Task 7E

A VHDL Cache in a Virtual Platform Co-Simulation
Dr. Matthias Jung,Dr. Stefan Weithoffer, Éder F. Zulian (2012-2017)

Organisational Matters

- Use the server koa for this task
- 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 [1]

A memory hierarchy in a computer or a microprocessor consists of multiple levels of memory with different speeds and sizes. The faster memories are more expensive per bit than the slower memories and thus smaller. Today, there are three primary technologies used in building memory hierarchies. Main memory is implemented with DRAM (dynamic random access memory), while levels closer to the processor (caches) use SRAM (static random access memory). DRAM is less costly per bit than SRAM, although it is substantially slower. The third technology, used to implement the biggest and slowest level in the hierarchy, is magnetic disk. The access time and price per bit vary widely among these technologies.

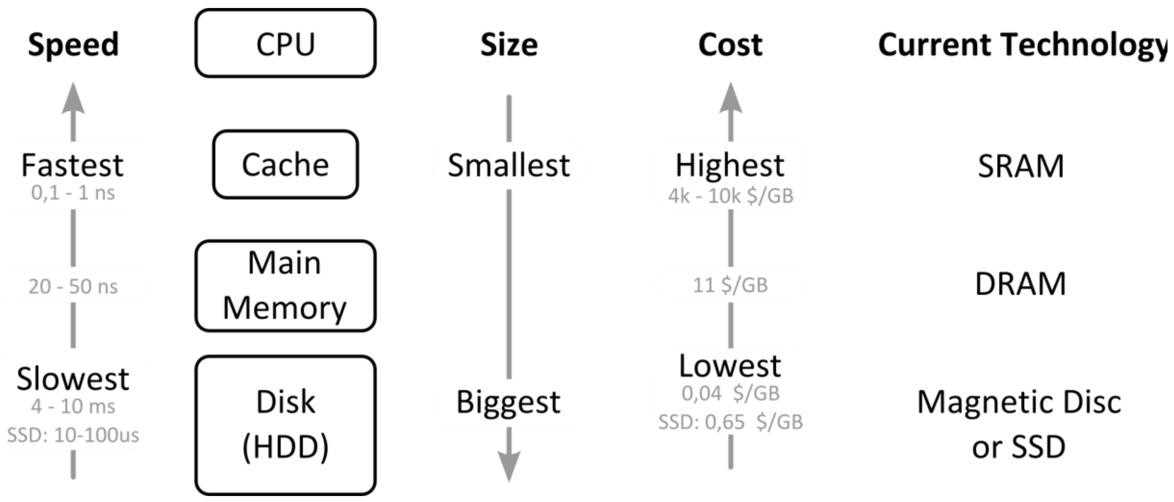


Figure 1: The basic structure of a memory hierarchy, values from 2004 [1]

The task of a cache is to provide copies of the currently active main memory segments during the program execution allowing a fast access to the data.

A cache takes the advantage of two principles of Locality:

- *Temporal Locality:* The principle stating that if a data location is referenced then it will tend to be referenced again soon.
- *Spatial Locality:* The locality principle stating that if a data location is referenced, data locations with nearby addresses will tend to be referenced soon.

Special techniques and strategies are needed for addressing and also for the loading and storing of the data, to fully exploit these principles. This functionality is realized completely in hardware and is not visible to the processor.

In this task the Virtual Platform of Task 6E is extended with a VHDL cache module, which has to be finished by you.

Overview: Cache Architectures

This chapter will give you a quick overview over the most common cache architectures, which are used in applied processor engineering. Cache architectures can be regarded as a collection of implementation strategies, which can be generally categorised in the following types:

- Where the data from the main memory is stored in the cache (block placement)?
- How the data is found if it is placed in the cache?
- Which cache block is replaced after a cache-miss?
- What happens in case of a memory write access (integrity of the data)?

Cache Structures and Block Placement

Direct mapped cache

The direct mapped implementation represents one of the simplest cache implementations due to the small degree of freedom of the block placement and low complexity of the hardware.

The cache memory is organised in **cache rows** containing several words, which form the so-called blocks in the memory. In the figure each row has a **data block** (or **cache line**) of four words (the actual data fetched from the main memory). Additionally, to each data block a **tag** is written which is used to map the data precisely to one address in the main memory.

The placement of blocks is organised with an incrementing address according to the lines in the cache memory in a circular way. After placing the last line of the cache memory, the next memory block is placed into the first line. This implies that a unique mapping of a data block in main memory to a cache block exists. For every memory access only one line of the cache has to be considered.

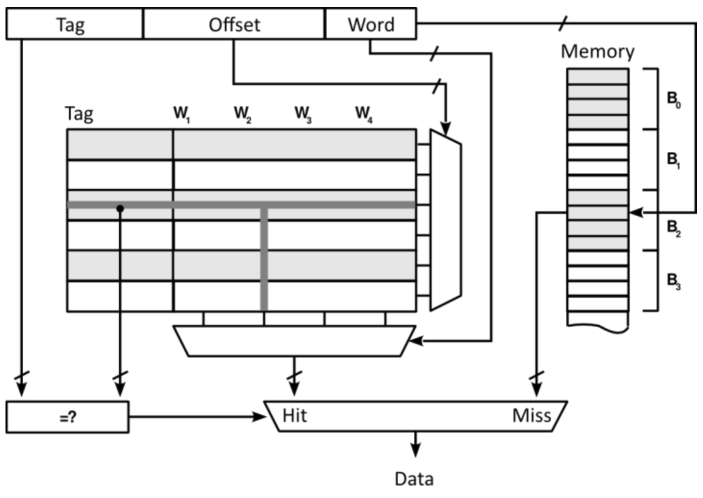


Figure 2: Functionality of a direct mapped cache

On a cache access the **physical address** is divided into the parts **tag – offset – word**. The offset part is used for addressing the cache line. The stored tag of the selected line is compared to the tag part in the address. On a match, the word of the cache line, which is indexed by the word part, is read. The bit-width of the offset depends on the number of cache lines. Capacity of the cache depends on the number of cache lines and number of words per cache line.

Fully associative cache

The fully associative cache represents the most complex implementation, but it offers the highest degree of flexibility in the implementation of different block replacement strategies.

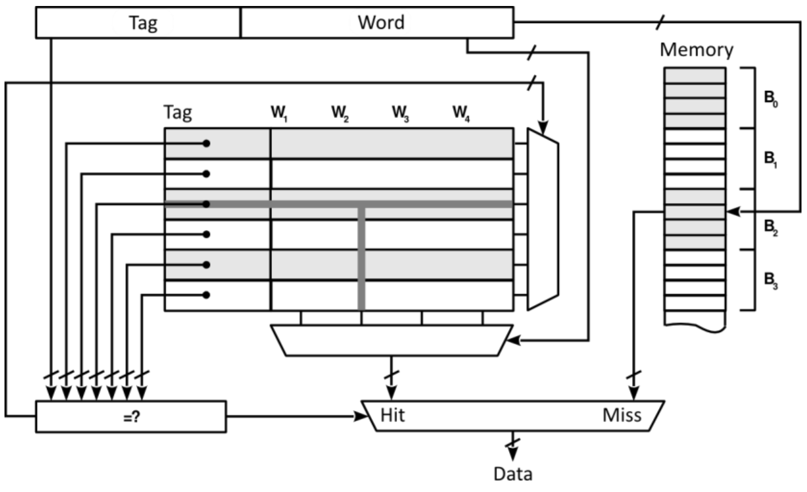


Figure 3: Functionality of a fully associative cache

The placement of memory blocks in the cache is known a priori for direct mapping approach; the fully associative cache allows every line to be placed individually. The complexity for the address calculation for the addressed data increases because there are no restrictions for line/block mapping. The upper part of the incoming address on a cache access is interpreted again as a tag which must be compared with all tags of the cache. Therefore, the tag is representing the physical address of the first data word inside the block.

If there is match of the tags, the corresponding line is selected and the requested data word indexed by word is read.

Due to the very high implementation complexity a pure fully associative cache is very rarely used; the rare realizations feature a small capacity.

K-way associative cache

The k-way associative cache represents a compromise between the extremes in complexity of the fully associative and the direct mapped cache.

The cache lines are divided into sections; so called **sets**. **The number k of lines, which are packed in one set, determines the associative rank k**.

According to the direct mapped concept, an **Offset** part of the memory address **serves as Set index**. This marks the unique position of a Set in the cache memory. The tag array in the selected set is now fully associatively compared with the input Tag. Since the fully associative search is limited to size of the Set with k lines, only k comparisons have to be executed in parallel (requiring **k** comparators).

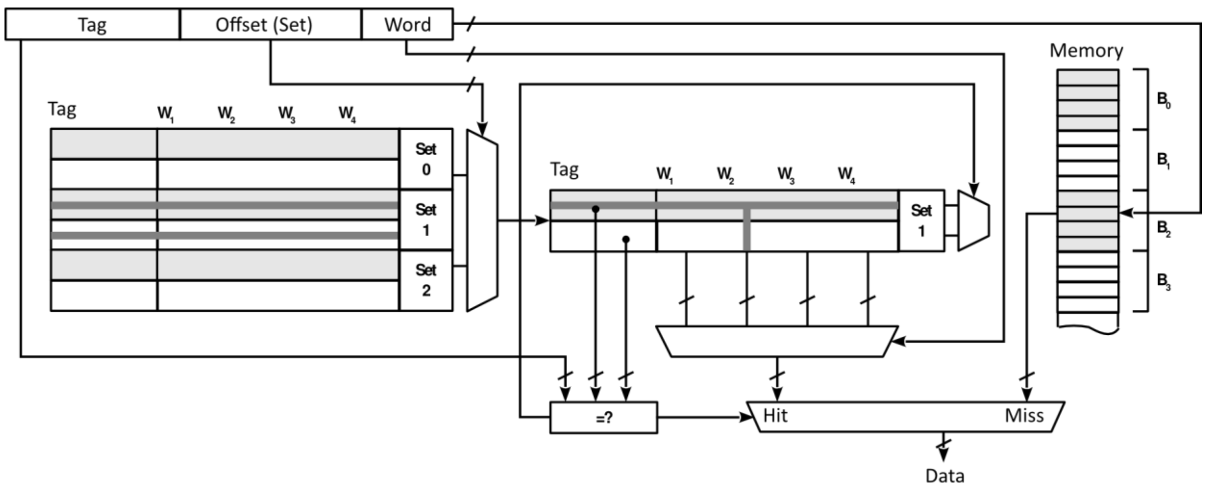


Figure 4: Functionality of a k-way associative cache

The implementation of this structure in hardware needs exactly the same amount of memory modules with read-/write-ports as the number of lines (k) in a Set. This enables the reading of all lines in a set in parallel by applying the Set-address (Offset). The Tags are compared by k parallel comparisons directly.

This k-way associative structure is the most commonly used implementation. The value of k is mostly between 2 and 6, which gives a good trade-off between performance and circuit complexity.

This structure is therefore used for the data cache implementation in this task.

Block Replacement Strategies and Data Consistency

A significant task of the cache control unit is ensuring data consistency from the processors point of view. The processor has to access the newest version of all data at all times. Inconsistencies between cache and main memory content can generally occur by processor write operations which creates the need for the implementation of an appropriate update strategy (only applies to data caches because only here the content can be changed).

In such a case either the cache, the main memory or both are rewritten with the new data. In any case it must be ensured that the subsequent read operations always access the latest version of the data instead of outdated data.

Write-Through-Procedure

If the Write-Through-Procedure is applied, the main memory is updated regardless if a write-hit or a write-miss of the cache occurred. On a write-hit, the cache entry is updated as well, which ensures the equivalence of the contents of main memory and cache at all time.

Drawback of this method is the loss of the cache's speed advantage on write-operations because the processor has to wait for the relatively slow main memory access to finish the write operation. This is often mitigated by using special write-buffer so that the next cache access can occur before the write operation to the main memory is finished. An advantage results only if the following access is a read operation which can be satisfied by the cache.

If only the main memory is updated on a write-miss and not the cache, it is called the "no allocation" variant (data is loaded into the cache on read-misses only). This variant is most commonly used. More rarely is "write-through-allocation" which loads data block from the missed-write location into cache first and then updates the data in the cache and memory (avoids so called "address aliasing").

Copy-Back-Practice

In opposite to Write-Through only the cache entry is updated on a write access, so the cache keeps it efficiency on a write-hit. The main memory is updated only on a block replacement by copying the changed block to memory before it gets replaced. A changed block can be identified by an additional so called dirty bit in every cache line.

To cut the waiting time on write operations on the main memory, write buffers are commonly used (buffered copy back).

Replacement-strategies on cache-miss

Due to the limited capacity of the cache only a small part of the address space is kept in the memory. Temporal locality and the high likelihood of a subsequent memory access targeting a neighbouring address are essential for the cache principle to work. Therefore the address space, which is kept in the memory, must be constantly adapted to the CPU context. If an address block is loaded from the main memory, the cache line which is to be used must be determined (except in the direct-mapping implementation). In practice the two major following types of conflict resolution exist:

Least recently used (LRU)

Each cache line is extended with age information. This is commonly realized with a counter like implementation. This counter is incremented if the current cache access does not access this line. The counter on the loaded line is set to zero and therefore marked the “youngest” line.

In case of an imminent block change the line with the least recent access (highest counter) is removed. In respect to the associativity of the cache the need for administration of these age information can be excessively high.

Random

Block replacement happens randomly in this procedure. This implementation is simpler than the LRU-procedure but has a worse overall-performance because it does not exploit the locality principle.

For the data-cache implementation the random procedure is to be used due to its simplicity.

Task Description

Figure 5 shows the Virtual Platform used in the previous task after extension with a data-cache and the memory controller unit. The external memory is from now on connected with the memory controller unit, which is not shown in the Figure. The caching principle is transparent to the CPU-core, so no changes to the related interfaces are allowed.

This given structure is fixed in the course of this task. Changes are not allowed.

The objective of this task is the implementation of the control logic for the data cache in VHDL. The cache is to be designed as a 2-way associative memory with write-through write strategy and a random replacement strategy. Additionally, for each write operation the data, which address is in the cache at the time of the access, is updated. Therefore no explicit valid-bit is needed and no write back is needed on replacement.

Data-cache module (dcache.vhd)

In the given code frame you will find the declaration of global signals and the instantiation of two cache ram modules (`set0_ram, set1_ram`) with synchronous write port and asynchronous read port. The controlling of these input signals shall happen by combinational (not clocked) processes.

The generation of pseudo random numbers happens in a clocked process which implements a linear feedback shift register (LFSR).

Read-process (not clocked):

In this process it must be determined whether a read access by the CPU-core can be satisfied by the cache (hit detection). If the requested data exists in the cache memory, a complete and aligned data word is transmitted back on the data read port (dc_rdata). For the aligning the auxiliary function `align` is to be used.

k

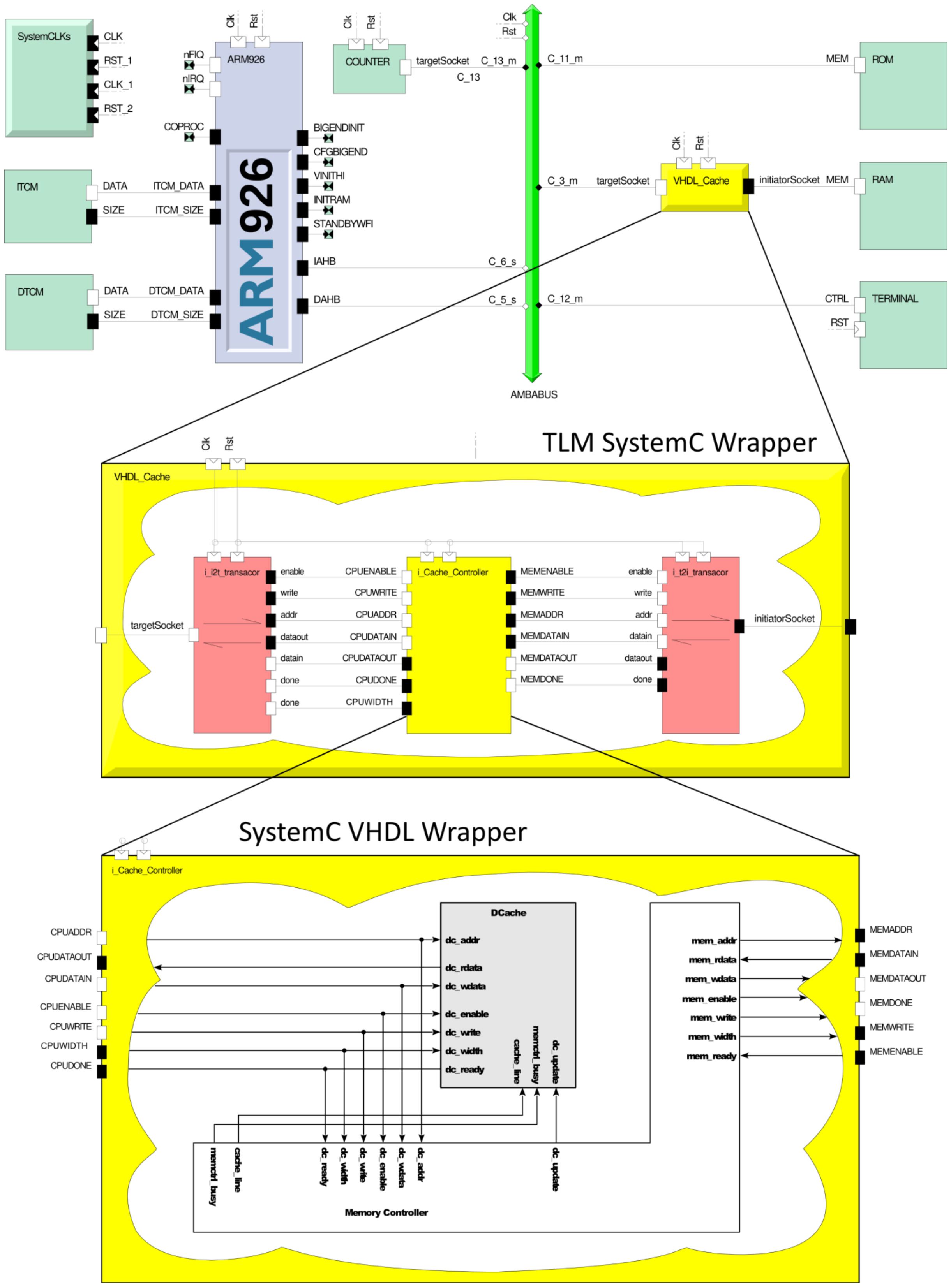


Figure 5: Structure of the VHDL in the Virtual Platform

Update-process (not clocked):

In the required implementation the cache as well as the main memory is updated on a write access by the CPU-core. There are two sources which have to be taken into account as source for the cache-RAM:

CPU-core:

On a write-hit the entry in the cache line must be updated accordingly to its position. Use the auxiliary function update.

Memory controller:

On a write-miss a cache line is discarded and will be reloaded from the main memory by the memory controller. The data cache receives always complete cache lines, so aligning of the data is omitted.

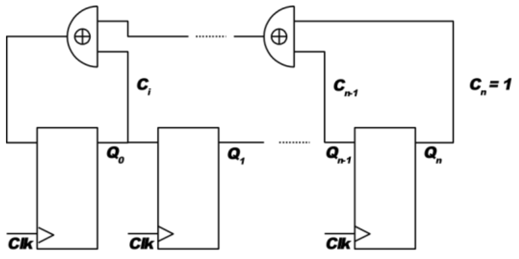
The selection of the discarded cache line is done randomly. To generate the pseudo random numbers the so called *Linear Feedback Shift Register* (LFSR) is used (see below).

LFSR-process (clocked):

You have to implement an 8 bit width LFS register (signal: `lfsr`) in this process. The LSB (Bit 0, `rand_bit`) is used to decide the replacement behaviour. Use this characteristic polynomial:

$$\mathrm{P}(x) = 1 \oplus x^1 \oplus x^6 \oplus x^7$$

Figure 6 shows the abstract implementation principle of an LFSR to generate pseudo random bit sequences. The implementation consists of n flip-flops which are organised as a shift register. The stored values are looped back to the input via several EXOR gates. Which position is used to get the input signal is specified by the characteristic polynomial. The binary coefficient c decides if the stored value of the stage *i* is considered (c = 1) or not (c= 0). For a detailed description see reference [**2**].



$$P(x) = c_0 \oplus c_1 x^1 \oplus \ldots \oplus c_{n-1} x^{n-1} \oplus x_n$$

Figure 6: circuit of a LFSR

Auxiliary functions (cache_support.vhd)

Two functions are implemented here which simplify the positioning and aligning of the read and written data from/to a cache line. The hardware which is generated during synthesis will be a complex multiplexer structure.

Function align:

Selects a data word in a cache line `line_in` at the position of `line_select` and aligns them according to the width of the access (`data_width`) into the output word at a lower place.

```
function align (
line_in : in std_logic_vector( 0 to bw_cacheline -1);
line_select     : in std_logic_vector( 1 downto 0);
word select     : in std_logic_vector( 1 downto 0);
data_width      : in Mem_width
) return dlx_word;
```

Function update:

The data `word_in` is aligned in the cache line `line_in` dependent on the width of the written data (`data_width`). The correct number of bits is written to the position given by `line_select` and the byte position is given by `word_select`.

```
function update (
line_in : in std_logic_vector( 0 to bw_cacheline -1 );
word_in : in dlx_word;
line_select     : in std_logic_vector( 1 downto 0 );
word_select     : in std_logic_vector( 1 downto 0 );
data_width      : in Mem_width;
) return std_logic_vector;
```

Simulation Techniques:

Simulation of the cache system is realised as a co-simulation that is done with *SystemC Explorer* and *Modelsim*. There are two different modes of simulation:

- **HDL Simulator Hidden:** *Modelsim* is not shown; it runs in the background and is totally controlled by the *SystemC Explorer*. You can only analyse the behaviour of the external signals of the cache component.
- **HDL Simulator Interactive:** *Modelsim* and *SystemC Explorer* are both executed. The internal signals of the cache can be analysed with Modelsim. Only one of the tools can take control over the simulation, the other tool gets blocked. So if, for instance, Modelsim seems to hang, you have to simulate in the SystemC Explorer some additional 10ns.

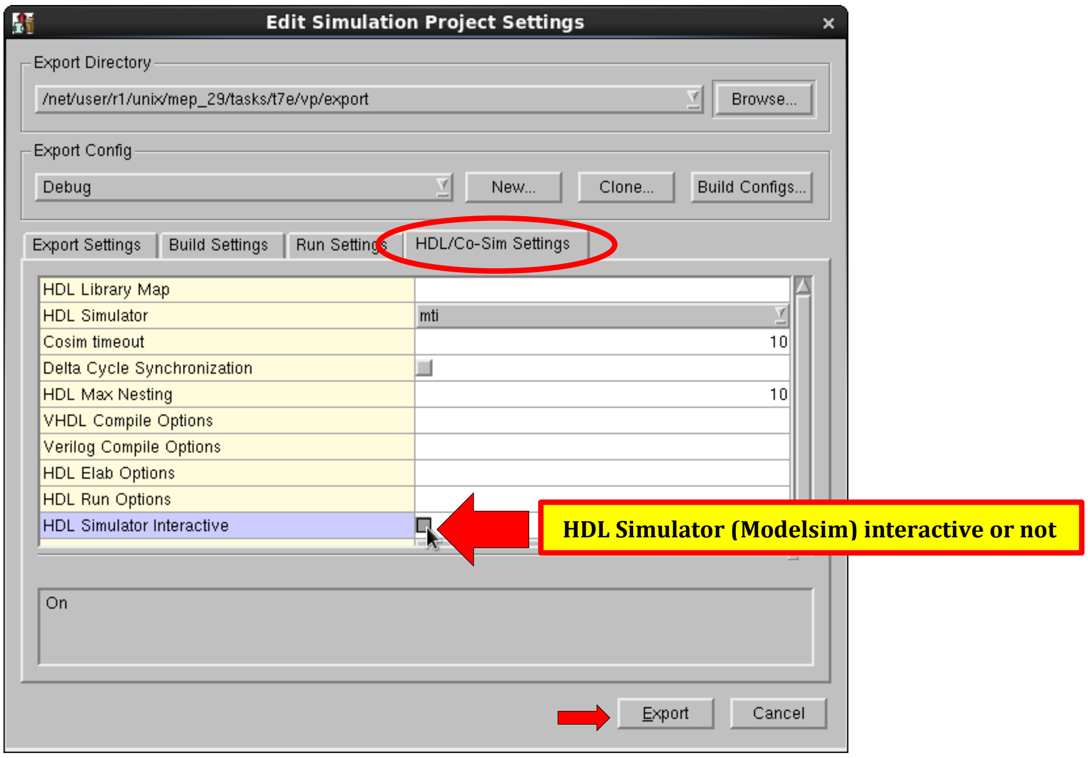
Starting the Work

Please use the same files as in Task 6E. To start the platform go to the `~/tasks/t7e/vp/` folder and execute the script as follows.

```
$ cd ~/tasks/t7e/vp
$ ./startWithCache.sh
```

The Platform Creator should open now.

In the settings of Platform Creator you have to insert following values:



Edit Simulation Project Settings

Export Directory: /net/user/r1/unix/mep_29/tasks/t7e/vp/export

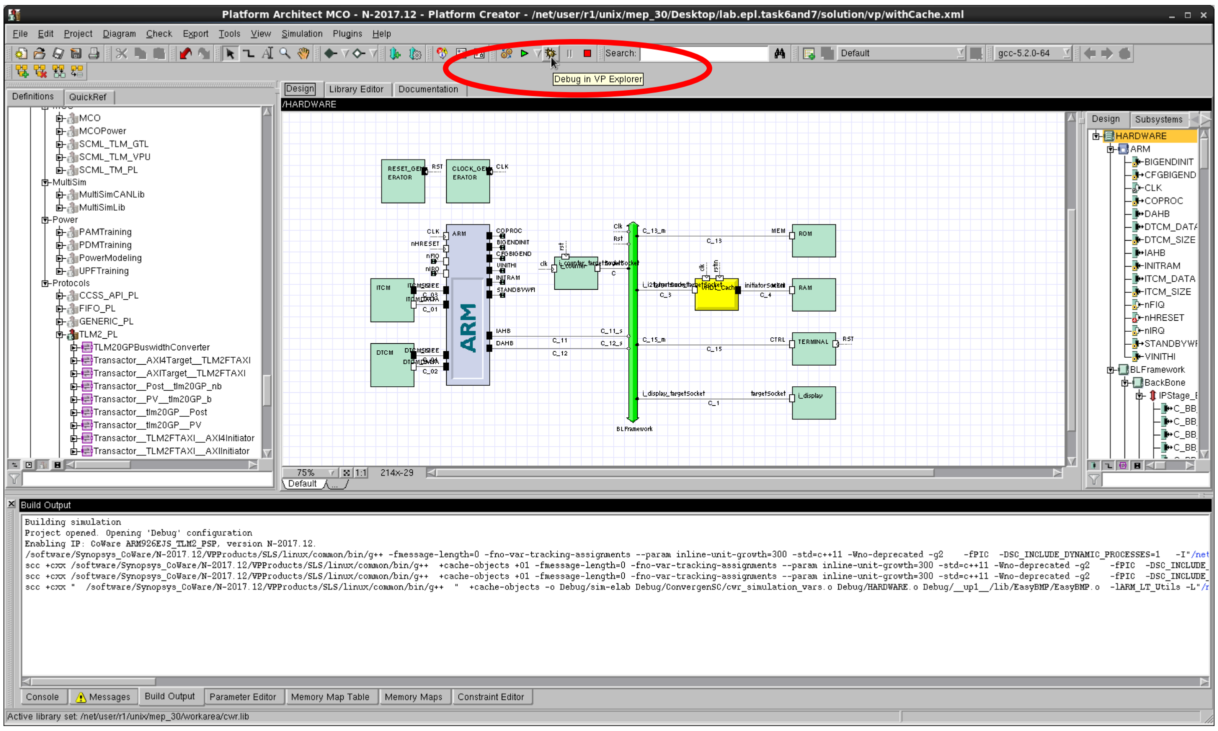
Export Config: Debug

HDL Library Map	
HDL Simulator	mti
Cosim timeout	10
Delta Cycle Synchronization	
HDL Max Nesting	10
VHDL Compile Options	
Verilog Compile Options	
HDL Elab Options	
HDL Run Options	
HDL Simulator Interactive	

On

HDL Simulator (Modelsim) interactive or not

Now you can start writing your VHDL code. All the files are located at the `~/tasks/t7e/vp/lib` folder. After finishing your implementation, you can build the simulation and start the debugging by calling the VP Explorer tool.



Platform Architect MCO - N-2017.12 - Platform Creator - /net/user/r1/unix/mep_30/Desktop/lab.epl.task6and7/solution/vp/withCache.xml

Build Output

```
Building simulation
Project opened. Opening 'Debug' configuration
Enabling IP: CoWare ARM926EJS_TLM2_PSP, version N-2017.12.
/software/Synopsys_CoWare/N-2017.12/VPProducts/SLS/linux/common/bin/g++ -fmessage-length=0 -fno-var-tracking-assignments --param inline-unit-growth=300 -std=c++11 -Wno-deprecated -g2    -fPIC  -DSC_INCLUDE_DYNAMIC_PROCESSES=1   -I"/net
scc +cxx /software/Synopsys_CoWare/N-2017.12/VPProducts/SLS/linux/common/bin/g++  +cache-objects +01 -fmessage-length=0 -fno-var-tracking-assignments --param inline-unit-growth=300 -std=c++11 -Wno-deprecated -g2    -fPIC  -DSC_INCLUDE_
scc +cxx /software/Synopsys_CoWare/N-2017.12/VPProducts/SLS/linux/common/bin/g++  +cache-objects +01 -fmessage-length=0 -fno-var-tracking-assignments --param inline-unit-growth=300 -std=c++11 -Wno-deprecated -g2    -fPIC  -DSC_INCLUDE_
scc +cxx "  /software/Synopsys_CoWare/N-2017.12/VPProducts/SLS/linux/common/bin/g++  "  +cache-objects -o Debug/sim-elab Debug/ConvergenSC/cwr_simulation_vars.o Debug/HARDWARE.o Debug/__up1__/lib/EasyBMP/EasyBMP.o  -lARM_LT_Utils -L"/r
```

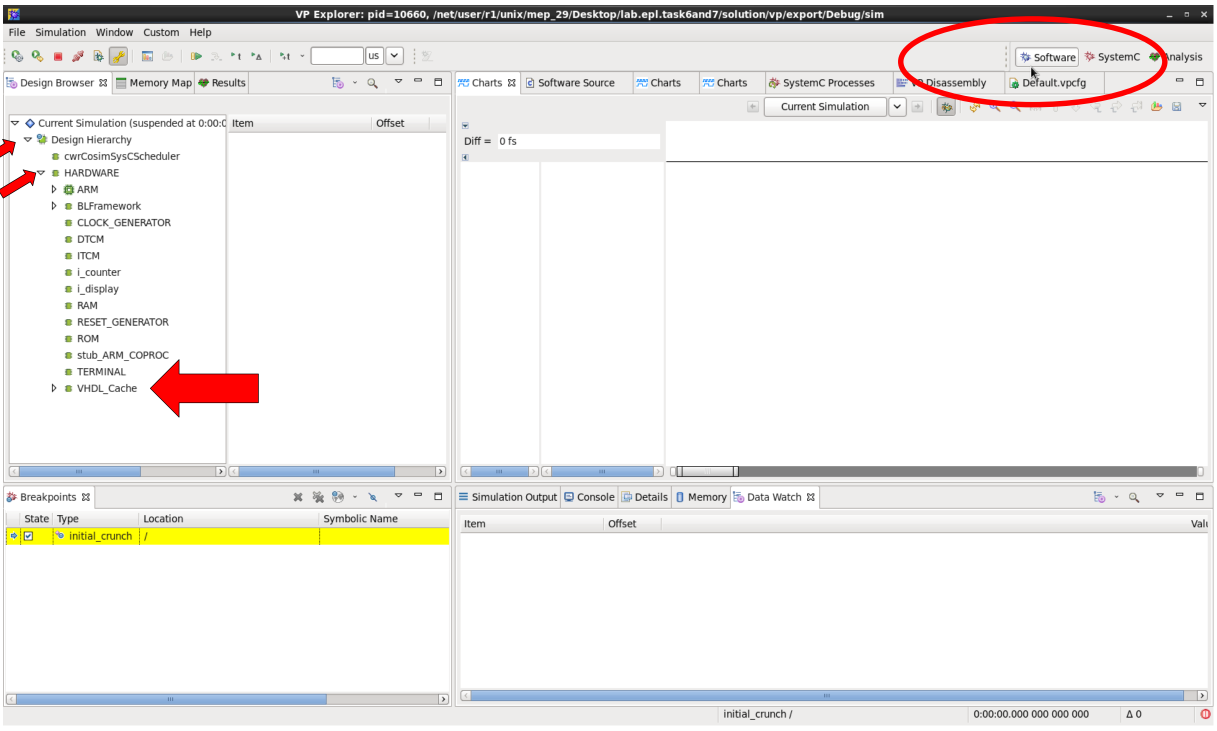
Active library set: /net/user/r1/unix/mep_30/workarea/cwr.lib

First try with Modelsim in the background (ensure the proper HDL simulator interactive configuration). In VP Explorer you can start the simulation by pressing the Play button. Please open in the *Design Browser View*:

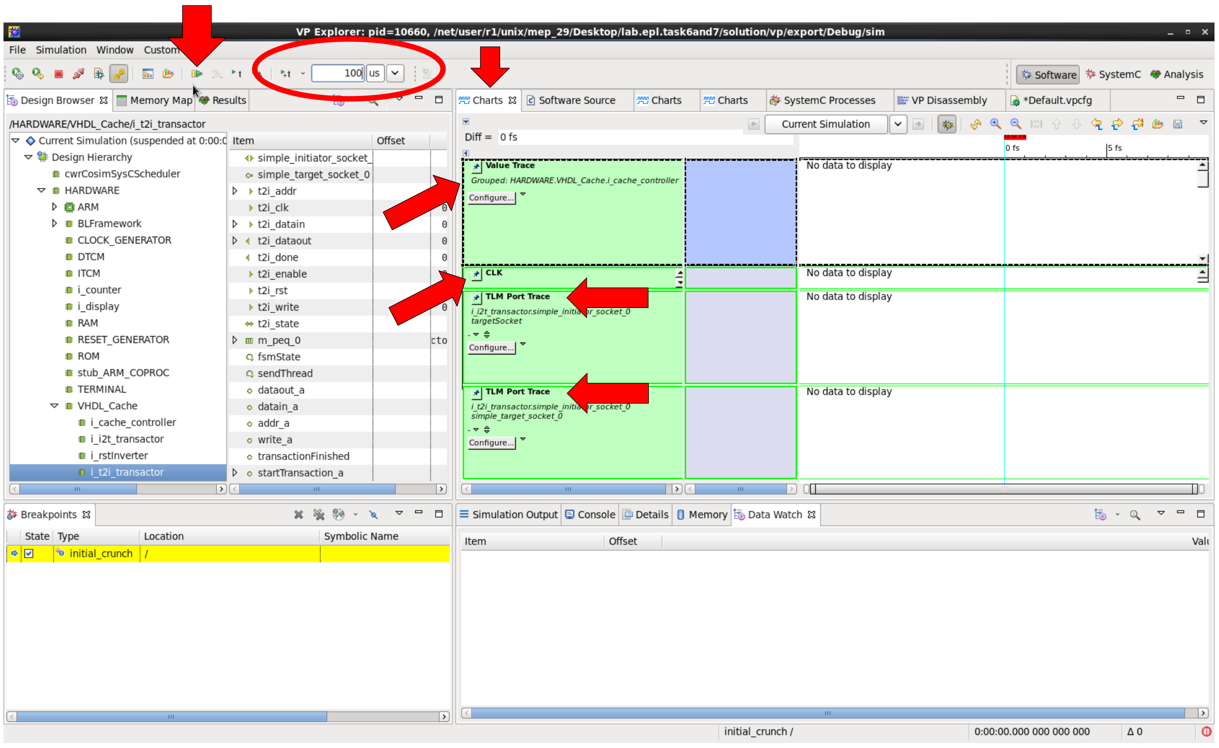
Current Simulation → Design Hierarchy → HARDWARE → VHDL_CACHE

Do a right click on it and select *Analysis* → *Display Charts*:

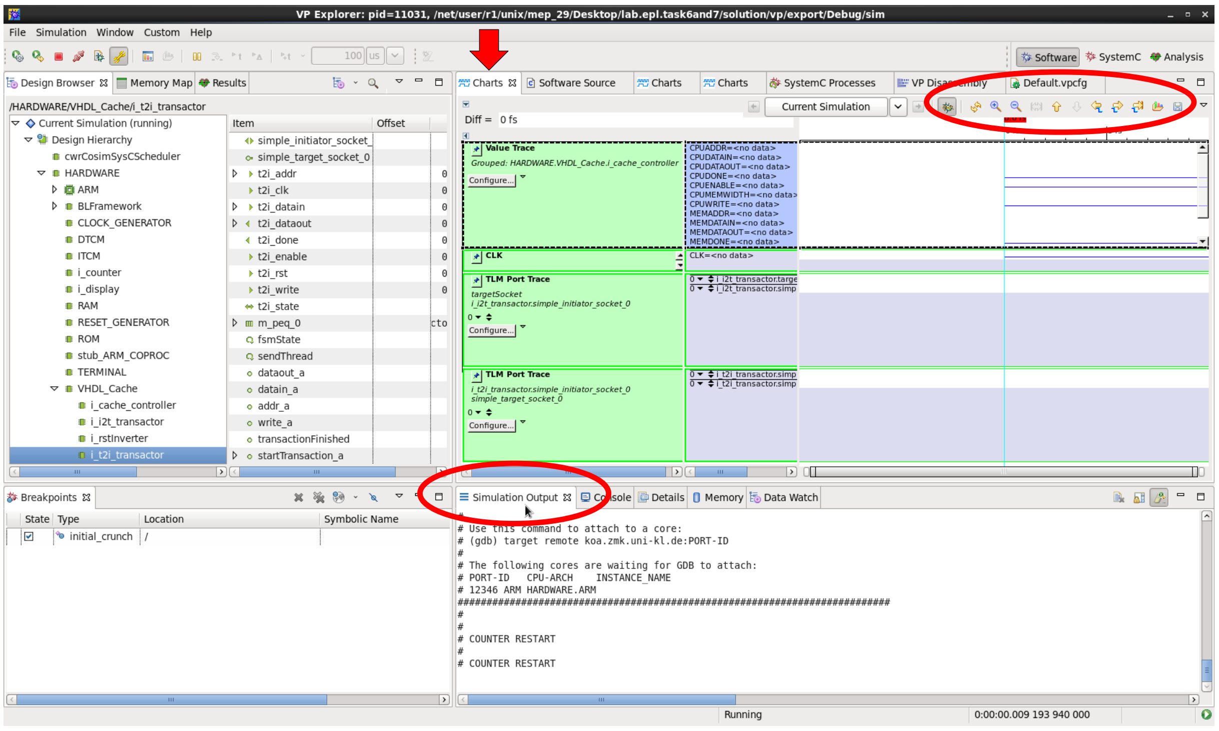
- General → Value Traces
- General → TLM Port Traces



Now you can run the simulation for a certain amount of time:

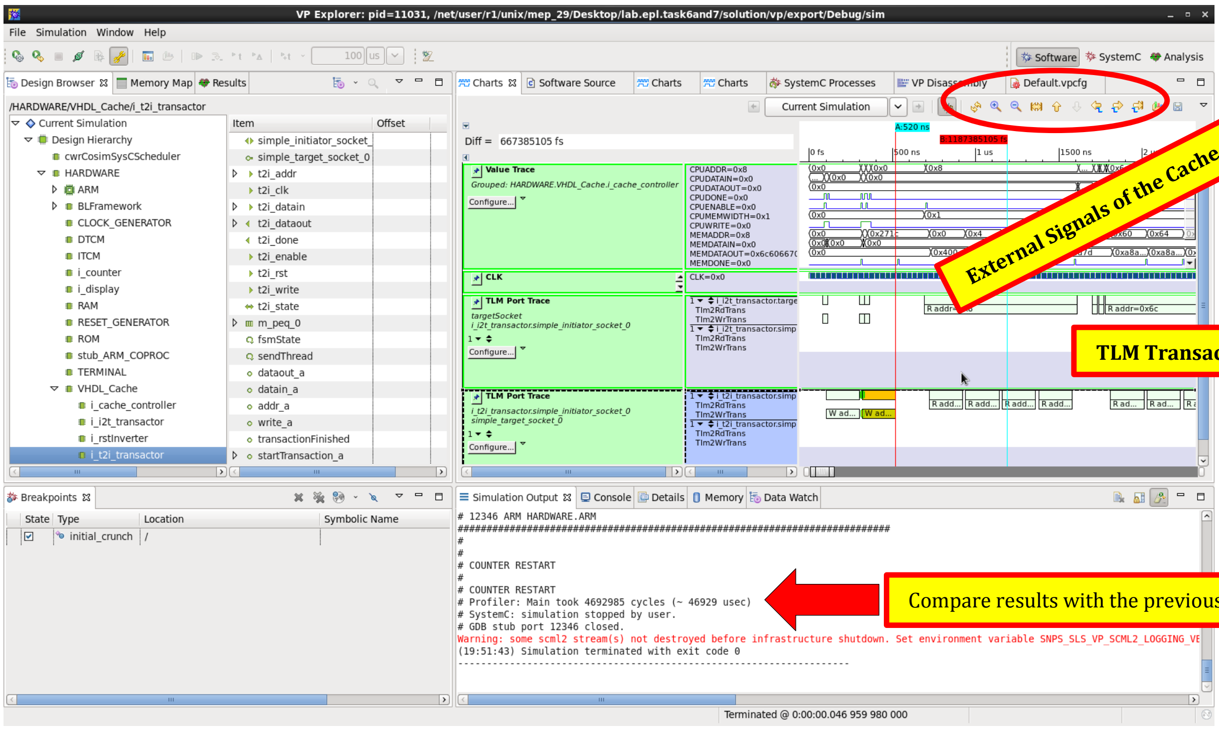

VP Explorer: pid=10660, /net/user/r1/unix/mep_29/Desktop/lab.epl.task6and7/solution/vp/export/Debug/sim
File Simulation Window Custom
100 us
Software SystemC Analysis
Design Browser Memory Map Results
Charts Software Source Charts Charts SystemC Processes VP Disassembly *Default.vpcfg
/HARDWARE/VHDL_Cache/i_t2i_transactor
Current Simulation (suspended at 0:00:0
Current Simulation
Value Trace
CLK
TLM Port Trace
No data to display
Breakpoints
initial_crunch
Simulation Output Console Details Memory Data Watch
0:00:00.000 000 000 000

You can check the Simulation Output and the Chart View.


VP Explorer: pid=11031, /net/user/r1/unix/mep_29/Desktop/lab.epl.task6and7/solution/vp/export/Debug/sim
File Simulation Window Custom Help
Software SystemC Analysis
Design Browser Memory Map Results
Charts Software Source Charts Charts SystemC Processes VP Disassembly Default.vpcfg
Current Simulation (running)
Current Simulation
Value Trace
CLK
TLM Port Trace
Breakpoints
initial_crunch
Simulation Output Console Details Memory Data Watch
Running
0:00:00.009 193 940 000

```
#
# Use this command to attach to a core:
# (gdb) target remote koa.zmk.uni-kl.de:PORT-ID
#
# The following cores are waiting for GDB to attach:
# PORT-ID   CPU-ARCH    INSTANCE_NAME
# 12346 ARM HARDWARE.ARM
##########################################################################
#
#
# COUNTER RESTART
#
# COUNTER RESTART
```

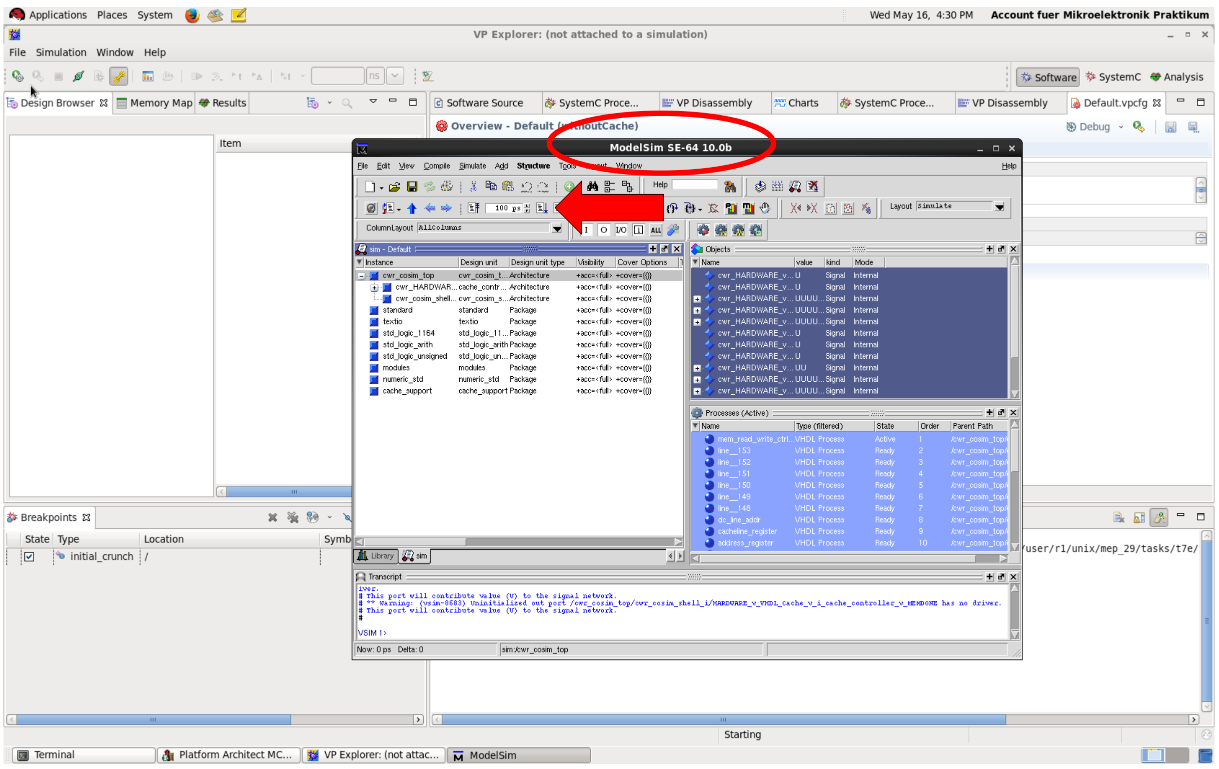
Even before the simulation is finished you can use the refresh and zoom icons to visualize partial results. You can also use right click and middle click of the mouse to position the red and blue timelines shown in the figure below.


External Signals of the Cache
TLM Transactions
Compare results with the previous task

```
# 12346 ARM HARDWARE.ARM
###########################################################################
#
#
# COUNTER RESTART
#
# COUNTER RESTART
# Profiler: Main took 4692985 cycles (~ 46929 usec)
# SystemC: simulation stopped by user.
# GDB stub port 12346 closed.
Warning: some scml2 stream(s) not destroyed before infrastructure shutdown. Set environment variable SNPS_SLS_VP_SCML2_LOGGING_VE
(19:51:43) Simulation terminated with exit code 0
-------------------------------------------------------------------
```

Terminated @ 0:00:00.046 959 980 000

If you want to see the internal signals, you have to start the simulation as an *HDL Interactive Simulation*. Modelsim will be started in foreground and you can use it. **Remember to set time and run on both tools. If one tool seems to be frozen or does not respond try to set more time and run the other tool**.


VP Explorer: (not attached to a simulation)
ModelSim SE-64 10.0b
Overview - Default (withoutCache)

```
iver.
# This port will contribute value (U) to the signal network.
# ** Warning: (vsim-8683) Uninitialized out port /cwr_cosim_top/cwr_cosim_shell_i/HARDWARE_v_VHDL_Cache_v_i_cache_controller_v_MEMDONE has no driver.
# This port will contribute value (U) to the signal network.
#

VSIM 1>
```

Now: 0 ps   Delta: 0   sim:/cwr_cosim_top

After debugging and fixing your implementation **remember to check the output of your median filter** that is now running on a new system (improved with a cache). The output image has to be properly generated as in the previous task, but now faster!

Verify the output file generation time. Make sure it is not an old version of the file previously generated (you can remove the Output.bmp file before simulating to make it sure). Finally, open the file.

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

The expected output is the same as in the previous task.

References

[1] John L. Hennessy and David A. Patterson, *Computer Organization and Design*. San Francisco, California: Morgan Kaufmann Publishers, Inc., 2005.

[2] David R. Kaeli Philip M. Sailer, *The DLX Instruction Set Architecture Handbook*. San Francisco, California: Morgan Kaufmann Publishers, Inc.