Task 6E

MICROELECTRONIC SYSTEMS DESIGN RESEARCH GROUP

Embedded Software development with Virtual Platforms
Dr. Matthias Jung, Éder F. Zulian (2017)

Organisational Matters

- Before you start with the task please read *Introduction to Virtual Platforms* carefully
- Use the server koa for this tasks
- After finishing your work, please end all simulations and logout properly
- If you have questions or problems, please contact:

Eng. Éder F. Zulian

Room: 12/251
Mail: zulian@eit.uni-kl.de

Introduction

In this task you have to develop a piece of embedded software on a virtual ARM Platform. The platform, as shown in Figure 1, consists of a clock generator, a reset generator, the ARM926 core, a ROM, a RAM and an AMBA TLM bus. Furthermore, we have a system tick counter, a terminal for text output and a special device for displaying pictures.

Your task is to implement a median filter for image processing as embedded software on this virtual platform. In the next paragraphs you will be introduced to image processing, especially to the median filter.

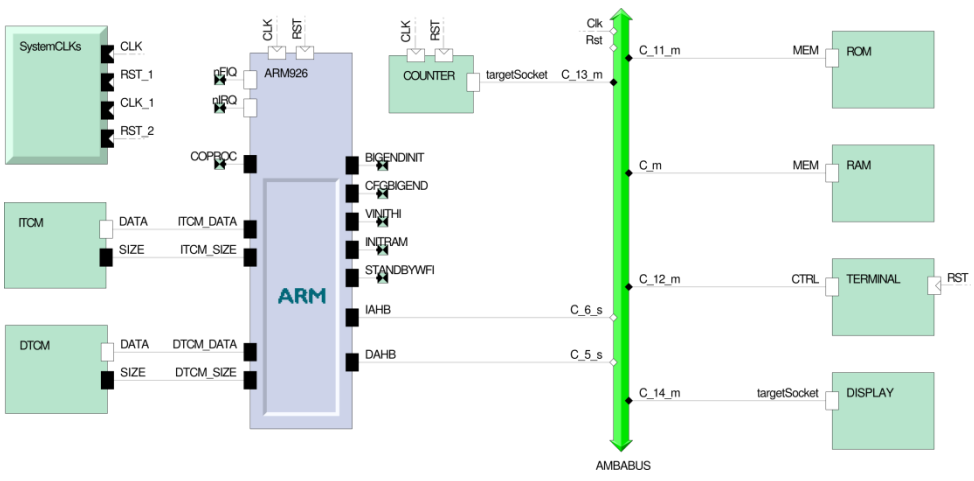


Figure 1: Simple Virtual ARM Platform

Image Processing [1] [2] [3]

Today's digital technology such as multimedia, communication, computers, robots and medical technology assists us in our daily life. For instance, industrial robots on an assembly line are able to recognize simple objects like screws and sort these automatically according to their size or detect and dump faulty work pieces. Computers are able to digitalize a text on a paper with OCR (Optical Character Recognition). Another example is a magnetic resonance scanner which is used to find and qualify tumours in a human body. All these applications use digital image processing algorithms.

Neighbourhood Operators

To extract attributes of an image analysing each pixel individually is not enough. It is important to consider adjacent pixels as well. Neighbourhood operators combine adjacent pixels in a small area and compute a new image. After applying a neighbourhood operator, information is lost and you cannot reconstruct the original image anymore. For this reason, neighbourhood operators are called filters.

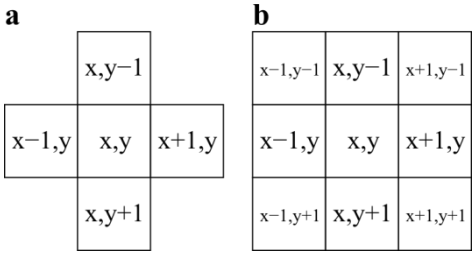


Figure 2: A 4-Neighbourhood- and a 8-Neighbourhood-Operator

These operators are able to do the following:

- Detection of structures like edges or lines
- Reconstruction, correction and restoration
- Averaging and diffusing
- Morphology

The neighbourhood operator moves pixel-by-pixel and line-by-line over the input image and computes the output image as shown in Figure 3. Close to the border of the image, when the filter mask extends over the edge of the image, we run into difficulties as we are missing some pixels. There are three common methods to compensate the missing pixels:

- Avoid extending
- Set the value to zero
- Extrapolate a value by considering the border pixels

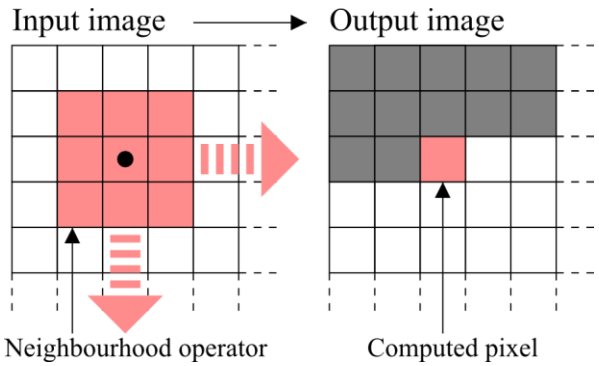


Figure 3: Applying a neighbourhood operator to an image. (grey = already computed)

Median Filter

The median filter is a good approach to remove noise which appears e.g. due to transmission errors. A low pass filter would enhance this error in the adjacent pixels. The median filter sorts the values in the neighbourhood operator in ascending order as shown in Figure 4. If the values are sorted, the value in the middle of the list is used for the output image.

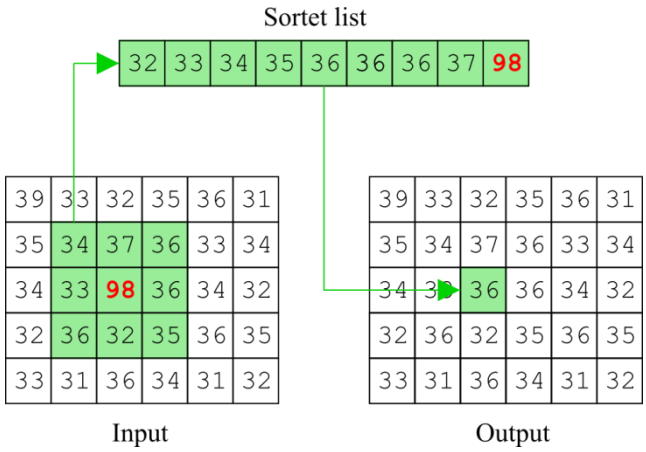
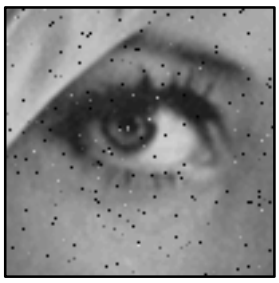


Figure 4: Idea of the Median filter [1]

Example:

We want to transmit an image of Lenna[1] in Figure 5(a). After the transmission we got some transmission errors due to bit flips in the eye region of Lenna (b). A low pass filter, which is usually used for noise removing will enhance the error(c). These outliers can be removed effectively with a median filter(d).

(a) Original image



(b) Image after transmission

(c) After applying a lowpass filter

(d) After applying median filter

[1] http://en.wikipedia.org/wiki/Lenna

Figure 5: Example for median filter

Task

Starting and Building the Virtual Platform with Platform Creator

Open a terminal and navigate to the `~/tasks/t7e/vp/` folder.

```
$ cd ~/tasks/t7e/vp
```

Check that you have the appropriate version (N-2017.12) of Platform Creator available. If not, please contact the assistant.

```
$ pct --version
$ N-2017.12
```

Optionally, have a quick look at the help. This is a good practice when starting with a new tool.

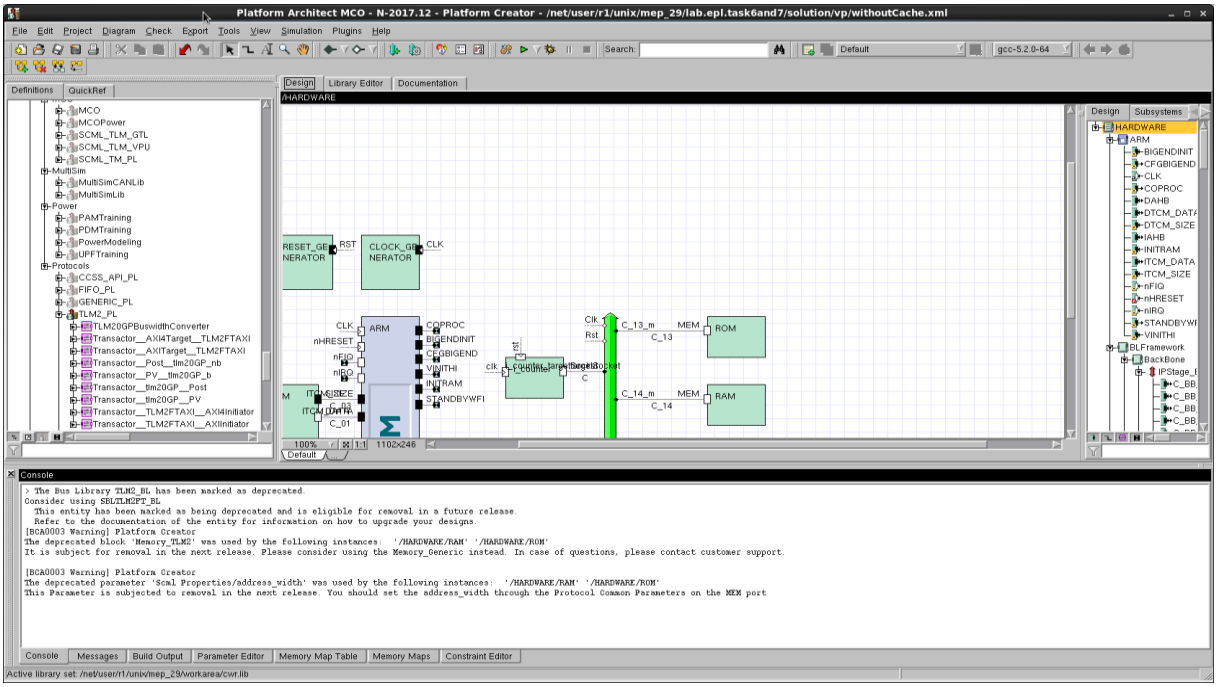
```
$ pct --help

Usage: pct [options] [file]...

Options:
  --product       start Platform Creator with the specified product mode.
                  Allowed values are VAUTH and PA.
  -v, --version   print version information and exit
  -h, --help      display this help and exit

File:
  You can pass two types of files to PlatformCreator: Tcl scripts (.tcl)
  and PlatformCreator data files (.xml). You can pass multiple script
  and/or data files, these get executed or loaded in the specified order.
```

For this task you are going to use the PA (Platform Architect) product. For convenience a project configuration is provided as an XML file which is passed as parameter to Platform Creator.

Please start the Platform Creator as follows.

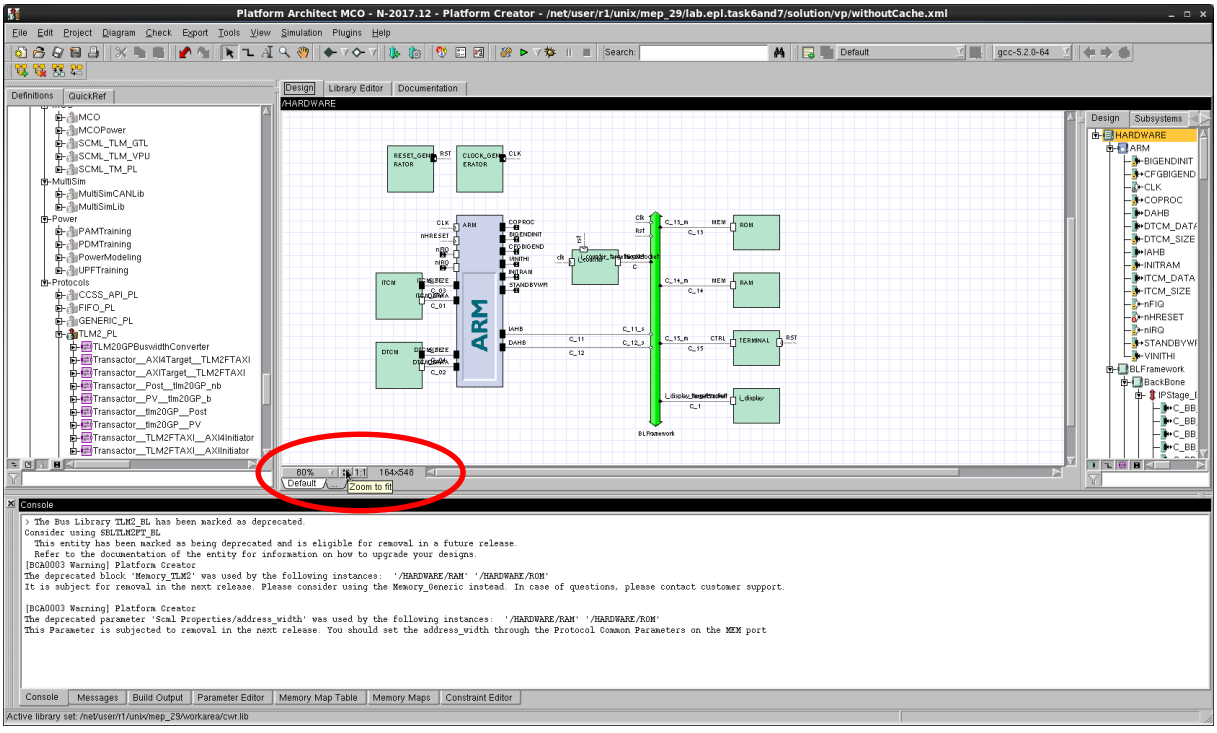
```
$ pct withoutCache.xml &
```

After a while you should get this screen:

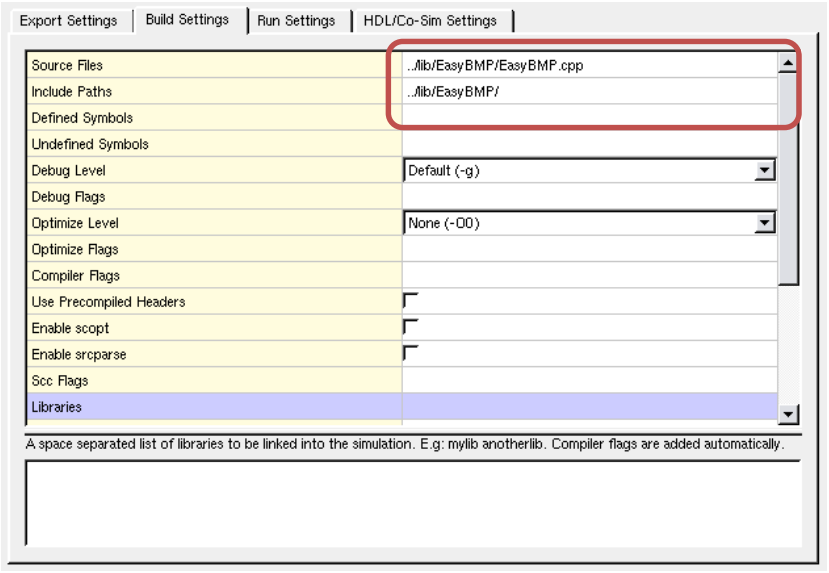


For this exercise, some warnings can be ignored.

You can use the zoom to make the image fit the available area.



Navigate to: *Export* → *Export Design* → *Build Settings* and configure the path for the EasyBMP source code as follows (if it is already configured, no problem):



This library is needed by the DISPLAY device to generate a bitmap as output.

The EasyBMP library is used by the DISPLAY component to generate a bitmap file from the data in the frame buffer. More information about the EasyBMP library can be found in http://easybmp.sourceforge.net/. **Learning the details of the EasyBMP library is not mandatory and must be done after completing the task**, if desired.

Writing the Software

Navigate with the terminal to the `~/tasks/t7e/sw/` folder and open the main.cpp with an editor of your choice (e.g., gedit, vim, nedit, etc.).

```
$ cd ~/tasks/t7e/sw
$ gedit main.cpp
```

Find the following lines in the code:

```
// ...
int main(void)
{

        initTerminal();
        // Do a profiling of this block:
        {
                Profiler p("Main");

                // Insert your code here:

        }

        flushDisplay();
        flushTerminal();

        // simulation should end here
        // automatically
        // ...
}
```

Insert your code here.

As a suggestion start with the code below.

```
// Insert your code here:
int x;
for (x = 1; x < (WIDTH -1); x++) {
        int y;
        for (y = 1; y < (HEIGHT -1); y++) {
                setPixel(x, y, IMAGE[y][x]);
        }
}
```

Now build your software.

Make sure you are in the **sw** folder.

```
$ cd ~/tasks/t7e/sw
```

Clean the old files if any (for more details open and read the Makefile).

```
$ make clean
```

Build the new version.

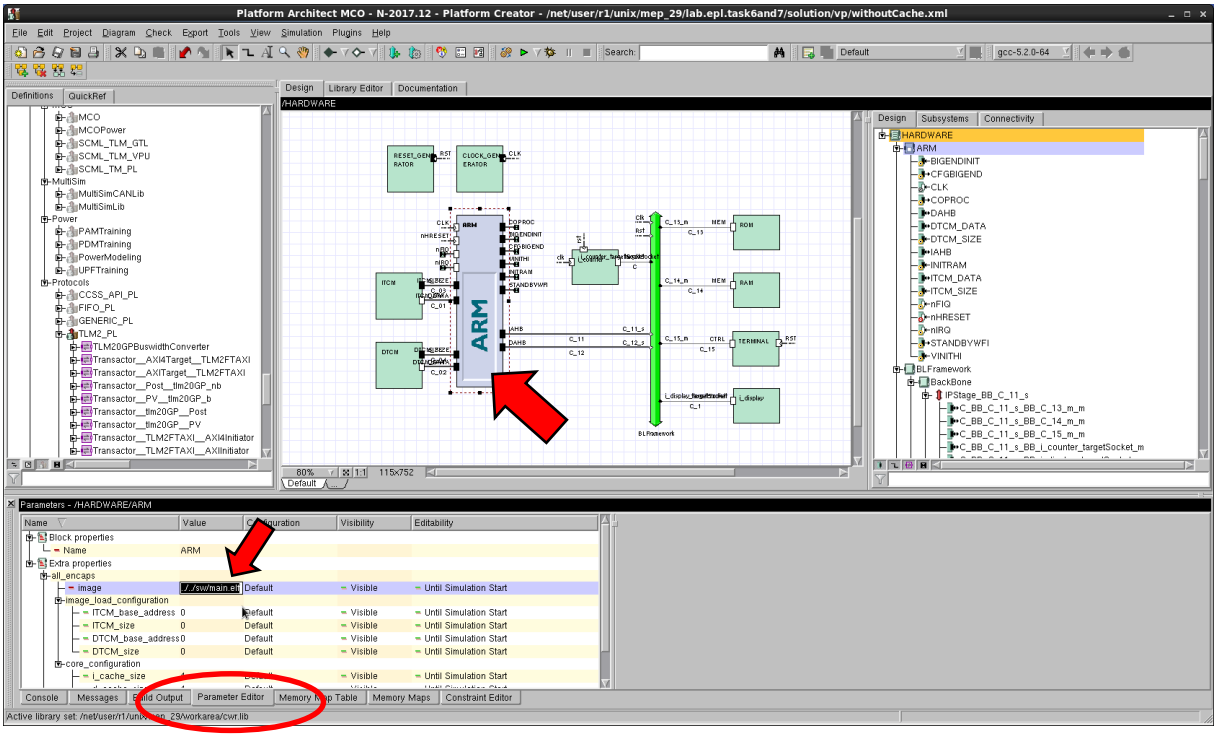
```
$ make
```

Make sure that **main.elf** was generated. You can list all files in a folder with the command below.

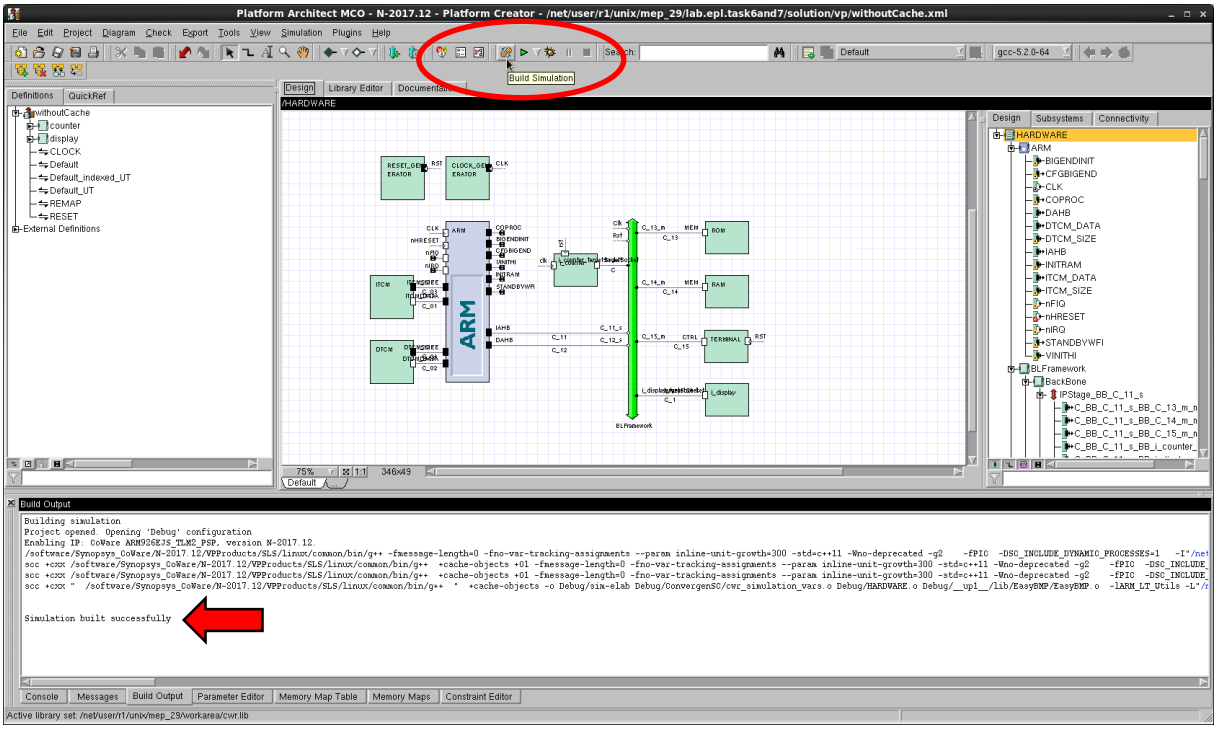
```
$ ls -l
```

To see how your software (the executable file main.elf) is specified to be loaded follow the steps below. **Please do not change this configuration**.

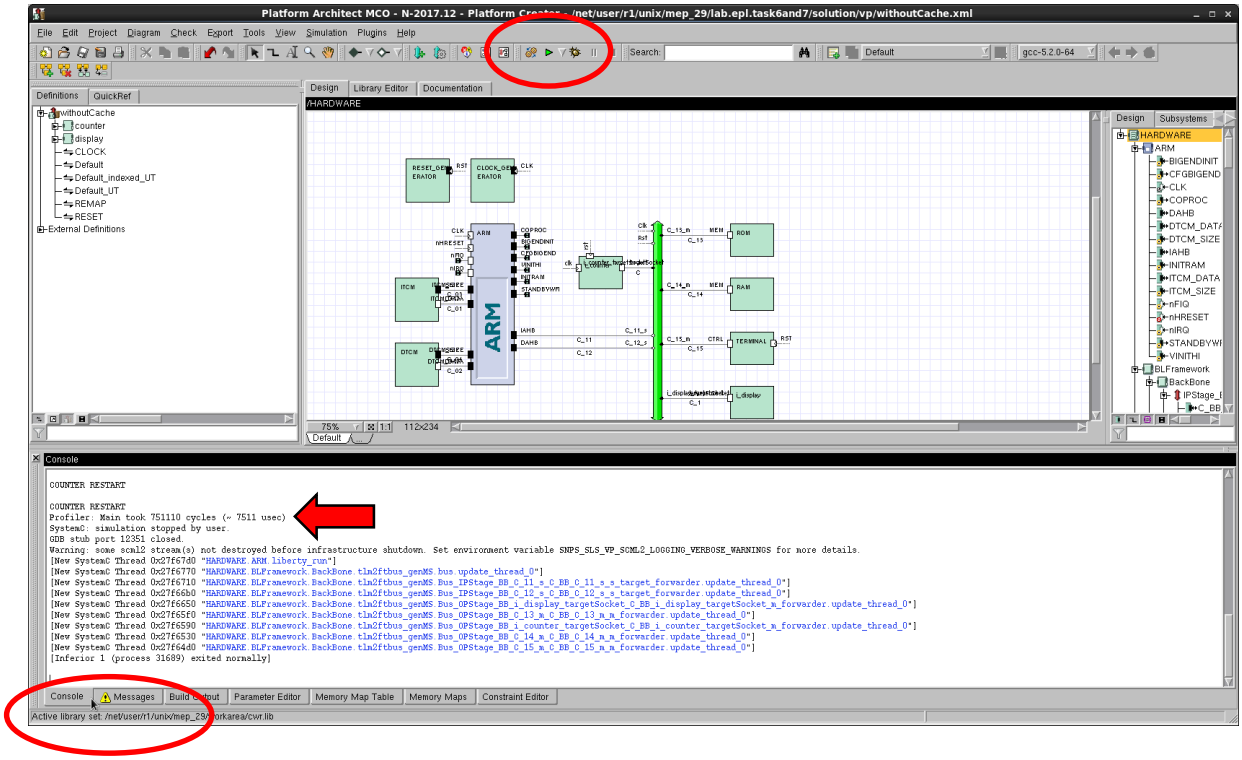
1. Click in the ARM processor model.
2. Go to the Parameter Editor tab.
3. Extra properties → all_encaps → image



Now build the simulation.



Run the simulation by pressing the green play button. Check the console tab for the profiler output to see how fast your program is.



Note that two new directories were created: **cwr** and **export** (you can use **ls** in the terminal).

```
$ ls ~/tasks/t7e/vp/*
```

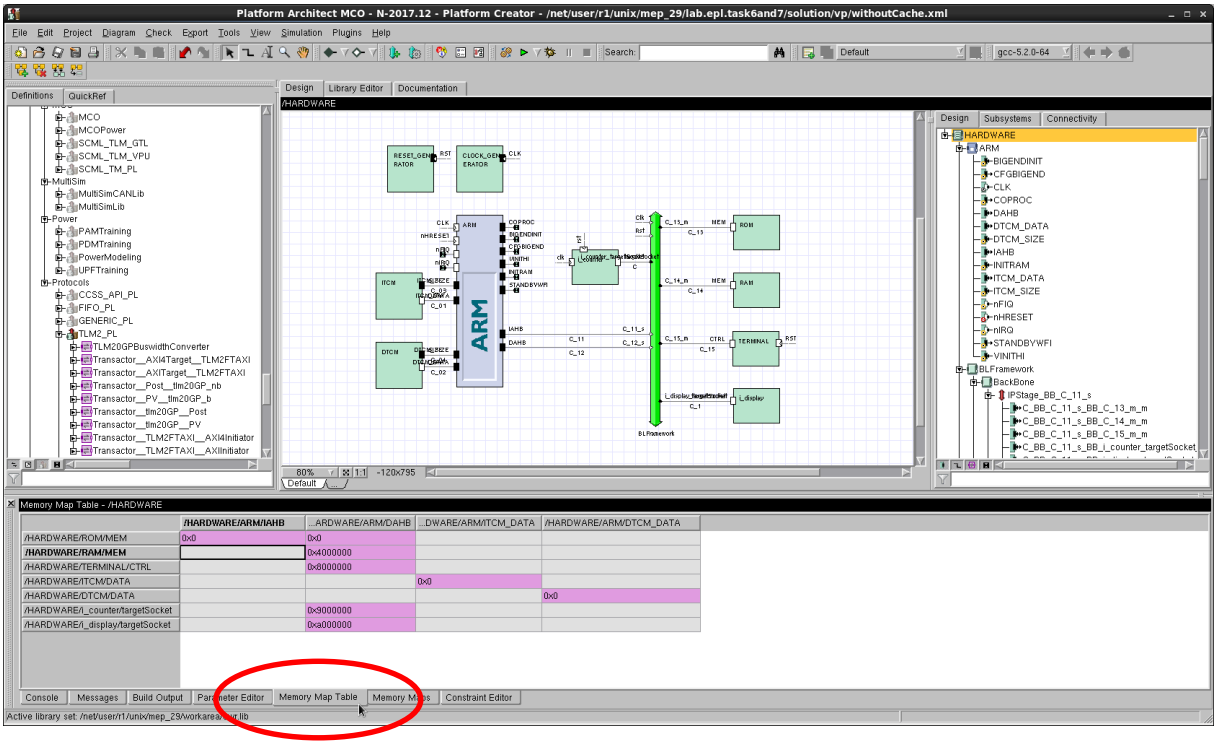
Visualize the output file generated with the command that follows.

```
$ gthumb ~/tasks/t7e/vp/export/Output.bmp
```

Since no filter was applied the output corresponds to the input image located in the RAM. The image is embedded into your program. It is compiled as part of the executable **main.elf.** The linker script specifies where it is placed in the memory. When the simulation starts the processor model uses a functionality provided by SystemC/TLM2.0 called *debug transport interface* in order to initialize the memory. Debug transport gives an initiator the ability to read and write memory in the target without causing any side-effects and without simulation time passing.

Optionally, for a better understanding you can open the linker script file and compare its contents with the memory map of your virtual platform. **This is not mandatory for completing the task**. If desired, do it after finishing the task.

```
$ gedit ~/tasks/t7e/sw/arm_boot.ld
```



Now it is time to project your median filter. When you finish the task you will be able to compare both the execution time and the quality of the output image.

Your task is to project and implement an *8-neighbourhood median filter* in C. The input picture has a size of 100x100 pixels and it is the same as in Figure 5(b). You should avoid extending/overhanging. This makes the implementation easier because you do not have to interpolate missed pixels. However, this results in some undefined pixels at the border of the output image.

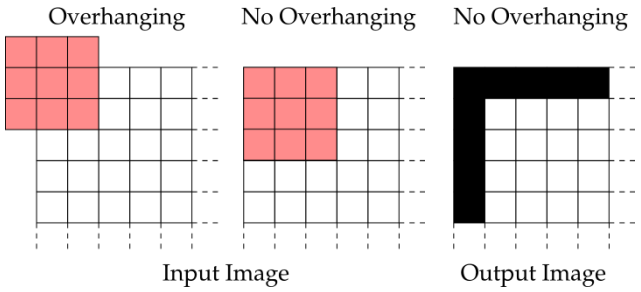


Figure 6: Different starting points of calculation

The image is <u>already</u> located in the RAM and can be accessed by a 2D array, for example:

```
unsigned char grayValue = IMAGE[y][x]; // 0 <= x <= 99, 0 <= y <= 99
```

There is a driver function for writing the filter output to the virtual display device:

```
setPixel(unsigned int x, unsigned int y, unsigned char grayValue);
```

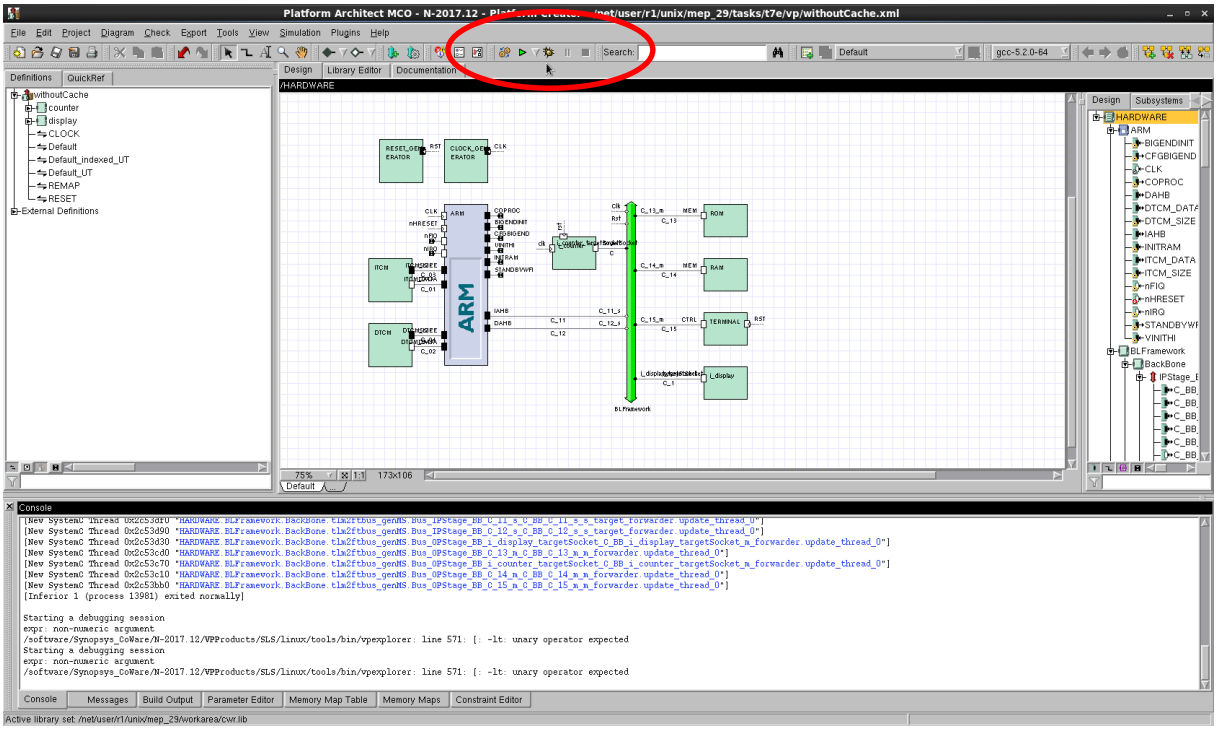
Please do not change anything else in the source code yet. Before you start coding your solution read this document to the end for learning about debugging techniques that will help you to accomplish the task. You will also learn how to accelerate your program using compiler optimization options, but you should apply optimizations only after being sure that your program works properly otherwise you may experience problems debugging it.

In the code there is a profiler used which estimates a rough execution time of your code.

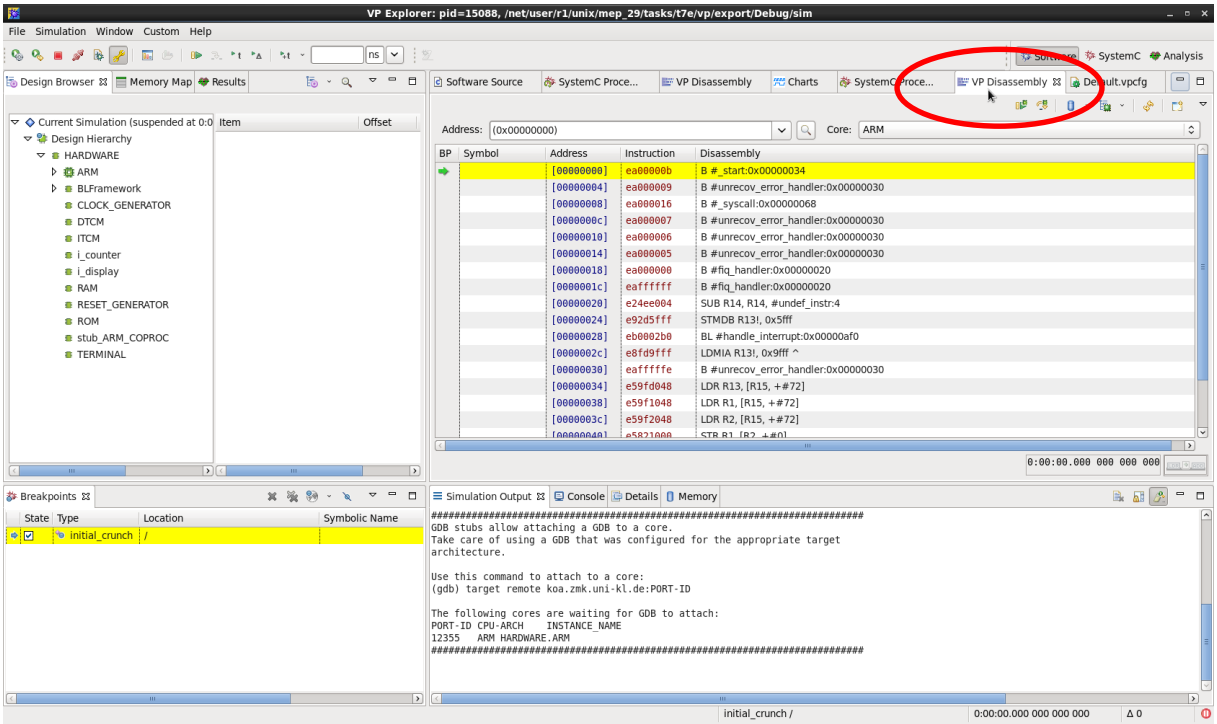
Remember that after changing your code you can compile it by typing `make` in the `~/tasks/t7e/sw/` folder as mentioned before. Changes in the source code will only take effect after compiling.

Starting a debug session in VP Explorer

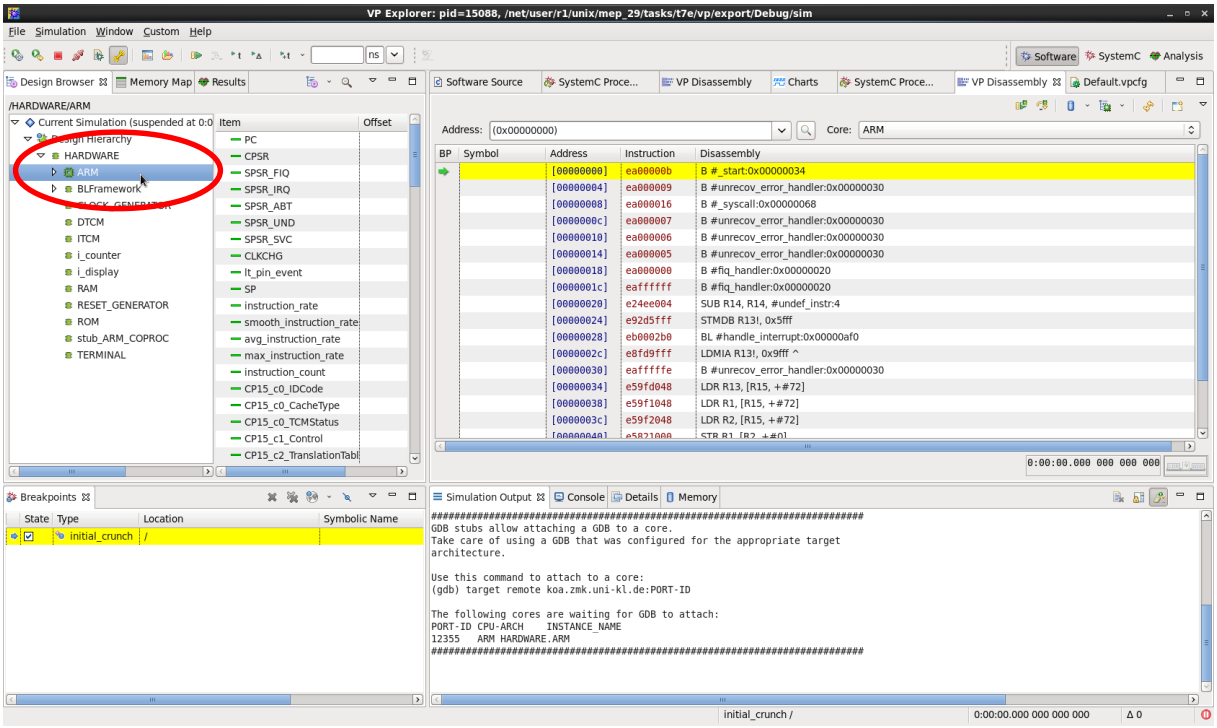
Start a Debug session in VP Explorer.



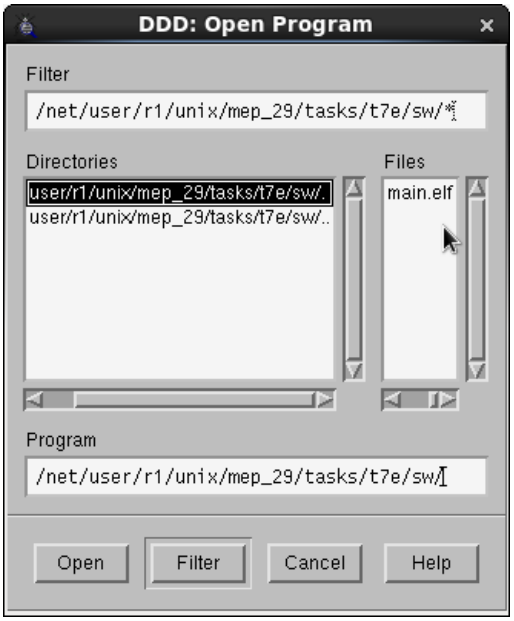
Open the VP Disassembly tab.



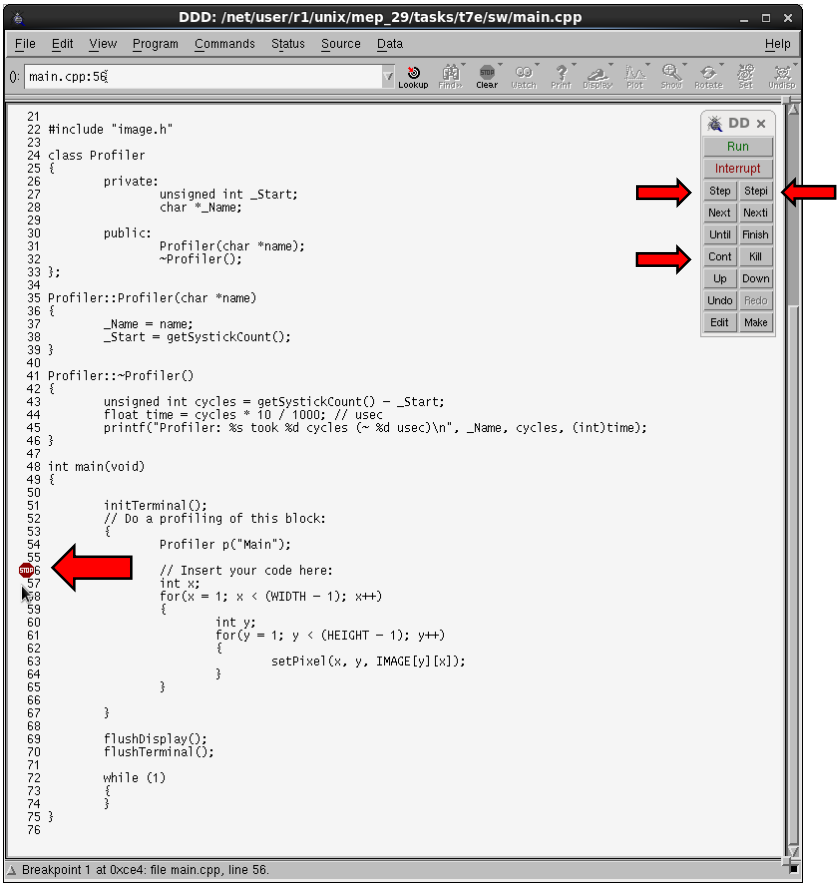
Right-click in the ARM model and launch the DDD.



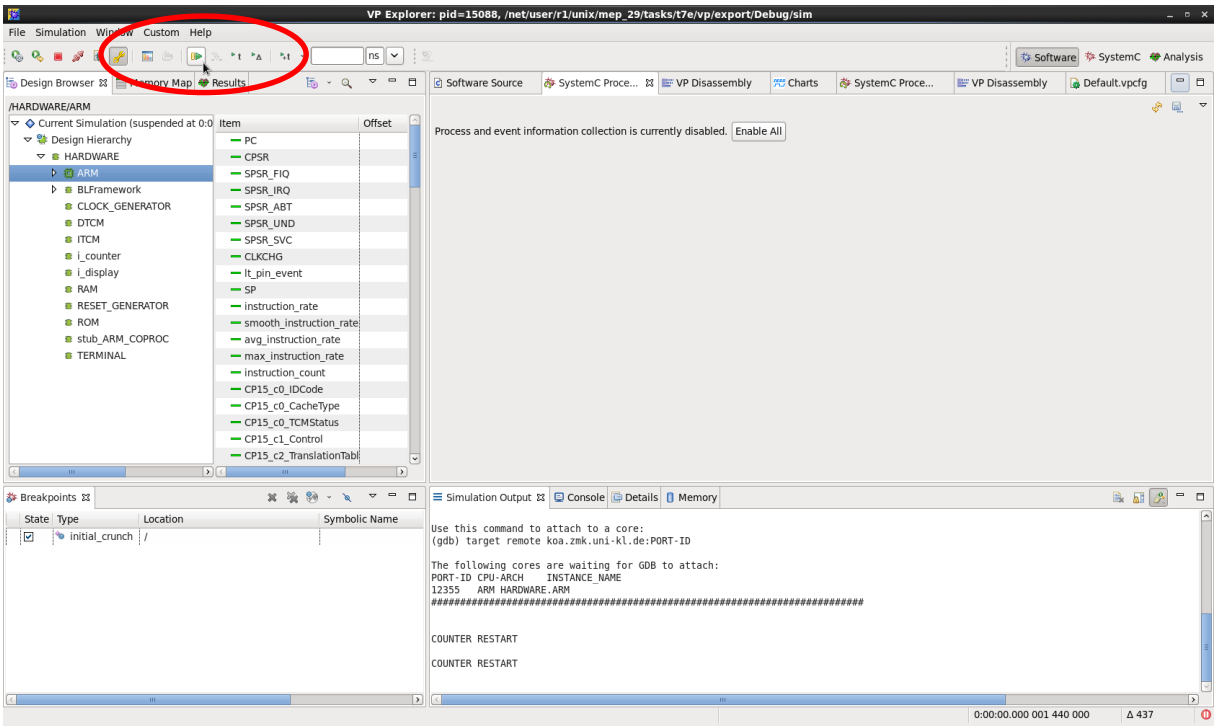
Then open your program main.elf.



You can create a breakpoint by double-clicking in the left side of the line you want to insert the breakpoint. Breakpoints are very useful when debugging.



Now you can start the Simulation by pressing cont in the DDD interface and then the play button. After a while the simulation should finish (unless you have an active breakpoint or a bug in your code…).



Check the profiler output in the simulation output to see how long your program takes to run. Try to optimize your program to see how fast you can do it.

```
Profiler: Main took 5417645 cycles (~ 54176 usec)
```

You can see the output image by using following command:

```
gthumb ~/tasks/t7e/vp/export/Output.bmp
```

When debugging with breakpoints you are able to single-step through your program's code with [F5] (Step), step over functions with [F6] (Next), set multiple breakpoints with the mouse and run the program until the next breakpoint with [F9] (Continue).

More information can be found on the DDD Website: *http://www.gnu.org/software/ddd/*

Tuning the Program:

If you have correctly implemented the program, you should have an execution time of ~150ms. You can improve the performance by changing minor things in your code.

Think about this question: *"Do you have to sort the complete Neighbourhood?"*

(Maybe you already implemented it this way. ☺)

Furthermore you can switch on the compiler optimisation:

Go to the `~/tasks/t7e/sw/` folder and open the Makefile with an editor of your choice.

Change the following lines:

```
CFLAGS   = -g -O0 -lunistd -I .
CXXFLAGS = -g -O0 -lunistd -I . -fno-exceptions
```

into:

```
CFLAGS   = -g -O3 -lunistd -I .
CXXFLAGS = -g -O3 -lunistd -I . -fno-exceptions
```

Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program. The compiler performs optimization based on the knowledge it has of the program [4].

- `-O0:` Reduce compilation time and make debugging produce the expected results. This is the default.
- `-O1:` Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
- `-O2:` Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed trade-off.
- `-O3:` Optimize yet more. `-O3` turns on all optimizations.

Please observe the difference in Runtime!

Summary

What are the expected results and what do you have to know for the acceptance test?

- Basic knowledge about Virtual Prototyping including SystemC and TLM.
  (Read *Introduction to Virtual platforms*!)
- A flow chart or Nassi-Shneiderman diagram of your algorithm.
  (You have to explain it to the examiner!)
- The actual implementation, which has to work correctly!
  (Please take note of the runtime of the non-optimised and optimised version!)

References

[1] B. Jähne, Digital Image Processing, Springer, Ed., 2005.

[2] M. Pandit, Methoden der Digitalen Bildverarbeitung für die Anwendung, VDI, Ed., 2001.

[3] M. Jung, 2010. [Online]. Available: http://jung.ms/media/uni/thesis.pdf.

[4] April 2012. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.