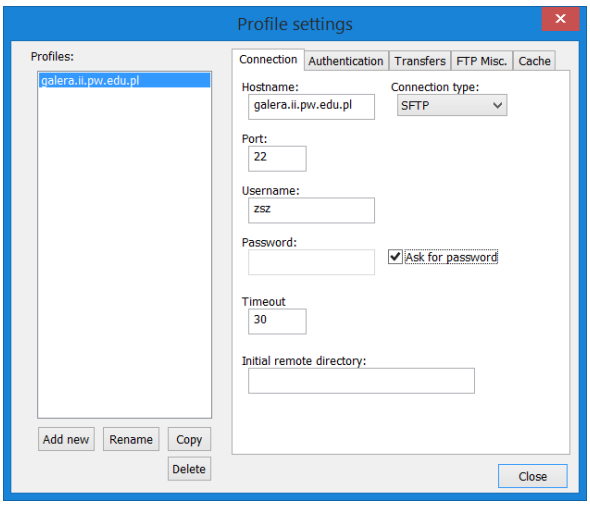

Figure 4.9: Connection profile configuration.

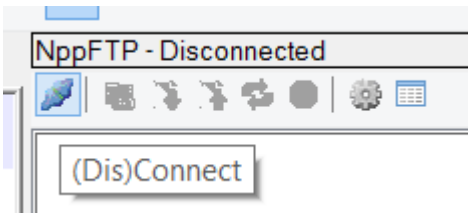

Figure 4.10: (Dis)Connect button.

The connection is established by clicking on the (Dis)Connect button (Figure 4.10) 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 (Figure 4.11). 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.

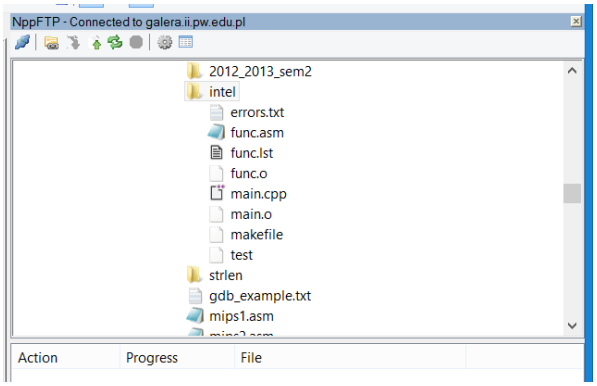

Figure 4.11: File list of the remote computer.

5. x86 Linux programming environment on a personal computer

A standard PC with x86-architecture processor may be used for x86 hybrid programming exercises. This way the whole development process may be carried out locally on PC running Linux or Windows, without the need for permanent connection to *galera* server.

The lab projects should be developed under Linux. If your personal computer is running Linux, skip to chapter 5.2. If your computer is running Windows operating system, it is possible to install the Microsoft-supplied lightweight Ubuntu Linux environment under Windows.

5.1. WSL Ubuntu setup

The WSL2 environment needed for 32- and 64-bit x86 program development requires 64-bit Windows 11 or Windows 10, version 2004 or newer. If your computer runs Windows 10, please check the version in System information and update if necessary.

As a first step, install *Windows Terminal* from Microsoft store. It will be useful as Ubuntu Linux terminal.

To install Ubuntu Linux on Windows 11, open the *Windows Terminal* with administrator rights, then type:

```
wsl --install
```

The installation process is automated; it will require system reboot at the end. At the last stage of setup you will be prompted to create a Linux administrator account protected by password. The password will later be needed for software installation and updates.

To use Linux commands after WSL is installed, open the Ubuntu session under Windows Terminal.

5.2. Programming tools setup

To install the required programming tools, open Linux terminal or Ubuntu session in Windows terminal, then enter the following commands:

```
sudo apt update
sudo apt upgrade
sudo apt install gcc gcc-multilib nasm make git
```

5.3. Using Visual Studio Code

Visual Studio code is a popular IDE. It may be used for the development of hybrid x86 programs.

Visual Studio Code may be installed under Linux on a PC running Linux system as native OS. The installation routine depends on the system version – check the current information online.

To use *Visual Studio Code* with WSL Ubuntu, install it under Windows (not in Linux) from Microsoft Store. When run from Windows for the first time, *VScode* should detect the presence of WSL and suggest the installation of a package enabling WSL software development with *VScode* running in Windows.

5.3.1. Running VScode

After the *VSCode* package is installed, open Ubuntu session in *Windows Terminal*, create a working folder for your project, `cd` to it, then enter the command:

```
code .
```

(code<space><dot>), to tell the *VScode* to use the current directory. Watch the messages displayed and allow for installing the missing and suggested *VScode* components if required.

5.3.2. Getting the examples

The examples include the makefiles and *VScode* configuration files. To get the examples, in Ubuntu terminal type:

```
git clone https://github.com/gbm-ii/x86-cc-nasm
```

Alternatively, you may clone the repository from within the *VScode*.

To work with examples, change directory to a selected example version, e.g.:

```
cd x86-cc-nasm/x86
```

Then start *VScode* by typing:

```
code .
```

In *VScode*, install the support extensions for C/C++ and NASM. The recommended packages are:

- C/C++ extension from Microsoft

- NASM Language Support extension by doinkythederp

While in *VScode*, you may compile the example projects by pressing Ctrl+Shift+B; the *tasks.json* files providing the appropriate build configuration and error message parsing are supplied together with the example projects.

5.3.3. Creating a new project for VScode

To create a new project:

- make a folder and **cd** to it

```
mkdir mynewproject
cd mynewproject
```

- copy the example *Makefile* to the new project folder

```
cp -r ../<old project path>/Makefile .
```

- copy the *.vscode* folder with *tasks.json* file from the example folder to the new project folder

```
cp -r ../<old project path>/.vscode .
```

- open *VScode*

```
code .
```

- in *VScode* create the new .c and .s files
- edit the *Makefile* using *VScode* – change the executable and object file names to match the newly created files' names.

6. gcc

The gcc package is a set of compilers and accompanying tools for creating programs, originally written for Unix/Linux environment but also ported to other operating systems. The toolset consists of *gcc* C/C++ compiler, *as* assembler, *ld* linker and few others. Since the *as* assembler for x86 uses an alternate (AT&T) assembly language syntax, incompatible with the one used in Intel/AMD documents, in the lab environment NASM assembler is used instead.

6.1. Using gcc

Gcc is a compiler driver which may invoke all the programs needed to compile a C or C++ program and link the modules to create the final executable file.

Full documentation of gcc may be found at:

https://gcc.gnu.org/onlinedocs/

To compile and link a C program contained in a single source file with default compiler settings, use:

```
cc -o myprog myprog.c
```

The **-o** option is needed to set the name of an executable file. If the option is skipped, the executable program file name is *a.out* (regardless of the source file name).

The program created using the command above may be run by typing:

```
./myprog
```

By default, the Linux command search path does not include the current directory, thus it it necessary to explicitly specify the path (`./`) before the program file name.

In a 64-bit x86-64 environment, *gcc* by default produces 64-bit code. The **`-m32`** option causes the compiler to produce 32-bit x86 program instead.

Gcc may compile the source file to intermediate (object) form, skipping the linking step. The command:

```
cc -m32 -c myprog.c
```

causes the compiler to emit *myprog.o* object file containing the 32-bit version of compiled code. To create an executable program from a compiled 32-bit object module, use:

```
cc -m32 -o myprog myprog.o
```

Multiple source or object file names may be specified in the command line to compile and/or link the program containing multiple modules.

Gcc may also produce the assembly source representation of compiled program. To see the assembly code created by the compiler, use **`-S`** (uppercase 'S') option:

```
cc -m32 -fno-pie -fcf-protection=none -fno-asynchronous-unwind-tables
-masm=intel -S myprog.c
```

The above command produces the file *myprog.s* which may be viewed using a text editor. The set of **`-fxxx`** options causes the compiler to produce a simplified version of code, which is easier to read and analyze then the default one, with advanced features. The **`-masm=intel`** option sets the assembly output syntax to Intel-style (the default syntax is AT&T). **`-Ox`** (uppercase letter 'O') options may be used to set the optimization level (**`-O0`**, **`-O1`**, **`-O2`**, **`-O3`**, **`-Os`**, etc.).

7. Assembly programming

7.1. General rules and common techniques

The assembly language program should not contain any obvious inefficiencies. In particular:

- Multiplication of integers by powers of 2 should be implemented via left-shifting.
- Multiplication by 2^n+1 or 2^n+2^m should be implemented using left-shift with addition (on x86 platform LEA instruction may be useful).
- Division of integers by powers of 2 should be implemented by right-shifting.
- Modulo by powers of 2 should be implemented by bitwise AND operation.
- Conditional branches followed by unconditional ones should be avoided whenever possible (usually by negating the branch condition).
- Temporary variables should be placed in registers whenever possible, especially in inner loops, to avoid slow memory references.

- When possible, loops which are iterated at least once should be written with the condition check at the end of the loop so that a backward conditional branch is used for closing the loop.
- If the number of iterations of a loop is known before the loop start it is usually easier to write the loop in such a way that it ends with decrementing the loop counter and checking if it is zero.
- Avoid the sequences of consecutive branches, esp. conditional branches followed by unconditional ones (unless really necessary – a rare case).

In case of nested loops, it is particularly important to optimize the innermost loop to avoid unnecessary memory references.

If a task involves scanning (converting) a number from text representation to binary, the basic algorithm is to process the characters from the first (most significant) to the last (least significant) digit, multiplying a number scanned so far by system base and adding a value of the current digit. If the system base is fixed and it is a power of 2, the multiplication should be done by left-shifting. The value of a decimal digit represented by character x may be obtained by subtracting the character '0' (code of digit zero).

7.2. RISC-V RV32I assembly

The full list of RISC-V instructions, pseudoinstructions and assembler directives may be found in RARS help system.

Some important remarks related to RISC-V assembly programs in RARS environment:

- Make sure to terminate the execution of a program by calling exit service.
- Use the alternate register names as defined by RISC-V ABI rather than generic names x0..31.
- In RISC-V assembly language, comments start with # character.
- Use `li` pseudoinstruction for loading numeric constants; use `la` for loading addresses.
- Use `mv` pseudoinstruction to copy register content.
- Treat text characters as unsigned integers – use `lbu` instruction for loading them, use conditional branch instructions with `u` (unsigned) suffix while checking or comparing their values.
- To compare register content against a constant, the constant value must be loaded to another register. If the compare operation occurs in a loop, load the constant before the loop.
- A non-leaf procedure should store the return address on the stack in the prologue and retrieve it from the stack in the epilogue, right before returning to the caller; see the RISC-V presentation for the implementation of stack operations.

7.3. x86 assembly techniques

Some of x86-specific good programming practices:

- avoid `cmp <reg>, 0` – use `test <reg>, <reg>` instead;

- avoid **cmp/test** after an arithmetic operation – the flags are set by arithmetic and logic instructions, so there is no need to set them again;
- use **xor <reg>, <reg>** to set the register to 0, it is usually more compact and faster than **mov <reg>, 0**;
- use **inc/dec** instead of **add <rm>, 1** and **sub <rm>, 1**;
- in 32-bit programs, use both addressable bytes of a register to store two byte-size variables in a single register (**ah** and **al**, **ch** and **cl**, etc.);
- in 64-bit mode, avoid using the upper byte registers (**ah**, **ch**, **dh**, **bh**) – they cannot be used in instructions using **sil**, **dil** or any of **r8..r15** registers.

7.4. x86 32-bit hybrid programming

The project should consist of two source files – the main program file written in C and a procedure/function written in x86 32-bit assembly. The C program should provide all necessary input and output operations; no system functions should be called from assembly code. Use NASM assembler (nasm.us) to assemble the assembly language module. Use C compiler driver to compile C module and link it with the output of assembler.

The two files must have different names (not only extensions). The assembly file should have an extension .s. Assuming that the source files are named *program.c* and *routine.s*, the commands used to produce the executable program are:

```
cc -m32 -c program.c
nasm -f elf32 routine.s
cc -m32 -o program program.o routine.o
```

The assembly module must conform to C calling convention as described in System V Application Binary Interface, Intel386 Architecture Processor Supplement, Version 1.1 (2015, file intel386-psABI.pdf). The last known good link to this document is:

https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/intel386-psABI-1.1.pdf

The basic information on x86 calling convention may be found in lab presentations, module *llp4*.

The assembly routines called by C code should use frame pointer-relative addressing for accessing arguments and local variables. The prologue and epilogue must follow the guidelines of a calling convention. With the prologue conforming to the calling convention, the first argument is at **[ebp + 8]**. The procedure must preserve the values of registers belonging to the saved register group. If the assembly routines called by C code call other assembly routines not directly called by C code, the latter may use any argument or value passing mechanism – they don't need to conform to the C calling convention; the only requirement is that the whole procedure tree, entered by calling the top-level procedure, does not violate the rules.

7.5. x86-64 64-bit hybrid programming

The program, composed similarly to the previous one but with assembly module written for 64-bit x64 (x86-64) architecture, may be obtained by executing the following sequence of commands:

```
cc -c program.c
nasm -f elf64 routine.s
cc -o program program.o routine.o
```

The x86-64 calling convention document is at:

https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

Note that the register grouping in x64 calling convention is different from x86 convention, for example the RDI and RSI registers are argument registers and, as such, belong to the caller-saved (temporary) group.

In 64-bit mode, instructions with register destination in which arguments are unsigned numbers with no more than 32 significant bits should be written as 32-bit ones – the 32-bit instructions clear the upper half of a destination register, effectively yielding 64-bit unsigned results, while having shorter binary encodings than 64-bit instructions.

In particular:

- `xor edx, edx` sets 64-bit register `rdx` to 0
- `mov eax, 12345` stores 64-bit value 12345 in `rax`

8. Make utility

Make utility simplifies the development of programs composed of multiple source files by automatically invoking the commands needed for the processing of source files.

The purpose of *make* is to perform only the necessary work, avoiding the unnecessary one. In case of a project with multiple source files, invoking *make* causes only the recently modified source files to be compiled into intermediate (object) files, then all the object files are linked into the final executable file.

8.1. Using make

The operation of *make* is controlled by the content of a text file carrying the default name *Makefile*, present in the folder from which make command is invoked. Te appropriate makefiles are included with the example projects.

The commands to be executed and relations between project files are described in the *Makefile*. The make compares the modification dates/times of source files, called prerequisites, and target files and processes the source file only if the target file does not exist or if it is is older than source.

A full make documentation may be found at:

https://www.gnu.org/software/make/manual/html_node/index.html

In the case of a hybrid project created from C and assembly source files, if only one of the source files is modified, *make* will compile only the modified source file into object, then link both object files into the executable program.

A makefile defines a set of rules for creating targets. A single *Makefile* may specify multiple actions by defining multiple targets. To process a target, make must access its prerequisites. If a

prerequisite is a target of another rule, it must be processed before it is used. If *make* is invoked without arguments, the first target defined in *Makefile* is processed, which may result in processing some other targets. The first, default target is usually used for normal target creation. Frequently, a target named **`clean`** is defined to remove all the derived project files (object, executable, listings), leaving only the original source files for archiving.

8.2. Makefile example

The example of a simple makefile producing executable file *rev* from C source file *rev.c* and assembly source file *mystrrev.s* is shown below. The text following # character in every line is treated by *make* as a comment. This makefile uses only basic *make* feature – explicit rules – to describe the build process.

```
# the first, default target is executable program rev
rev: rev.o mystrrev.o   # executable rev is made from rev.o and mystrrev.o
 cc -m32 -o rev rev.o mystrrev.o # command to make rev from rev.o & mystrrev.o

rev.o: rev.c      # rev.o is made from rev.c
 cc -m32 -c rev.c # command to make rev.o from rev.c

mystrrev.o: mystrrev.s  # mystrrev.o is made from mystrrev.s
 nasm -f elf32 mystrrev.s     # command to make mystrrev.o from mystrrev.s

# "target" to remove files - use "make clean" to do it
clean:
 rm -f *.o *.lst rev
```

A more advanced version of makefile, using variable definitions and implicit rules, is included with the example project available at github. The use of variables and implicit rules increases flexibility and portability of makefiles and enables easy modification and adaptation of makefiles for new projects.

9. Working with .BMP image files

Many of the lab projects are dedicated to processing of images stored in files. The Windows .BMP format, used to store images in uncompressed form, is one of the simplest image file formats. The description of .BMP file structure may be found at:

https://en.wikipedia.org/wiki/BMP_file_format

It should be noted that, regardless of pixel data format, a single horizontal line of image pixels in .BMP file always occupies a multiple of 4 bytes, with possibly some bits or bytes left unused at the end of line. For processing of the .BMP images, it is necessary to know the three key image parameters:

- image height in pixel lines,
- image width in pixels,
- size of a single image line in bytes, obtained by rounding the used size of image line in bytes up to the closest boundary of 4.

The height and width are stored in DIB header structure, being a continuation of .BMP file header. Since the members of BMP header structure are not size-aligned, special care must be taken to properly read/interpret the BMP header data fields – the header cannot be simply read as a whole into a C language representation of its structure. To obtain image line size (frequently referred to as **stride**), use the expression like:

```
stride = (image_width_in_pixels * bytes_per_pixel + 3) & ~3
```

or

```
stride = (image_width_in_pixels * bits_per_pixel + 31) / 32 * 4
```

(Note that while implementing these in assembly, neither multiply nor divide instructions should be used; the above / and * operations require only bit shifts and bitwise AND.)

10. Command line processing in C programs

The C program may access the command line arguments passed to it. The **main()** function in a hosted environment is declared as **int main(int argc, char *argv[])**, where **argc** is the number of arguments in the command line and **argv** is an array of pointers to strings, with each string representing one argument. The first element of this array, **argv[0]**, is a pointer to the program name. If the program is invoked without arguments, **argc** has a value of 1. The first argument of a program, if exists, may be accessed as **argv[1]**.

An example of a program printing its arguments:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("program name: %s\n", argv[0]);
    for (int i = 1; i < argc; i++)
        printf("arg %d: %s\n", i, argv[i]);
}
```

The operating system shell is responsible for parsing the command line; the arguments are separated with spaces. If an argument string should contain a space or any of the special characters interpreted by shell ('<', '>', '?', '*' and few others), in the command invoking the program, the string must be included in quotation marks, for example:

```
./myprog 1stArg "The Second Argument" "Arg#3*with'special'characters"
```

While **argv[i]** represents the argument string, a single character argument may be referred to as **argv[i][0]**. If a number is passed as an argument, it must be scanned from a string using **sscanf()** or a similar function from C standard library. **sscanf()** function returns the number of successfully scanned objects; if a number cannot be scanned, it returns 0.

```
int x;
if (sscanf(argv[i], "%d", &x))
    printf("arg %d is number %d\n", i, x);
else
    printf("wrong argument %d\n", i);
```

11. Lab exercises

The course includes 5 laboratory project exercises. The first two exercises should be completed during the lab sessions dedicated to them with a time limit of two lesson hours. The other, "big" projects should be presented during the respective lab sessions.

11.1. Simple RISC-V assembly program – exercise 1

The task is to write a simple RISC-V (RV32I) assembly program running in RARS simulator environment, processing a string read from console and producing the desired console output. The project must be completed in 2 hours during a single lab session. The task will be similar to the ones listed below. Input string should be read from the console using read_string function. Most of the problems may be solved using a single text buffer, without a need to copy the characters from one buffer to another. Typically a task which changes the length of a text string will involve copying the characters within a single buffer using two, or maybe three pointers.

Example tasks:

- Write a program replacing all digits in a string with their complement to 9 (0→9, 1→8, 2→7 etc.)
- Write a program displaying the longest sequence of digits found in a string.
- Reverse the order of digits in a string.
- Scan the biggest unsigned decimal number found in a string and display its value using print_int function.
- Display the number of decimal numbers (sequences of decimal digits) found in a string.
- Remove all digits from a string, display the resulting string.
- Remove from a string the sequence of characters specified by positions entered as integers; handle correctly out-of-range cases and both possible orders of positions:
  "abcdefgh" 2 4 or "abcdefgh" 4 2 should produce "abfgh"
  "abcdefgh" 90 100 should produce "abcdefgh"
  "abcdefgh" 10 5 should produce "abcde"
- Remove characters preceded by digits: "abc5f67gh" → "abc56h"
- Change every 3rd lowercase letter to uppercase: "ab1cde2f3gh4ij" → "ab1Cde2Fgh4Ij"
- Remove all letters but the first from each sequence of uppercase letters: "ABCdefGHi" → AdefGi"

11.2. Simple x86 hybrid program – exercise 2

The task is to write a simple x86 32-bit hybrid program with main module written in C and a function written in x86 assembly language (NASM assembler). The assembly module must conform to C calling convention as described in System V Application Binary Interface, Intel386 Architecture Processor Supplement (see chapter 7.4).

The assembly routine should process the text string passed to it. The C program should get the values of arguments passed to the function from the command line. Every command line

argument is a character string; if an argument to a function is a number, `sscanf()` function should be called to get the number's value into a variable (see chapter 10).

The project must be completed in 2 hours during the lab session. The task may be similar to the ones listed below.

```
char *removerng(char *s, char a, char b);
```

Remove characters with codes from a to b, a < b.

```
char *remnth(char *s, int n);
```

Remove every n-th character.

```
char *leavelastndig(char *s, int n);
```

Leave last n digits, removing all other characters.

```
char *remrep(char *s);
```

Remove repetitions of characters.

```
char *leavelongestnum(char *s, int n);
```

Leave only the longest sequence of decimal digits.

```
char *leaverng(char *s, char a, char b);
```

Leave characters with codes from a to b, a < b.

```
char *remlastnum(char *s);
```

Remove the last sequence of decimal digits.

```
unsigned int getdec(char *s);
```

Scan the first unsigned decimal number.

```
unsigned int gethex(char *s);
```

Scan the first hexadecimal number found in a string.

```
char *reversedig(char *s);
```

Reverse the order of digits, leaving the other characters in their original places.

```
char *reverselet(char *s);
```

Reverse the order of letters, leaving the other characters in their places.

```
char *reversepairs(char *s);
```

Swap characters in pairs.

```
char *replnum(char *s, char a);
```

Replace each sequence of digits with a specified single character.

```
char *capwords(char *s);
```

Capitalize the first character of each word in a string.

11.3. RISC-V assembly project – exercise 3

Write a program in RISC-V (RV32I) assembly using RARS simulator. The program should not rely on uninitialized register values.

Avoid the sequences of consecutive branches, esp. conditional branches followed by unconditional ones (unless necessary). The procedures written in assembly usually do not require a full prologue/epilogue sequence saving and restoring registers like in a code generated by a HLL compiler.

All the text processing programs with file i/o should define getc- and putc-like functions providing proper buffering of input and input operations with at least 512-byte buffers.

Graphic programs should display images using RARS graphic display mapped to heap address range.

Maximum score is 6 points. The project may be shown during two lab sessions. Each week of delay started will introduce a penalty of one point.

Project tasks are published in a separate file, specific to a lab group.

11.4. x86 hybrid project – exercise 4

Write a program containing two source files: main program written in C and assembly module callable from C. The C declaration of an assembly routine is given for each project task. The C program should use command line arguments to supply the parameters to an assembly routine. It should also perform all I/O operations. Arguments for bit manipulation routines should be entered in hexadecimal. No system functions nor C library functions should be called from assembly code.

Routines processing .BMP files may receive as arguments either the pointer to the whole .BMP file image in memory or the pointer to bitmap and its sizes read by main program (the declaration of an image processing function should include the appropriate argument list. The routines should correctly process images of any sizes unless stated otherwise.

The same program should be implemented in two versions:

- exercise 4 – 32-bit (x86), following the calling convention described in System V Application Binary Interface, Intel386 Architecture Processor Supplement,
- exercise 5 – 64-bit, (x86-64/x64) conforming to 64-bit Unix calling convention described in System V Application Binary Interface AMD64 Architecture Processor Supplement.

Both conventions are also described in a document available from www.agner.org. Maximum score for (any) single version is 6 points. The second version is worth 2 points.

Projects violating the calling convention will be rejected (not checked). An attempt to submit the project explicitly violating the calling convention may be punished by subtracting one point from the final score when the project is accepted.

11.5. x64 version – exercise 5

While converting the assembly program from x86 to x86-64, try to remove memory variables (if they were used in x86 version) and place the variables in extra processor's registers available in 64-bit mode. Note that usually only the addresses will be 64-bit. All the other data should be of the same size as in x86 version. Also note that 32-bit operations with register destination in 64-bit mode produce full 64-bit results, zero-extended from 32 bits, thus most of operations on numbers and memory displacements may be calculated with 32-bit operations. 64-bit operations are required for computing the addresses (adding 64-bit base address and displacement). The secondary components of addresses (indexes, variable displacements) may be computed as 32-bit.

An attempt to submit the project explicitly violating the calling convention may be punished by subtracting one point from the final score.