Technical University of Munich
Department of Electrical and Computer Engineering
Chair of Integrated Systems

Accompanying Lab to the Lecture

HW/SW Codesign

Prof. Dr. sc.techn. A. Herkersdorf

Manual

Lab Setup

Login to one of the Linux workstations in the lab course room N2135 using your **LRZ account**. When the GUI is up and running, please open a terminal window by selecting the appropriate entry from the Applications menu (top left side of the desktop). Enter the subsequent commands in this terminal window.

When the terminal starts the current working directory of the command shell is your home directory. You can always change to that directory by entering "*cd* " or "*cd ~*". The command "*ls*" displays the contents of the current working directory ("*ls -l*" delivers an extended list, including size and last modification date of the files; directories are identified by the letter d at the beginning of the line). Entering "*cd dir_name*" changes to the directory *dir_name*, "*cd ..*" goes one level up in the directory hierarchy. Alternatively, you can also use the Linux file browser for navigating through the directory hierarchy. It is started by clicking on the "File Manager" entry in the Applications Menu. For editing files with SystemC code you can use an arbitrary text editor installed on the Linux workstations, e.g. *gedit*, *nedit*, *kate, emacs or vi*.

Create a working directory for the lab exercises (e.g. *hwswc_lab*) in your home directory and change to it by issuing the following commands on a terminal

```
cd
mkdir hwswc_lab
cd hwswc_lab
```

Important:
The toolchain and various scripts required to perform HW/SW Codesign lab exercises are loaded using the command

lislab load hwswc

You have to execute this command first, every time you login and on every new terminal/shell/tab, to carry out the lab exercises!!!

Download all the lab exercises necessary to carry out different tasks as part of this HW/SW Codesign course, to the working directory, using the following commands

```
lislab copy hwswc all
```

Hint:

To download (or reload a fresh copy) only particular exercise folders use the command with options as shown

```
lislab copy hwswc "exercise_1a , exercise1b"
```

(The last parameter is a comma separated list of the folders to be copied without quotes.) Existing folders in your working directory with the given names are not overwritten but renamed; the new copies get the desired names.

When you want to finish your session please log off from the workstation by clicking the "Log Out" entry from the Applications Menu. **Please be sure to log out when you leave the workstation!! If you leave a workstation with locked screen your session might be terminated by system administrators** (except short term locks of course)**.**

External Network Access

If you want to login into the LIS workstations from the Internet, you can do this only via *ssh*. All popular Linux distributions contain an ssh client. For Windows you need to install an appropriate client (e.g. PuTTY).

You can access any of the lab workstations prakt01 to prakt18. For establishing the connection please use the hostname “praktXX.lis.ei.tum.de” (01 to 18).

In case you run an X server on your local computer you can also display windows that are opened by the tools (e.g. the gtk wave form viewer). For more details consult the manual of your ssh client.

To log out from the lab workstation enter *exit* on the command line, which will log you off the LIS workstation.

Organizational Matters

Deadline and modalities for the delivery will be announced on the course web page in the TUM Moodle eLearning portal (https://www.moodle.tum.de)

If you have problems using either hardware or software on the workstation, please contact either the tutor or send a mail to thomas.wild@tum.de.

If you find errors or inconsistencies in this manual, or if information is lacking that is needed for an exercise, please do not hesitate to inform me by sending a mail to thomas.wild@tum.de. This would enable us to both inform your colleagues and correct the manual for the subsequent semesters.

Many thanks in advance!

Part 1: System Simulation

Experiment 1. Modeling with SystemC

This and the following experiment make up the first unit of the lab, which addresses system level modeling. The main learning target of this unit is to convey a first impression of system level modeling with SystemC, especially on transaction level, and to use SystemC for the exploration of a System-on-Chip (SoC) architecture.

Experiment 1 covers the basic principles of SystemC, which is the widely used description language that allows modeling and simulation at system level. This experiment is used to give an impression how SystemC supports concurrency, an inherent property of all pieces of hardware, and how SystemC models are structured. The concrete exercise consists of modeling the interaction of a simple system's function blocks first with signals and then on transaction level.

Please use the handout of the introductory presentation as input for carrying out the exercises. Further information that is required for the tasks is given in the subsequent description.

The working directory (*hwswc_lab*) in your home now contains folders named *exercise_1a* and *exercise_1b*, which encompass all given files for the first lab experiment. These files contain either the complete SystemC code for certain components of the model or part of the code, which has to be completed by you before you can generate the simulation program. Further on, Makefiles are given, which automate compiling and linking the simulation program without the necessity to call compiler and linker manually. The concrete commands that have to be issued are given explicitly in the following description.

As shown in the introductory presentation, the program generated by the compiler encompasses the system model as well as the simulation kernel. The simulation, i.e. the execution of the program, is simply started by entering the program name on the command line of the terminal window, optionally followed by additional command line parameters. During the simulation run, the simulator outputs messages in the terminal window, which characterize the simulation flow. In addition, files containing waveforms of certain signals of the system may be recorded as well (file suffix *.vcd*). You can display these waveforms with the waveform viewer gtkwave by issuing the command "*gtkwave wave_file_name.vcd*".

Application example

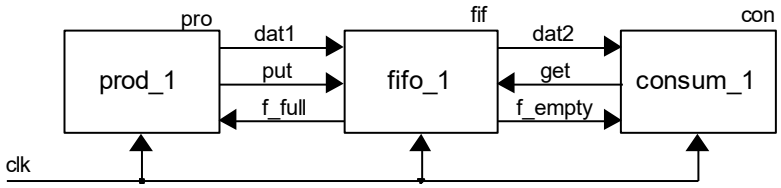
For demonstrating the basic properties of SystemC we will have a look at a very simple application example. It consists of a model with a producer and a consumer communicating via a FIFO. The following figure shows the configuration and the flow of data.

In the first part of the exercise, the interaction between the different function blocks is modeled via signals, like they are known from hardware description languages. In the second part, the system is modeled on a higher abstraction level, i.e. the blocks communicate by means of transactions.

Basic Properties of SystemC

For carrying out the first part of this experiment, please change into the directory *exercise_1a*. The first model is synchronous to the clock *clk*, i.e. all actions are carried out with rising clock edge. In the following, all data signals are assumed to be of type *int*, all control signals to be of type *bool*.

The file *main_1.cpp* contains the top level of the model depicted below. The main parts are the instantiation and connection of the modules, the control of the simulator and the specification of the signals that have to be recorded for later inspection with the waveform viewer.



The producer *prod_1*, modeled in the files *prod_1.h* and *prod_1.cpp*, is given. This module contains two SC_THREAD processes, which communicate via the signal *send*. At particular points in time, the process *send_trigger* sets *send* to high and thus initiates the process *produce* to execute a write action with the next rising edge of the clock. If *send* is high and the FIFO is not full the process *produce* outputs a certain value on the output port *dat1* and sets *put* to high. The FIFO takes the value with the subsequent rising clock edge and stores it internally. If the FIFO becomes full it sets its output port *full* (connected to signal *f_full*) to high.

When two values v1 and v2 have to be written in subsequent clock cycles and the FIFO has only one free position, the following situation will occur: The producer initiates the write attempt for v2 with the same rising clock edge as the FIFO writes v1 to its last free position. Therefore, there will be no free space in the FIFO for v2 and the write request for v2 will be ignored by the FIFO. This situation can be detected in the producer by considering the status of the signals *f_full* and *put.* If at a rising clock edge *put* is set (indicating a write attempt in the previous clock cycle) and at the same time the *f_full* is set as well, the previous write was not successful. In this case, the value that should have been written must be restored (in our example where subsequently incremented values are written to the FIFO, a decrement is necessary) and a new write attempt is initiated next time when a write trigger occurs, and the FIFO is not full.

The described interaction between producer and FIFO allows writing data in each clock cycle and avoids loss of data due to FIFO overflows.

(Another solution could be to provide a further control signal to inform the producer when only one entry is free. This would easily allow avoiding subsequent write attempts with only one free FIFO location.)

The other modules used for simulating the setup shown before are not yet complete. The SystemC code for both FIFO and consumer will have to be complemented by you to get the simulation working. (The sections are marked with appropriate comments.)

FIFO: The FIFO model is quite simple and contains two processes, one for reading (*do_read*) and one for writing (*do_write*) data. The FIFO content is stored in an internal member variable (integer array). Further member variables are used to hold the addresses for the next write and

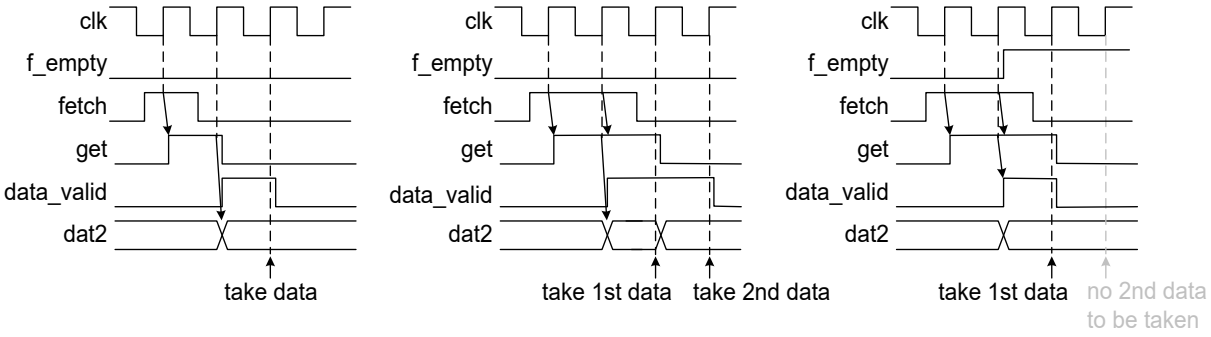
read action as well as the current fill level. The model of the FIFO is complete with the exception of the missing registration of the two processes in constructor (file *fifo_1.h*). Considering the implementation of the processes in *fifo_1.cpp*, first determine the type of the processes and then complete their registration and the specification of the sensitivity in the constructor.

Consumer: The module declaration as given in the header file *consum_1.h* does not yet contain the port declarations. You have to add these declarations using the same names as referenced in the instantiation of the consumer in *main_1.cpp*. (Otherwise you will have to change the names there).

Moreover, the implementation of the process *consume* is missing in the file *consum_1.cpp*. The consumer should also be able to read data continuously from the FIFO (as it is possible in the producer to write data). As can be seen in the header file, two thread processes should be used that interact with each other using the signal *fetch*. The process *fetch_trigger* initiates a read action in the process *consume* by setting the signal *fetch* to high.

With a rising edge of the clock the process *consume* first sets *get* to high if the FIFO is not empty. The read is recognized by the FIFO with the subsequent rising clock edge, and in turn it outputs the value fetched from its internal memory on *dat2*. With the next positive clock edge, the value is read by the consumer via its input port. This means, a read needs 2 clock periods to complete (see the following timing diagram on the left).

In order to identify the clock edge when to take the data value *dat2* a further signal *data_valid* is introduced, which is basically the signal *get*, delayed by one clock period if the FIFO does not become empty. The diagram in the middle shows this situation for two subsequent reads.



However, if the FIFO gets empty after the first data read operation a subsequent read command cannot be finished successfully. The signal *data_valid* has to be set to low in order to avoid taking the previous data again, as shown on the right side. The mechanism described above allows continuous reading from the FIFO.

When completing the consume process you will have to generate the signals *get* and *data_valid* appropriately and also store the read data in the member variable *consumed_data*. In addition to the pure functionality, please generate also a message in the consumer indicating the simulation time and the read data value when a read operation action has been carried out.

(Note that, the *main_1.cpp* file contains also commands that enable to generate waveforms for several signals. You should not have to change these lines of code as long as you use the same port names for the consumer.)

After complementing the SystemC code compile your model by entering the following commands in the following order

make depend
make

on the command line of the terminal window. This starts the compilation of the files containing the source code and – if successful – link the compiled object code with the SystemC library to the executable program *sim_1.x*. Verify that this program has actually been generated in the directory of the current exercise.

Then, run the simulation by entering the command

sim_1.x

(if you want to execute the simulation for a different time period than the default value of 5000 ns, start the simulation by ***sim_1.x sim_dur***) and observe the output in the terminal window.

Further on, verify the functionality of the model by inspecting the traces of the signals that connect the sub-modules of the system.
The waveform viewer is started with the command

gtkwave traces_1.vcd

After the wave file has been loaded into gtkwave you have to select the signals that should be displayed. After startup of gtkwave, the sub-window "SST" is shown in the upper left part of the gtkwave main window. Click on "SystemC" and you should get the list of recorded signals in the "Signal" sub-window below "SST". Select the desired signals (pressing CTL+A selects all signals) and click on the button "Insert". Now, the waveforms of the selected signal should be displayed in the main window of gtkwave.

You can change the time interval that is shown by clicking on the zoom buttons. By selecting a signal name left to the associated waveform and pressing the right mouse button you can change the radix of the displayed signal (default hexadecimal) via the menu entry "Data Format".

Modeling on Transaction Level

Modeling like in the first part of this experiment is not suited for effectively exploring alternative system architectures because many details of the implementation, especially the communication protocols, e.g. the handshaking between modules, would have to be described in detail. These details, however, are not yet known in the exploration phase, which is done early in the design flow of an SoC. In fact, they become available only after several refinement steps of the chosen architecture. Further on, models that communicate by means of signals are very complex, cannot easily be modified and consume much simulation performance.

Therefore, the components of system models interact via so called transactions, which correspond to the execution of a function. Via a specific interface (socket) the initiator of a transaction calls the function, which is implemented in the target of the transaction. This approach allows to leave it open how the communication is actually realized in the final implementation, yet to capture the interaction, however, on an abstract level with low simulation overhead.

In the following we adhere to the TLM 2.0 standard, which is based on SystemC and mainly targets at the communication via memory mapped buses, however, can also be applied in a more general sense. A major constituent of TLM 2.0 is the generic payload. It is a class, which specifies for a transaction the required action to be performed (read or write), the data to be transferred and – depending on the environment – the address or other information (e.g. validity of the data via byte steering). Payload objects are exchanged when calling a transaction. As the interaction and the payload data format are standardized, interoperability of modules obeying the TLM 2.0 standard is granted.

TLM 2.0 defines many variants of interaction via transactions, which differ in respect to timing (loosely or approximately timed), the number of phases to capture different types of blocking and non-blocking interaction (begin and completion of both request and response). Details of the TLM 2.0 standard are quite complex and go beyond the scope of this lab. Therefore, we use a very simple communication mechanism to generate and simulate a TLM model for the FIFO system. We use the blocking transfer for loosely timed modeling with only one phase. This is based on the simple sockets from the TLM utils, an add-on to the TLM 2.0 standard, and uses the *b_transport* function as described on the TLM slides of the SystemC introduction. The following figure shows the configuration to be simulated.

For the second part of this experiment, please change into the directory *exercise_1b*, where most parts of the SystemC model are already given. However, consumer (*consum_2.h* and *consum_2.cpp*) and the top level (*main_2.cpp*) some lines of SystemC code – identified with "// fill in" – are missing, which have to be inserted by you. To do so, have a look at the corresponding slides in the introductory presentation and also inside the code given for the producer and the FIFO which should help to complete the model.

Please note that the scenario used in part 2 is a bit different from that in part 1 of this exercise. Both the producer and the consumer write/read not only one word, but an arbitrary amount of data in the range between 1 and 16 Bytes into the FIFO, which can store 40 Bytes. Further on, writing and reading of data in the producer and consumer is done with two processes, however, they interact with events (*sc_event*), instead of signals.

In the producer as well as in the consumer a *simple_initiator_socket* has to be included. The FIFO will have two instances of *simple_target_socket*, which will then be bound to the initiator sockets of producer and consumer, respectively (see figure above). The function *b_transport* will then be called from within the producer to write data into the FFO, and from within the consumer to read data from the FIFO. For this purpose, an appropriately generic payload has to be set up and given together with an *sc_time* variable as parameters to *b_transport*.

The FIFO provides the implementation of the function *b_transport*, which is registered with both instances of *simple_target_socket* in the constructor of the FIFO.

In the implementation of *b_transport* the operations read and write are differentiated and the read and write pointers as well as the array that stores the data are manipulated as required. Then the response status as part of the generic payload is set depending on the success of the operation. Further on, the delay value is set according to the number of clock cycles that would have been needed to transfer the data over a 32-bit wide bus with a clock period of 50ns. The

delay value set by the target is then used as parameter for the wait statement in the initiator to capture the correct timing behavior. (This flow corresponds to the sequence shown on slide 18 of the introductory presentation.)

After completing the SystemC model you can generate the simulation model by entering the same set of commands

make depend
make

If everything went fine the executable program *sim_2.x* should be available in the directory *exercise_1b.*

To run the simulation of the TLM model enter

sim_2.x

(if you want to execute the simulation for a different time period than the default value of 5000 ns, start the simulation by ***sim_2.x sim_dur***).

All output of this simulation is sent to the terminal window (waveforms are not generated). For each read or write operation consumer or producer and the FIFO output information on the terminal window concerning the request and also the respective status of the FIFO.

Verify the correct behavior of the communication between the modules. (Please note that in this version of the producer, data that has not successfully been written will be lost, i.e. no further write attempt for this data will be issued.

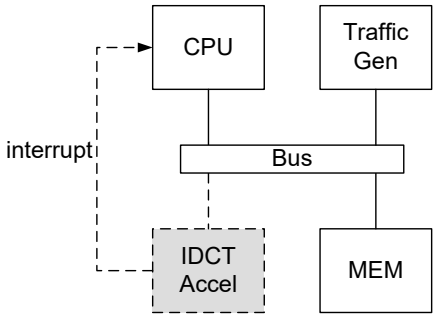
When comparing the different models from part 1 and 2 of this exercise you should note that a read or write request in the TLM model generates only one event (per transaction, independent of the transferred amount of data), whereas in the signal level model events are generated for each rising clock cycle in every edge sensitive process. Therefore, the simulation effort in a TLM model is significantly lower.

This concludes the introduction of SystemC, which should have given you a first insight into some aspects of modeling in SystemC. In the next experiment you will apply SystemC TLM modeling for the performance exploration of different SoC architectures.

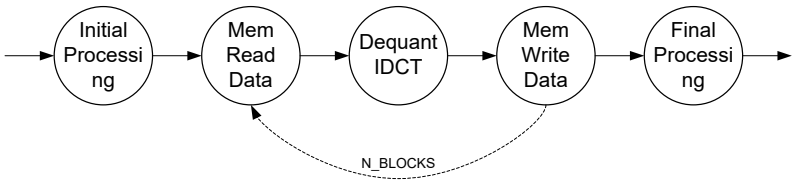
Experiment 2. Architecture Exploration with SystemC

In this experiment, three different alternative processor architectures have to be explored using SystemC TLM models as introduced in experiment 1.

The following figure shows the configuration consisting of a CPU, a memory block and optionally an accelerator module, which are all interconnected via a system bus. In one of the alternative configurations the accelerator has an interrupt to the CPU. In all variants, another master (traffic generator) is attached to the bus that generates background traffic on the bus by randomly reading/writing data from/to the memory.



The application that should be investigated on this architecture is a fraction of the jpeg application to display pictures, which will be realized on an FPGA prototyping board in the final part of the lab. The function to be considered here comes from the part of the application that decodes the image data, covering the de-quantization and the inverse discrete cosine transformation (IDCT), including the read and write operations from/to the memory. It is assumed that the picture data has to be fetched from memory before doing de-quantization and IDCT and that the decoded data have to be written back to memory. This application scenario is very simplified and makes up only a fraction of the overall jpeg application, however, it is sufficient for the investigations in this experiment. The following task graph can represent the functionality.

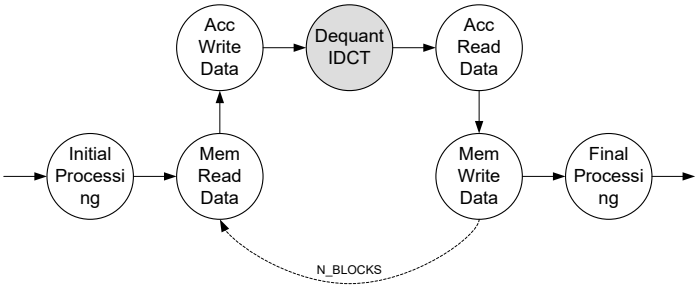


After some initial processing the image data have to be fetched from memory, then de-quantized and IDCT transformed. After that the decoded picture data has to be stored in the memory followed by some final processing. The picture to be decoded is subdivided into pixel blocks that iteratively have to be read, processed and stored until all blocks of the picture are decoded. This is indicated by the dashed loop arrow in the task diagram. (In our example the number of blocks to be processed is – for simplicity reasons only – 16. A block of encoded image data is 128 bytes, a block of decoded data is 64 bytes.)

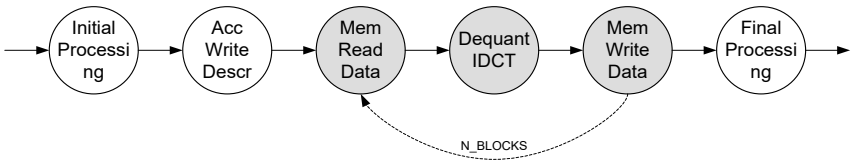
The architecture alternatives to be explored differ in the mapping of the de-quantization and IDCT operations:

A1 IDCT mapped onto the CPU
A2 IDCT mapped onto HW-Accelerator without DMA capability
A3 IDCT mapped onto HW-Accelerator with DMA capability

In case A1 the CPU has to perform exactly the sequence of tasks as depicted in the graph above. In case A2 the de-quantization and IDCT functions (marked grey in the below task graph) are mapped onto the accelerator. However, the CPU also performs the required memory transfers. I.e. in addition to the memory transfers it also has to write and read the data to and from the accelerator. It is assumed that the accelerator starts directly after receiving the encoded data. The CPU will directly try to read the decoded data back, which will, however, be not available until the accelerator has finished processing. This read from the accelerator will therefore stall until processing is done in the accelerator.



In the variant A3, it is assumed that the accelerator has DMA capabilities. I.e. it can access the bus itself and can read and write data from/to the memory without CPU intervention, which also avoids transferring the same data twice over the bus. Thus, the accelerator would not only perform the de-quantization and IDCT but also offloading the CPU from the memory operations (marked grey in the task graph below). The prerequisite, however, is that the CPU writes a DMA descriptor to the accelerator containing the amount of data to be read and written and the associated addresses.



In this exercise the different scenarios A1,A2 and A3 have to be explored for different CPU clock frequencies. For this purpose, first the appropriate TLM model has to be completed and a spreadsheet containing latencies and load values of the bus and the involved processing resources has to be filled in. The spreadsheet makes up the deliverable of this exercise.

Completing the Transaction Level Models for the Architecture Exploration

The first task is to complement the given SystemC TLM model that captures the scenarios described before. The directory *exercise_2* already contains major parts of the model: The models of CPU, bus, memory, and traffic generator as well as the top level are already complete and can be used as given. The given SystemC files of the accelerator contain the code that is modeling the accelerator without DMA capability. Your task will be to write the code for this feature. Before more details are given for this task, the simulation model and the underlying concepts is briefly described.

To enable the adaptation of the simulation model without the need to always modify and recompile the SystemC code, the simulation model is parameterized. In this case the parameters are the clock periods of the CPU, of the bus including the memory and of the accelerator and the availability of the DMA feature in the accelerator.

These parameters can be modified via command line switches to simulate the architecture configuration as desired. The simulation program *sim_3.x*, which will finally be generated, has the following command line parameters:

sim_3.x *[-c <cpu_clock_period>] [-b <bus_clock_period>] [-a <acc_clock_period>] [-D] [-v]*

(The clock period values must be entered as integers designating SC_NS.)

All parameters are optional. If no values are given, a configuration with 5ns CPU clock period, 10 ns bus clock period and without an accelerator is simulated. If you also enter a value for the accelerator clock period, then the simulation model will include an accelerator, which is run with the given clock period. If the switch ***-D*** is used in addition to ***-a***, the accelerator will be run in DMA mode.

In case the option ***-v*** is used, the simulator will output additional log information. Entering ***sim_3.x -h*** will result in an overview of the command line switches.

Please note that this simulation model is intended for performance simulation only and not for functional verification. Therefore, the model does not contain program code representing the actual functionality capturing de-quantization and IDCT. These functionalities are abstracted by the delay in terms of clock cycles that would be needed if these functions would be executed on the CPU or on the accelerator. Further on, no actual picture data are exchanged between the architecture modules, only the duration of the corresponding transfers are captured in the simulation model. Despite these simplifications, the model can realistically simulate the interplay between the involved resources including the resolution of competing accesses on shared resources. Therefore, it provides good insights into the behavior of the architecture and the usefulness of the chosen configuration and the associated parameters.

In the following the major parts of the model are introduced. To understand the simulation model, you should also have a look at the corresponding source code. However, it is not required to fully understand all details of the model. The description should help you in getting a clear view what is going on in the model and how it is implemented. Test sections in square brackets [] are not essential for doing the exercise.

(It is not claimed that the given realization is optimal. Nevertheless, proposals for improvements are welcome. Please let me also know if you find an error in the code…)

Top level (*main.cpp*): The command line parameters are parsed using the ArgvParser and the appropriate internal variables are set. (ArgvParser is available from the internet under GPL license. Therefore the files *argvparser.** are needed in the directory for exercise 2.) Then, the different modules are instantiated. To allow for a modification of both the model structure and the behavior of the different modules, the constructors of the modules have additional parameters, which are set when generating the instances.

(In order to allow such parameters, the constructors of most modules do not apply the *SC_CTOR* macro, but instead use usual C++ syntax and include the additional statement *SC_HAS_PROCESS()*).

The final section of *main.cpp* contains the interconnection of the modules. When the simulation is over (after the *sc_start()* command) the total simulation time (i.e. the processing latency) and the load of the relevant resources are output (call of their *output.load()* methods).

Bus (*bus.h, bus.cpp*): The bus interconnects the two masters (CPU and traffic generator) with the two slaves (memory and accelerator). If its DMA feature is active, the accelerator works also as a bus master and thus can access the memory via its master interface. A bus master is always connected to a target socket of the bus, a slave to the initiator socket. The bus model is written in a way that it allows specifying the number of these socket types dynamically at instantiation time. Therefore, it does not use socket objects of both variants but two pointers of the respective types. The required number of sockets will dynamically be generated in the constructor of the bus (see *nr_of_masters* and *nr_of_slaves*).

The addressing scheme is simplified: Instead of real addresses only two different values (0 and 1) are used in the generic TLM payload to determine which slave has to be accessed by the master. The numbering scheme of masters and slaves is as follows:

Master	Module	Slave	Module
0	CPU	0	Mem
1	TGen	1	Accel
2	Accel		

The bus models the following pipelined arbitration scheme:

- If the bus is free then a master requesting a transfer (by calling the *b_transport()* function of the associated bus target socket) will directly get access to the bus. I.e. the bus will call the *b_transport()* function of the appropriate slave's target socket. The bus will go into busy state, which will last for a time determined by the speed of the bus and the amount of data to be transferred.
- If the bus is busy, a secondary request can be arbitrated but it is delayed until the ongoing bus transfer (primary request) has finished. I.e. the *b_transport()* call of a master corresponding to a secondary access will be suspended using *wait(...)* until the event indicating the end of the primary request is notified. Then the secondary request becomes the primary one and gets access to the bus.
- If a secondary request has been arbitrated and a further bus request arrives, the corresponding call of the *b_transport()* function will result in an error as TLM response status. The master will then wait for a specific backoff time and try again until it is a secondary or primary access.

Consequently, a primary bus request will last for the transfer time to/from the slave. A bus request that has been arbitrated as secondary request will additionally have a duration depending on how long the primary request needed to finish. All bus requests – successful or unsuccessful – will consume a further delay representing arbitration latency.

[When looking at the code, you may notice that a different type of simple sockets is used in the bus, namely tagged simple sockets. The advantage is that these sockets provide *b_transport()* functions including the parameter *SocketID*, which allows identifying the master that actually called *b_transport()*, i.e. that wants to access the bus. This feature is not a functional

requirement; however, it is used for logging purposes to better understand which master is trying to access which slave.]

CPU (*processor.h, processor.cpp*): Depending on the presence of the accelerator in the system, the processor functionality is covered by different processes: Therefore, in the constructor of the processor, either the function *software_code()* or *hardware_code()* is registered with the simulation kernel as *SC_THREAD* process. If the latter process is use, it is made sensitive to the positive edge of the interrupt port of the processor. This is required to inform the processor about completion of the processing in the accelerator. Otherwise, the processor will not go on with the final processing.

Please have a look at the code. You should be able to identify the sections of the code that correspond to the different tasks shown in the above graphs for the different architecture variants. The durations of the different processing tasks are designated in terms of cycles of the CPU clock as specified in the header file *global_definitions.h*. When varying the CPU clock, the processing delays will therefore scale correspondingly. (The numbers of required clock cycles are e.g. the result of investigating the real code with a cycle accurate instruction set simulator.)

Please also have a look on the calculation of the time consumed in the processor for doing the transfers. (You will need this for your own code in the accelerator.) The access of a slave via the bus is always realized with a *do…while* loop, which is executed until the response status of the transaction results in *TLM_OK_RESPONSE*. The duration of this loop encompasses one or more arbitration times (depending on the number of trials), the time needed to transfer the amount of data to be read/written and possibly the time spent as a secondary request. The first two contributions could easily be determined (number of loop iterations times backoff time and the transfer delay related to the data). The last however, is a dynamic value depending on the bus load situation. Therefore, time consumption for doing the transfer is simply calculated by accounting for the time span between the start of the transfer (stored in *transfer_start_time*) and the current time when the transfer is successfully finished.

[Note that accounting the time consumed in a master for bus transfers – needed to finally calculate the corresponding load value – differs from capturing simulation latencies using *wait()* statements. Normally, in our model the delays of transactions are calculated in the slave (target), the corresponding *wait()*, however, is executed in the master (initiator, see slide 18 in the SystemC introduction). However, to capture the delaying nature of secondary bus accesses, the bus has to implement the associated delay and the corresponding call of b_transport() cannot terminate. This is the reason for accounting transaction latencies as described before.]

(Note that in the SystemC model the quantization parameters (needed for de-quantization) and their transfer to either CPU or accelerator are disregarded. A further assumption is that the encoded data are already available in the memory when needed.)

[**Traffic Generator (*traffic_generator.h, traffic_generator.cpp*)**: This module is used to generate some background traffic on the bus. The process *gen_traffic()* produces random read/write memory accesses of up to 256 bytes. (The behavior can be modified by parameters defined in *global_definitions.h*.)]

Accelerator (*acclearator.h, accelerator.cpp*): In accelerator.h, which is already complete, you can see that in addition to the *b_transport()* function (accessed as usual via the simple_target_socket), there is also an *SC_THREAD* process *idct_with_dma()*. This process is

used to implement the part of the functionality that covers the DMA feature. The implementation of this function is your task.

When writing the code please consider the following hints:

- The start of this process is triggered by the *b_transport()* function when data (i.e. the DMA descriptor) is written to the accelerator. For this purpose, the event *idct_with_dma_start* is declared as a member of the accelerator.
- For the time the accelerator needs to read, to process and to write data, it should not accept any request. For this purpose, a boolean member variable *accel_busy* is declared.
- The accelerator should – similarly to the processor – do the necessary memory accesses and it should also realize the processing delay with an appropriate wait statement. The number of clock cycles to represent the processing latency in the accelerator is defined in *ACCEL_PROCESSING*, the duration of a clock cycle in *ACCELERATOR_CLK_PERIOD* (see *global_definitions.h*),
- When the accelerator is done, it should notice the processor by issuing an interrupt. (Simply generate a short pulse on the *interrupt* output.)

You should also have a look at the code contained in *processor.cpp*. These pointers should help you to write the correct code for *idct_with_dma()*.

If you have completed the code in *accelerator.cpp* please run the following commands

make depend
make

to compile the simulation model. If everything went fine you should find the executable program ***sim_3.x*** in the directory *exercise_2*.

[As already mentioned before, there is a file ***global_definitions.h***, which contains the definition of several constants needed in different parts of the model. You may have a look at the file and play with the parameters if you like. (Recompilation of the model is needed if you make changes in *global_definitions.h*. Simply type ***make*** after storing the file and the recompilation will automatically be done for you.) However, be sure, that you do the simulations to be delivered in the spreadsheet with the original parameter values.]

Architecture Exploration

The second part of this experiments is the exploration of the architecture scenarios A1,A2 and A3 as described above. You should have completed the simulation model and have a working ***sim_3.x*** in your directory *exercise_2* before going on.

Spreadsheet – In addition to the SystemC code, your folder *exercise_2* also contains the file *HWSWC_ex_2.ods*, which is an OpenDocument spreadsheet prepared for collecting the results of your simulations. Open this file with the LibreOffice Calc by entering

soffice HWSWC_ex_2.ods

on the command line. This should load the spreadsheet.

Execute the simulations for the different scenarios for the CPU clock periods as given in the spreadsheet and enter the results output by the simulator into the marked fields. (The latency value is the simulation duration. Lower latency means better performance.)

The diagram on the right will show how the performance of the different architectures A1, A2 and A3 evolves with increasing CPU frequency (i.e. decreasing clock period).

Looking at the diagram, you should realize that mapping processing intensive tasks onto an accelerator is not in all cases the best solution. Transfer times via the system bus to make use of the accelerator (both are shared resources) must be considered as well.

Exploration Script – To run a predefined architecture exploration run *deliverable_1.sh* in folder *exercise_2* by typing

./deliverable_1.sh

This will generate a report file *hwswc_YYYY_deliverable_1_YOURNAME_LRZID.rpt* which is part of deliverable 1.

The completed spreadsheet *HWSWC_ex_2.ods* and the report file *hwswhc_YYYY_deliverable_1_YOURNAME_LRZID.rpt* are deliverable 1 of the HW/SW Codesign Lab. Please do not forget to also enter your name and matriculation number in the appropriate fields of the spreadsheet before you submit your file.

Part 2: VHDL Modeling

Experiment 3. Modeling with VHDL

In this lab experiment you get to know the basic concepts of the hardware description language VHDL. A Finite State Machine (FSM) is used as an example for simulation and synthesis of VHDL models. This experiment and the associated introduction cover only a part of the language and its use for hardware design. If you have further interest in this topic you should participate in a special VHDL lab.

For carrying out the exercises of this experiment you only need the language constructs described in the introductory presentation. The usage of Xilinx Vivado and its integrated VHDL simulator is described in the subsequent sections below.

The tasks of this exercise encompass modeling the FSM in VHDL, its simulation as well as its synthesis into a netlist of logic gates for an FPGA implementation.

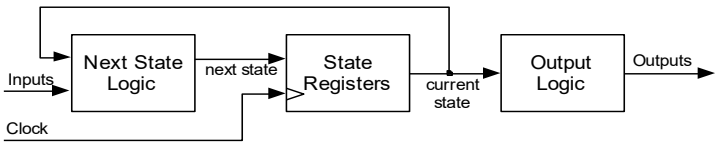
Finite State Machines

Finite State Machines play a central role in the design of digital hardware, especially for the control part of synchronous circuits. FSMs are used to control the interplay of the associated sub-modules. They have to determine, depending on input signals and the current state of the control unit, the required output signals as well as the next state of the controller.

The FSM shown in the following figure is a Moore machine, which is characterized by the property that the outputs are solely determined by the current state of the machine, but not by the input values. It can be split up into the following sub-blocks:

- The next-state logic determines – depending on the current state of the FSM and the current input values – the next state of the FSM.
- The state registers contain the value of the current state. With each rising edge of the global clock they store the next state value provided by the next state logic.
- Based on the current state, the output logic generates the output values.

(Mealy machines, whose output values are influenced by both current state and the inputs, are not considered here.)



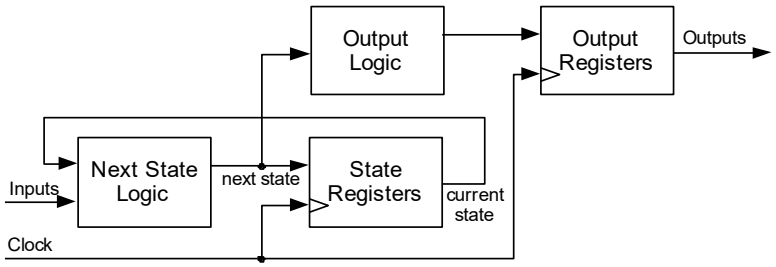
Structure of an FSM

There are many different approaches for modeling FSMs with VHDL, which especially differ concerning the number of processes that are used. The structure shown above is most simply captured with two processes. One synchronous process models the state registers; the other, unclocked process describes the two other blocks, which contain pure Boolean logic. (The FSM model shown in the introductory presentation adheres to this modeling style.)

The structure shown above, however, has a big disadvantage. In synchronous circuits the minimum clock period (i.e. the maximum frequency) is determined by the maximum propagation delay through the logic between subsequent register stages (in addition to clock-to-output latency and setup time of the registers, see slide 12 of the VHDL introduction). The higher the logic delay, the lower the possible clock frequency. Taking this into account, severe problems concerning the allowed maximum frequency could arise when connecting two HW modules with combinatorial logic at both inputs and outputs. Even if both modules individually fulfill the requirement of a particular maximum clock frequency, this frequency cannot be guaranteed any more when connecting them. The resulting circuitry may have a lower maximum clock speed, because connecting the output logic after the last register stage of the first module with the input logic before the first register stage of the second module may lead to a longer overall delay between registers than in each individual module.

This effect is particularly critical as the problem is not noticed when designing the individual modules but later in the integration phase. An approach to avoid this effect is to adhere to a modeling style that generates only registered outputs, i.e. there is no combinatorial logic at all after the last register stage. Connecting modules that are all designed according to this rule would thus avoid the problem. (Alternatively, registers inputs could be used.)

This concept should now be applied to modeling an FSM. The following figure shows a transformed version of the above FSM, which conforms to this modeling guideline while keeping the same functionality. (In order to avoid an additional latency of one clock cycle at the outputs, the output logic is now connected to the next state logic.) The maximum frequency of this circuit is determined by the maximum latency through next state and output logic (instead of the maximum latency through either one of them as in the first FSM version).



FSM with registered outputs

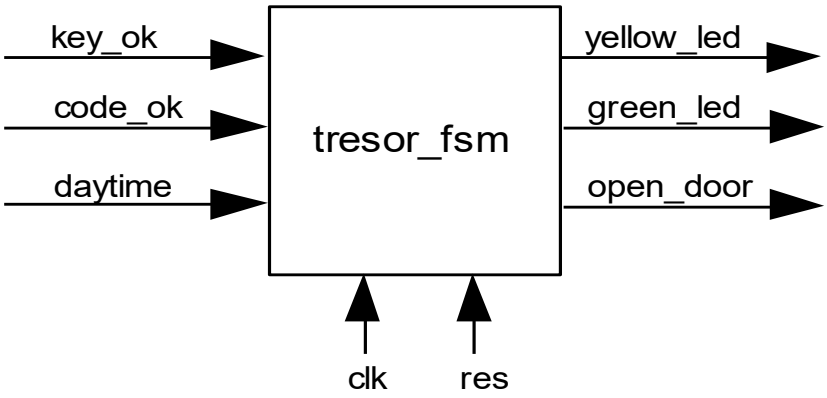
For modeling this version in VHDL, three processes are recommended: Two for describing the combinatorial logic blocks and a common process for state and output registers.

Modeling of an example FSM

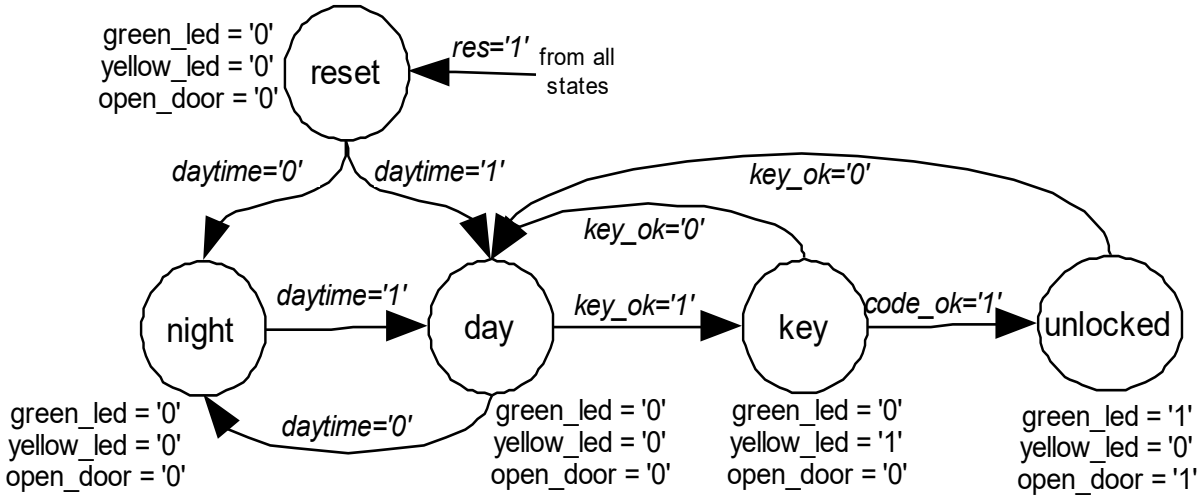
In this exercise you should model *tresor_fsm,* which implements a simple control mechanism for a door of a safe and synthesize the VHDL code for an FPGA implementation. The FSM should have two modes, for day and night, and allow opening the door by first using a key followed by entering a valid code. In day mode, the door should open after first inserting the correct key into the lock and afterwards entering the required code into a numeric pad. Unlocking the door during the night should be prohibited.

It is assumed that opening the lock with the correct key is signaled by the signal *key_ok*, entering a valid code is notified by the signal *code_ok*. The signal *daytime* determines whether the FSM is currently in night (‘0’) or in day mode (‘1’). The signal *res* should bring the FSM

synchronously to the reset state. All these signals are inputs for the *tresor_fsm* and are generated externally. The FSM should produce the following output signals: *open_door*, which controls the opening mechanism of the door, as well as *yellow_led* and *green_led*, which are used to show the status via LEDs.



The following figure shows the state diagram of the FSM. The circles denominate the different states, the arrows the possible transitions between the states. Arrows are annotated with the condition for the transition, states with the corresponding output values.



FSM State Diagram

Please write the VHDL model of this FSM by using the version with registered outputs as described before.

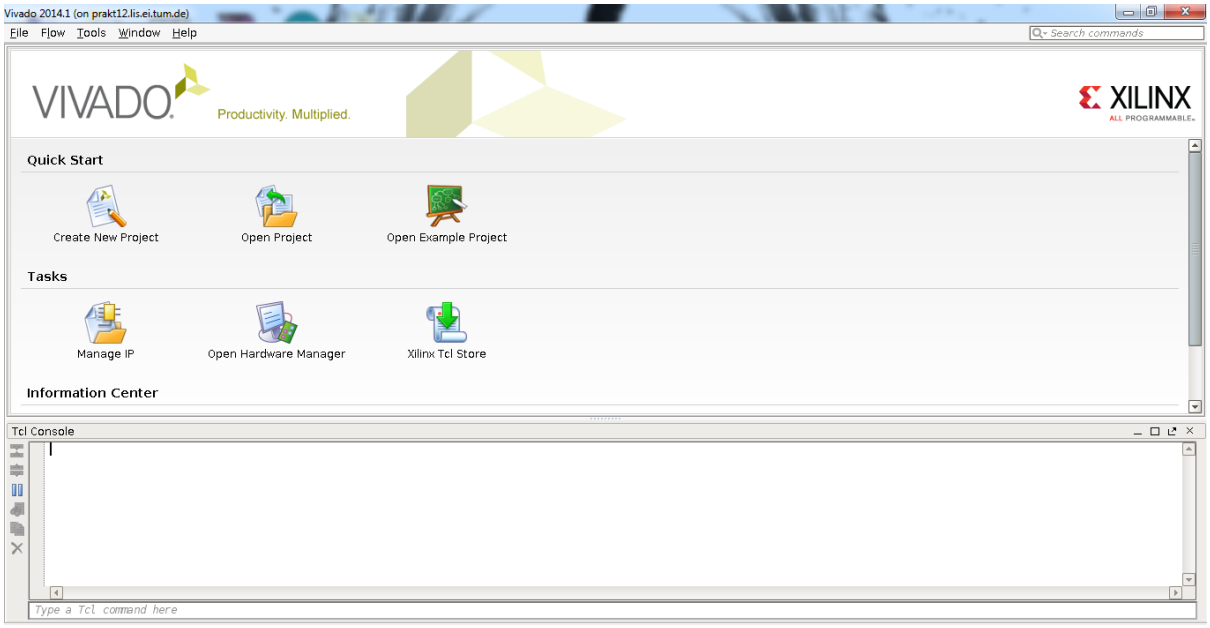
Approach and Tools

Be sure that you have loaded the given parts of the exercise_3 using the below command under your working directory ("hwswc_lab")

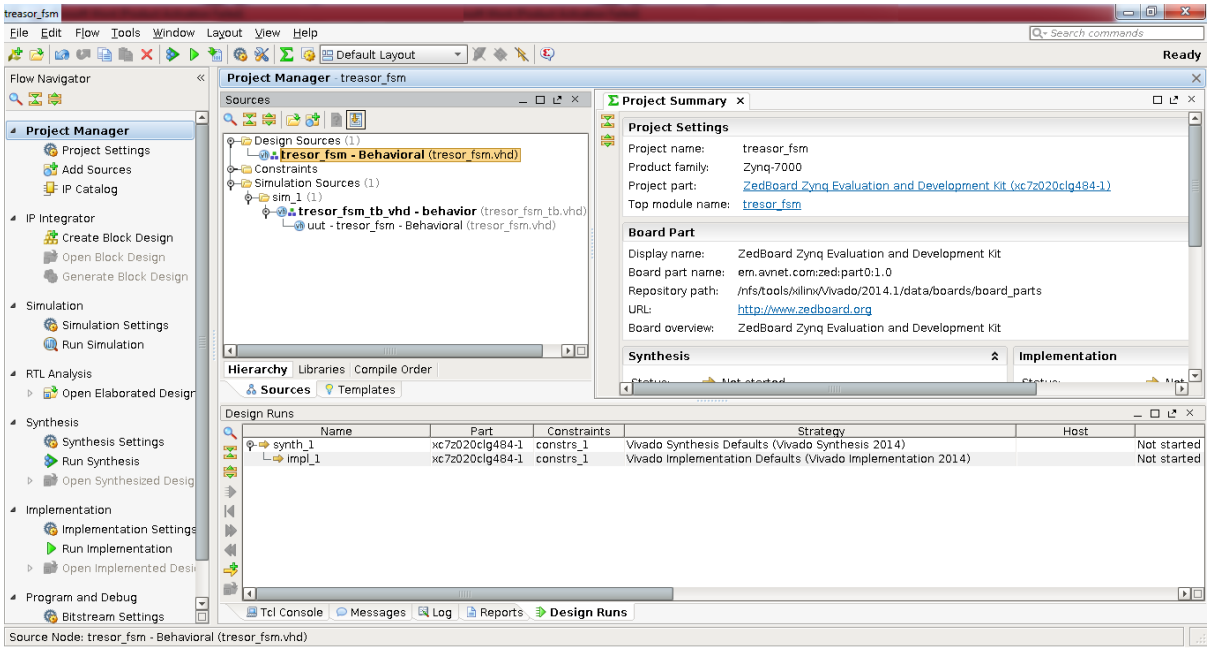
lislab copy hwswc exercise_3

In the directory exercise_3 you will find a tcl file containing the project setup for the Xilinx Vivado. This file references VHDL files with the partially given model that has to be completed by you.

First start the development environment by entering “vivado” on the command line of a terminal window. The following window of the Xilinx Project Navigator should appear.



Select the menu item “Tools -> Run Tcl Script…”, then go to the directory exercise_3 in the file selector box, which is then displayed, and pick the file *tresor_fsm.tcl*. When this file is read into the Project Navigator a script is run to create the project and the two given VHDL files are loaded into the environment.



In the “Hierarchy” tab of the “Source” tab in Project Manager, the source file *tresor_fsm.vhd* is listed under design sources. While the simulation test bench *tresor_fsm_tb.vhd* is listed under simulation source in sim_1 folder.

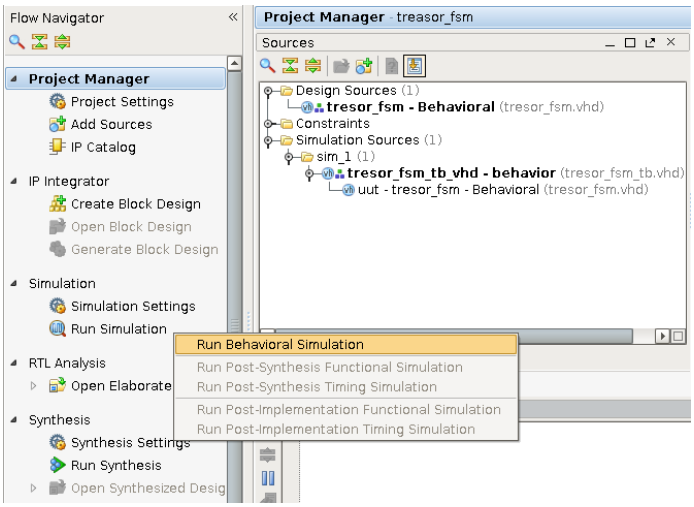
The file *tresor_fsm.vhd* contains the model of the FSM that you should complete. Double-click the file name in the Sources Hierarchy, which opens the file for editing in the main window of the Project Navigator.

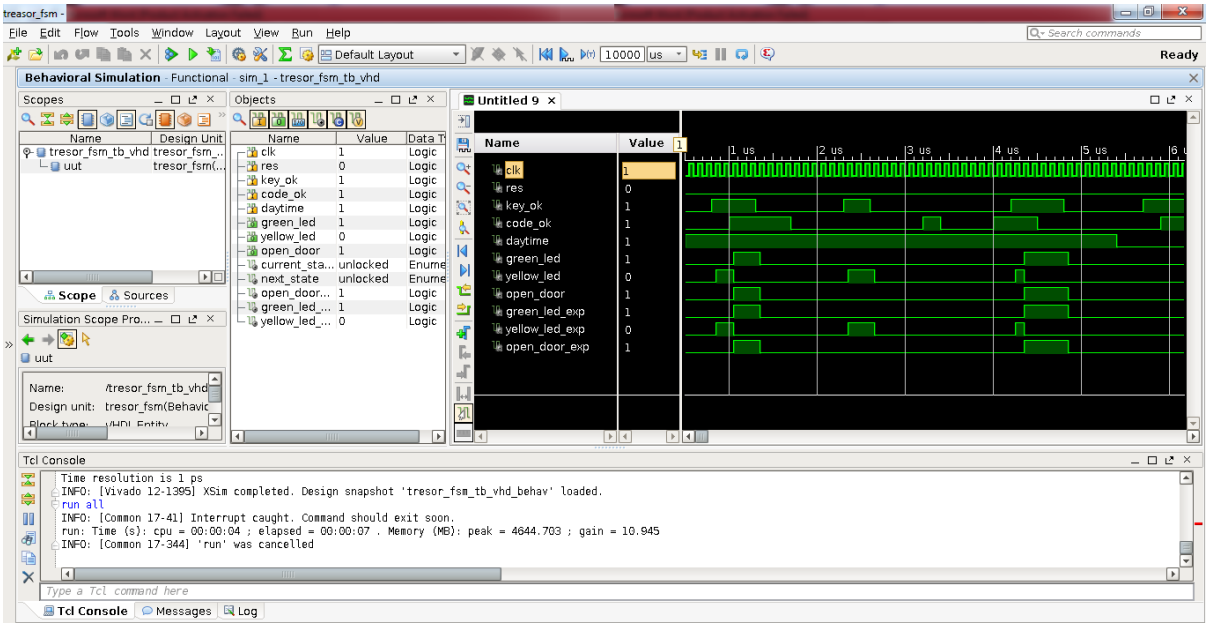
When completing the FSM VHDL code, be sure to implement the version with registers on all outputs. (The version shown in the introductory slides does not contain registers at the outputs!)

Be sure to add only the necessary signals to the sensitivity list of the required processes. Do not forget to save the file after editing.

The second file (*tresor_fsm_tb.vhd)*, which is already complete, contains the testbench with one instance of the FSM named *uut* (unit under test). Moreover, the test bench encompasses processes that generate stimuli for testing the FSM as well as signals with the expected output values (the names of these signals have the extension "*_exp*"), which can be used as a reference.

Once the desired changes are made in the *tresor_fsm.vhd* file the project should be simulated. The project could be simulated by clicking on the "Run Simulation -> Run Behavioral Simulation" under the Flow Navigator. Simulation could also be started from Tcl console using *launch_xsim -simset sim_1 -mode behavioral -noclean_dir* command.



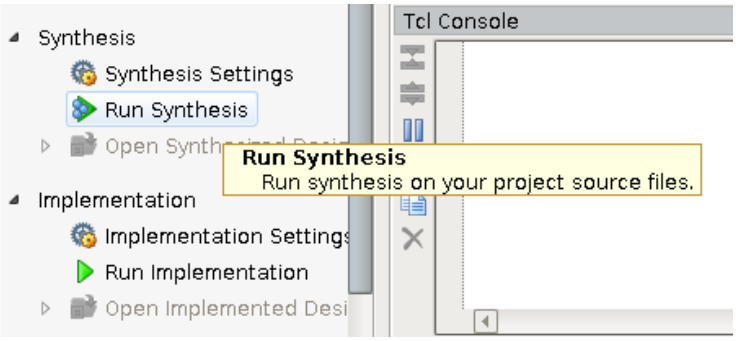


The user interface of Vivado simulation is split up into four parts:

- The left sub-window shows the hierarchy of the design including the processes and component instances of the different modules. (The top level of the hierarchy is always the testbench.)
- The pane in the middle lists the signals of the instance selected in the left sub-window and of the next higher hierarchy level.
- The right pane displays the waveforms of selected signals. The default setting is to display all signals that are available on the top level, i.e. in the test bench. You can add further signals for being recorded by selecting them from the signals list in the middle and drawing them to the waveform sub-window. In the case of the FSM, the signals containing current and next state are of special interest and should be added in this way. However, it should be noted that the values of the signals are drawn only for the simulation time after they have been added to the waveform window. In the second row of icons you can see two symbols of magnifying glasses. Using the "-" lens zooms out of, the "+" lens zooms into the waveforms.
- The wide bottom window gives information on all activities and outputs error messages if something goes wrong.

You can run the simulation for a specific amount of time by entering the corresponding value into the input box in the center of the first row of icons (a numerical value including time unit, e.g. ns or us) and clicking on the icon right of the input box. Alternatively, it is possible to enter the command "*run dur*" in the broad sub-window on the bottom, where *dur* denotes the time for which to continue the simulation (e.g. "*run 10 us*"). A re-start of the simulation can be initiated via the menu as mentioned above, by clicking on the button left to the input box for the simulation duration or by entering "*restart*" in the command window.

After verifying the correctness of your FSM model, return to the Project Manager for synthesizing the model. In the Flow Navigator, select "Synthesis -> Run Synthesis".



The progress of synthesis could be found on top right corner. Once the synthesis is completed successfully, the "Open Synthesized Design" folder in Synthesis tab of Flow Navigator becomes active.

Now you should have a look at the Design Summary tab of the right sub-window and the console output for information concerning the usage of resources by your design and the maximum clock frequency. Individual reports concerning timing summery, clock networks, clock interaction, design rule check, noise, utilization and power could be found in "Synthesis -> Open Synthesized Design" folder in Flow Navigator. The synthesis report can be viewed in the "Reports" tab of the lower sub window of Vivado.

Latches should definitely not have been generated. In case the report says that one or more latches have been generated (section Macro Statistics), you should check whether you have followed the guideline given in the introduction concerning signal assignments in processes. This rule says that all signals that get values assigned anywhere in a process, i.e. that are driven by the process, have to be assigned in every possible execution path through the process. Otherwise old values have to be buffered, and this will be done by the instantiation of latches. Such latches are highly undesirable in synchronous circuits. You can check this in the synthesis report (*tresor_fsm.vds*).

(The synthesis report is found under exercise_3/tresor_fsm/tresor_fsm.runs/synth_1).

Finally, examine the circuit that has been generated by the synthesis tool to get an impression what HW has actually been generated out of your VHDL description. For this purpose, click on "RTL Analysis -> Open Elaborate Design" in Flow Navigator.

The VHDL code of the FSM with registered outputs (*tresor_fsm.vhd*) and the associated synthesis report (*tresor_fsm.vds*) make up the second deliverable of the lab.

Part 3: HW/SW Co-design of Video Edge Detection

Introduction

The final part of the lab addresses the realization of a particular application on an FPGA based prototyping platform. The considered function is the detection of edges within video frames both as a pure SW and as a mixed HW/SW solution, which is finally analyzed in respect to their performance.

Edge detection is the process of locating edges within an image, which is very important for understanding image features. It is a primary step in image processing for boundary detection, motion detection/estimation, texture analysis, segmentation and object identification. Edge detection reduces significantly the amount of the image size and filters out information that may be regarded as less relevant, preserving the important structural properties of an image [1].

The goal of an edge detection algorithm is to locate the sharp changes within the image brightness. There are some well-known methods for edge detection such as Prewitt, Canny, Sobel, and Roberts algorithms, which are different in terms of performance on hardware, speed and simplicity. The Sobel operator is mainly used for hardware implementation due to its efficiency and the simple mathematical model that make it easy for real-time edge detection applications [2]. Therefore, Sobel filtering is applied in this lab experiment. Details on how this works are given below.

System Overview

For this final part of the lab the Zedboard is used as prototyping platform, which is an FPGA board based on the Xilinx Zynq-7000 (XC7Z020) device. In addition to the configurable HW resources this FPGA contains a dual-core Cortex-A9 processing system. The implementation of the Sobel filter is based on a reference design that uses the ARM Corex-A9 processor cores, which is supplemented by additional modules realized on the FPGA resources to establish connection to the peripheral components of the board.

Starting with a pure SW implementation of the Sobel filter in the subsequent experiments the major processing part of the Sobel filter will be migrated into an application specific hardware accelerator on the programmable FPGA resources. The following Figure 1 gives an overview of the overall setup of the HW/SW project. Figure 2 demonstrates the configuration of a Zedboard including the accelerator module. Each lab workstation has an associated Zedboard which is also connected to the network via Ethernet. The Zedboard can be accessed via SSH which facilitates full control from the workstation via network.

The Zedboard is preloaded with “Xillinux” (a Linux distribution) based upon Ubuntu LTS 14.04 for ARM, it makes the board behave like a PC with the SD card as its hard disk. An uncompressed raw video containing 812 frames of 854x480 pixel size in YUV420 format is preloaded on the SD card. This raw video has to be processed frame by frame using the Sobel edge detection filter. The processed video, i.e. the resulting sequence of processed frames, can finally be played using the *mplayer* program in Linux and be displayed on the workstation.

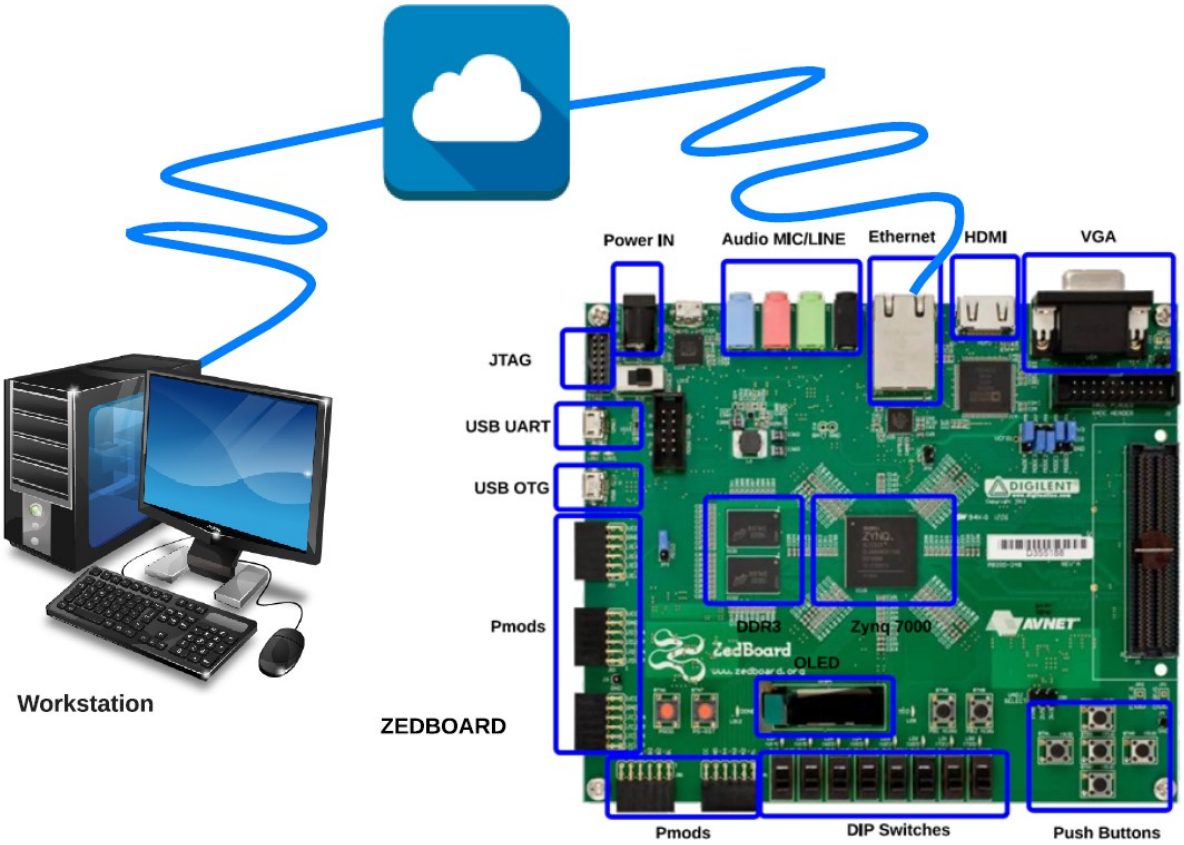


Figure 1 System configuration for the HW/SW project

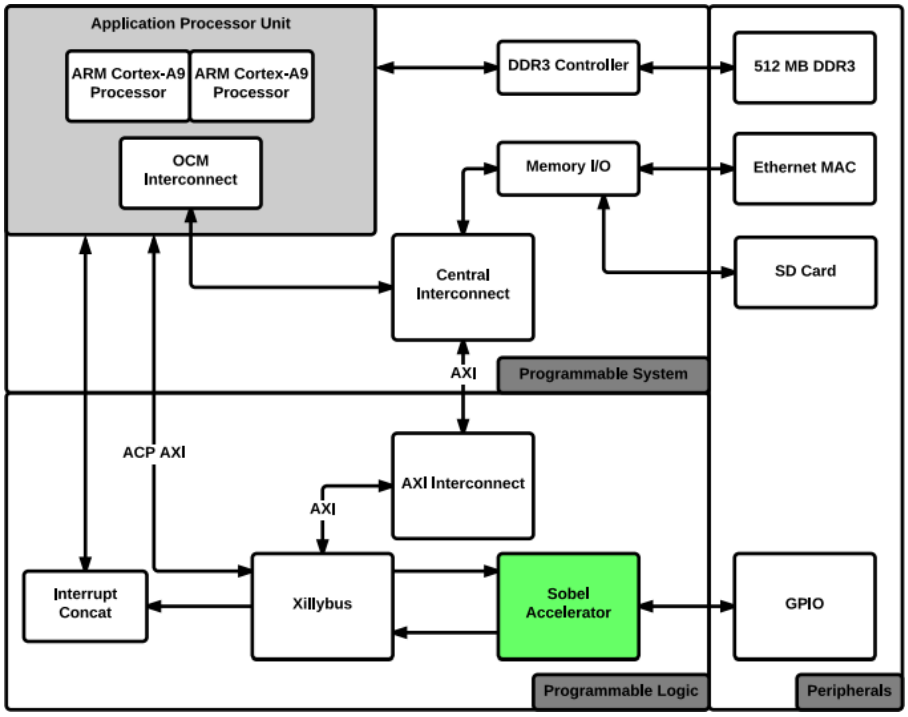


Figure 2 Zedboard configuration for the HW/SW project

This part of the lab is subdivided into the two subtasks “SW development” and “HW development”, each described as a lab exercise in the following. First, the SW implementation of the Sobel filter has to be complemented and analyzed. Then, the Sobel filtering has to be migrated to the HW. The final step is the analysis of the two solutions.

Sobel Edge Detection

Before describing the two exercises of the HW/SW project, an explanation of Sobel edge detection filtering is given, which is sufficient for completing the exercises. As specified earlier, the video provided for Sobel edge detection is an uncompressed raw video containing in YUV 4:2:0 format. Each frame of this video is to be processed individually and only the luminance component (an 8-bit value for each pixel) has to be considered.

Most edge detection methods work on the assumption that an edge in an image is characterized by a discontinuity in the intensity function or a very steep intensity gradient. Using this assumption, taking the derivative of the intensity value of neighboring pixels across the image and finding points where the derivative is higher than a threshold would detect edges in an image [3].

In the Sobel algorithm, horizontal and vertical masks (matrices) are used to calculate directional gradients at each pixel of image. From these directional gradients the overall gradient is computed and compared to a threshold. In case the threshold is exceeded the pixel is assumed to be part of an edge and the luminance value of this pixel in the resulting image is set to 255 (bright). Otherwise the pixel is considered not to belong to an edge and the associated value is set to 0 (dark).

The Sobel mask is very small compared to an image and could be a 3x3 matrix or a 5x5 matrix depending on the requirements. The pixel under consideration is always located in the center of the matrix. The figure below demonstrates the 3x3 Sobel masks for computing the horizontal and vertical gradients for a given image.

Figure 3 3x3 sobel masks for horizontal (left) and vertical (right) gradient

Let us define **A** as the source image matrix containing the pixel under consideration (***e***) in the center of the matrix and its neighboring pixels (***a, b, c, d, f, g, h,*** and ***j***).

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & j \end{bmatrix}$$

Convolving the masks with the original image will result in approximations of the derivatives, one for detecting horizontal changes and another for detecting vertical changes. ***Gx*** and ***Gy*** are the normalized horizontal and vertical derivative of the source image matrix ***A***. In the formula below, "*" denotes the 2-dimensional convolution operation.

$$G_x = A * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad G_y = A * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The resulting G***x*** and G***y*** for pixel ***e*** are:

$$G_x = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & j \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = \frac{1}{8}(a + 2d + g - c - 2f - j)$$

$$G_y = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & j \end{bmatrix} * \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} = \frac{1}{8}(-a - 2b - c + g + 2h + j)$$

The resulting overall gradient ***G*** of pixel ***e*** is then:

$$G = \sqrt{G_x^2 + G_y^2}$$

Finally, the overall gradient ***G*** is compared with the threshold ***T*** to get the dominating edge information. I.e. if ***G*** >= ***T*** then the output pixel at position ***e*** will be set to 255, otherwise to 0.

As the intensity of a pixel (the Y component) is represented in 8 bits, the maximum possible value for a pixel is 255. This results in a white output pixel indicating that ***e*** is part of an edge.

The described operations are carried out for every pixel of the image, row by row from left to right all over the image starting from the upper left corner. If the end of a row is reached it is started again from the leftmost pixel of next row. Pixels outside the image border are assumed to have a luminance value of 0.

The example below demonstrates a gray scale image of rice on a black background. Each value of the extracted 3x3 array denotes the intensity of the associated pixel in 8 bits.

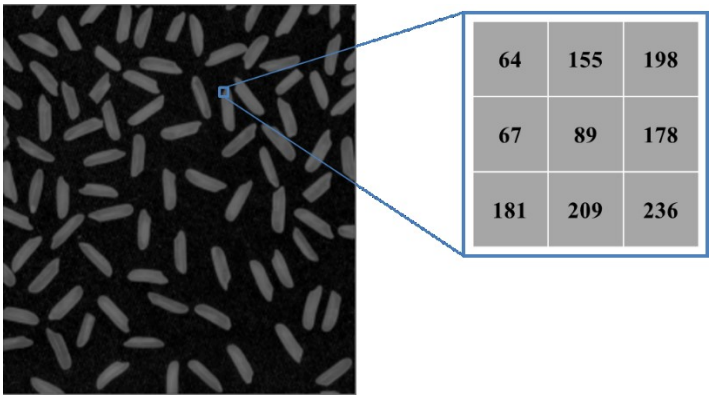


Figure 4 An example of source image matrix for a pixel with intensity 89

Computing the horizontal and vertical gradients using the 3x3 Sobel masks for the pixel location corresponding to intensity 89 of the source image matrix shown in Figure 4 results in $\mathbf{G}x$ = -51.375 and $\mathbf{G}y$ = 32.875.

The Figure below shows the normalized vertical and horizontal gradients of the image in Figure 5.

Figure 5 Image processed with normalized sobel mask for vertical (left) and horizontal (right) gradient

Now calculating the resultant gradient (G) using the horizontal and vertical gradient.

$$G = \sqrt{G_x^2 + G_y^2} = 60.99$$

If the threshold for the resultant gradient is set to be 30, then the resulting pixel value would be 255.

Software Implementation of Sobel Edge Detection

The starting point for the last part of the HW/SW Codesign lab is the SW version of a Sobel edge detection algorithm to be run under Linux on the Zedboard and the analysis of its execution on the ARM processor. Before this can be done the Sobel algorithm written in C has to be complemented.

First open a terminal and change to your working directory *hwswc_lab*. Load the required setup for performing the lab by using *lislab load hwswc*. Then fetch all the necessary files to perform this task by running the command *lislab copy hwswc exercise_4* on a terminal. Next change into the folder *exercise_4/Sobel_SW*. There you find the file *sobel_sw.c* containing major parts of the SW version of the Sobel filtering application. Load this file in an editor and insert the missing parts of the Sobel algorithm in the function *sobel_3x3*. The appropriate section in the file is identified by the comment "WRITE YOUR CODE HERE". When writing the missing C code, please refer to the description of the Sobel algorithm given in the section above and the following additional explanations.

The frames to be processed and also the output are raw images in YUV 4:2:0 format. The pixel arrangement per frame is demonstrated in Figure 6, how they are stored in memory is shown in Figure 7. The component of interest to perform edge detection is only the luminance component Y, which has the full resolution of the image. The chrominance components U and V are subsampled with half resolution in horizontal and vertical direction. They are disregarded in the input frame. For the output frame, these components have to be set to 128 (0x80), for getting a black and white image. (In the given code, the pixels of all components in the output frame are initialized to this value for simplicity and to set also the pixels at the borders.)

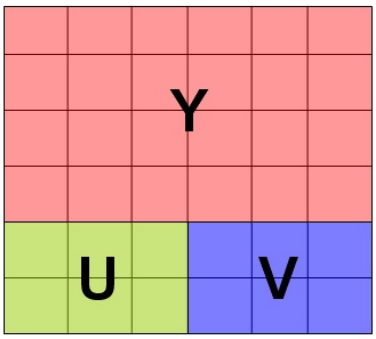


Figure 6 YUV 4:2:0 Frame

Figure 7 Sequential storage of YUV 4:2:0 frame in memory

The overall flow of the program is shown in Figure 8. First the threshold to be used for identifying edges is set. It can optionally be passed as an argument when starting the program. If no argument is given to the program the default value of 20 is used. Then the input file is opened, and memory space is allocated in main memory for the unprocessed and the processed frames. The U and V components of the processed frame are initialized to 128 for gray scale output.

Once all this is done correctly, then edge detection is done per frame in an appropriate loop: An unprocessed frame is loaded from the input file containing a sequence of raw images, then edge detection is done and the frame with the detected edges is written from memory to the output. This is iterated until all frames of the video sequence are processed. The flow diagram below sums up the working of the Sobel filter.

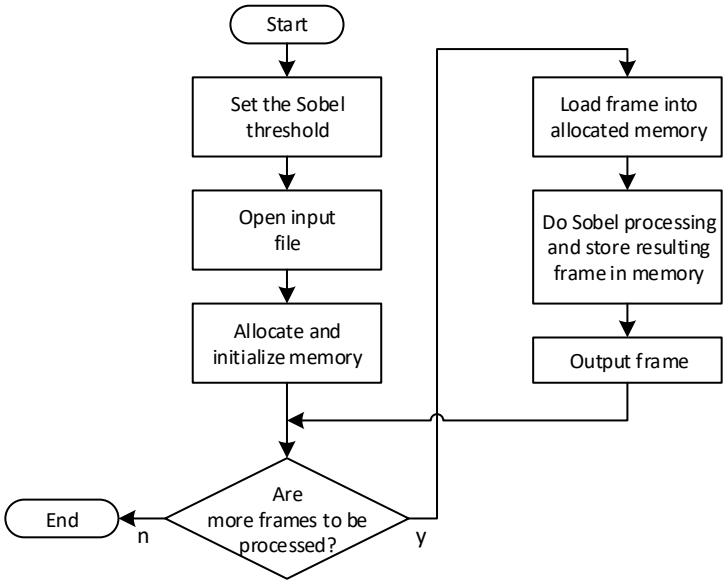


Figure 8 Software implementation of Sobel filter

The actual Sobel filtering is done in the *sobel_3x3* function, which has the following function prototype:

*void sobel_3x3 (unsigned char *in, unsigned char *out, const int frame_width, const int frame_height, const size_t threshold)*

- *in is the pointer to the unprocessed frame in main memory.
- *out is the pointer to the memory allocated for processed frame in main memory.
- frame_width is the width of the frame.
- frame_height is the height of the frame.
- threshold is the threshold for edge detection.

Please complement the code of *sobel_3x3* so that the pixels of the input frame are processed as described in the section above and the resulting pixel value is finally written at the appropriate position in the memory allocated for the processed frame. To go through the image row by row and column by column use two nested loops and calculate the vertical and horizontal gradients using the formulas mentioned earlier. Depending on the comparison of the overall gradient with the threshold the associated pixel value of the output frame is set appropriately.

Hint: At the edges of the frame, where pixels from outside the frame would be required for the calculation of the gradient, the value of missing pixels normally would be assumed to be 0x00. However, to ease writing the code, you can simply skip processing the pixels at the edges of the frame.

To assess the performance of the SW solution the duration of the *sobel_3x3* function is measured. The individual duration of each frame and the mean duration of all processed frames is continuously output on the console. However, be careful, this is only a rough estimate as it does not consider that the Linux OS may schedule other threads and temporally suspend the Sobel application. Nevertheless, the output timing values give a reasonable reference for our further analysis.

When you have completed the *sobel_3x3* function make sure to save the associated file *sobel_sw.c*.

Compile the code using by entering ***make*** while being in the same directory as the modified file to generate the executable file named ***sobel_sw***. The make command will cross compile the code so that ***sobel_sw*** can be executed on the Zedboard ARM processor.

Then it is time to power on the Zedboard adjacent to your lab PC which will also trigger booting of the Linux OS. (If the board is still running switch it off and the on again for proper initialization.) When the boot process is over (about 30 to max 60 s) you can copy your executable to the board.

For doing so, run the script ***load_my_code*** while being in the directory where the executable ***sobel_sw*** is located (i.e. *exercise_4/Sobel_SW/*). The *load_my_code* script uses the *scp* command to copy the binary from the current location to the Zedboard at */tmp/sobel_sw*. Please follow the instructions output by the script.

Then login to the Zedboard using the command ***login_to_fpga***, which is establishing *ssh* connection to the Zedboard adjacent to your lab workstation.

To test your Sobel edge detection program, use command ***run_sw <threshold for sobel filter between 0-255>*** on the Zedboard. A good choice as the threshold parameter is in the range of 8 to 30, however, you may want to experiment with values yourself.

The *run_sw* is actually a shell script that takes care of executing your binary for loading the sample video, processing it and pipelining the output video sequence to the ***mplayer*** program, which displays it in a window on the GUI of the Linux running on the lab PC. You should see the edge-detected version of an animated cartoon.

Have also a look at the processing times of the frames which are written on the console during execution of the program. They should be in the range between 50 and 55 ms.

As a reference you can also display the unprocessed video by entering the command ***play_cartoon***. It is a shell script that calls ***mplayer*** with the required parameters.

Hardware Implementation of Sobel Edge Detection

Now we want to accelerate the Sobel edge detection to improve the performance and get a smoother replay of the edge-detected version of the video. Therefore, we offloaded the Sobel filtering from the CPU and implement it in a specialized accelerator block on the programmable hardware resources of the FPGA. Loading the unprocessed frame and outputting the edged detected frame will stay in software.

Before explaining how the software which utilizes the hardware accelerator should work, we want to look first at the mechanisms to be used for communicating between the processor running the part of the application that stays in software and the hardware accelerator. For this purpose, the Xillybus IP core is employed, which allows communicating between processor and accelerator via FIFOs that are accessed from the software side using conventional file IO. I.e. the software can send data to an accelerator by first opening the device associated with an input FIFO and then writing data to it. The same applies in the reverse direction for reading the results from the associated output FIFO. This allows streaming data to and from the accelerator, which is exactly what is needed for our application example. The input FIFO is used to send the unprocessed frames to the accelerator, the output FIFO for receiving the resulting edge-detected frames. Figure 9 shows the internals of the Sobel Accelerator from Figure 2 with the mentioned FIFOs in the upper part. A third FIFO is shown as well, which we will use for profiling purposes as described later in this section.

The physical connection to the processor system is made up by an AXI Interconnect and the Xillybus IP core, which presents the mentioned FIFOs as memory-mapped Slave devices to the processor. Reading/writing a given number of bytes from/to these FIFOs is handled by the Xillybus device driver and the application software which has to take care that the requested amount of data is written or read to/from the FIFOs. (For details, later have a look at the *allwrite* and *allread* functions in the given code.).

The Sobel Accelerator continuously reads incoming pixel data, applies the Sobel filtering algorithm to them and writes the result to the output FIFO. As each row of pixels is required not only once but three times for processing, two shift registers are used as shown in Figure 9 to keep the rows available for later processing. The outputs of the input FIFO and of each shift register are fed to the Sobel processing unit, which processes the frame row by row according to the parameters of the Sobel mask matrix and the threshold, which is passed through the dip switches (termed GPIO, general purpose input/output in the figure). The data stream from the Sobel processing unit, line-by-line making up the resulting frame, is written into the output FIFO and finally read from the software running on the processor through Xillybus. The pipe control logic keeps the Sobel Processing unit busy as long as there are data in the input FIFO and there is free space in the output FIFO.

The diagrams in Figure 10 demonstrate how the SW which takes use of the hardware accelerator for the Sobel filter works. There are two independent programs: *write_sobel_acc* reads frame by frame from the input file and writes them via Xillybus to the accelerator for processing, *read_sobel_acc* reads the processed frames from the accelerator via Xillybus and outputs them one after the other to be read by *mplayer* (the third program executed as the end of the overall pipeline).

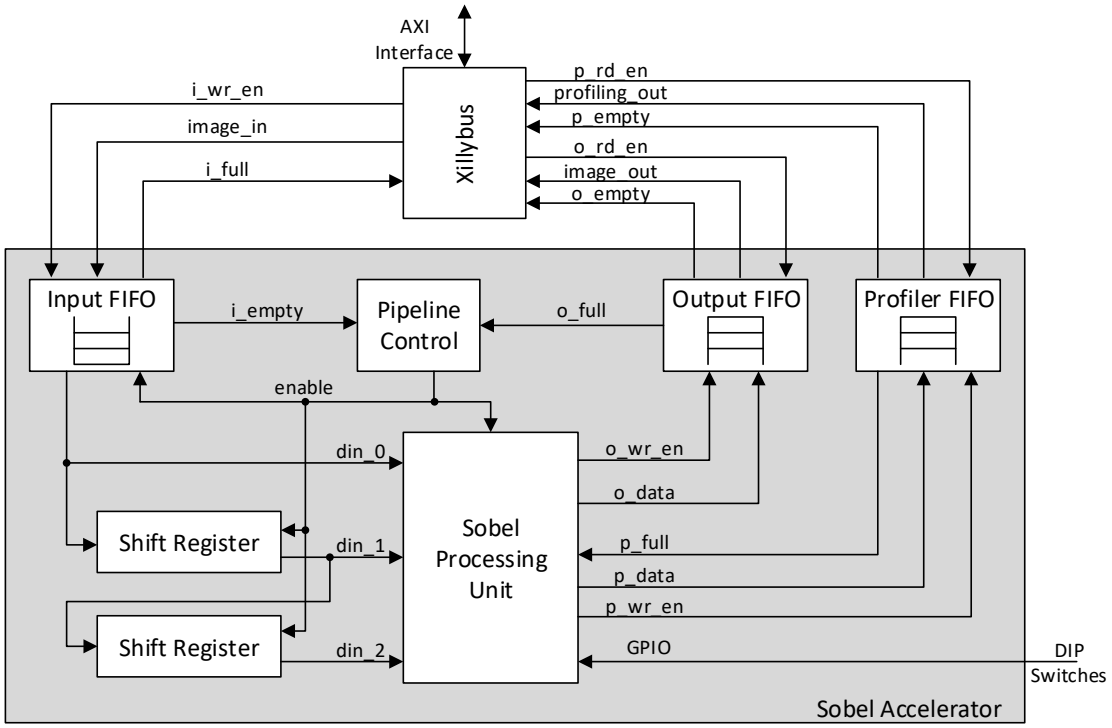


Figure 9 Architecture of sobel hardware accelerator

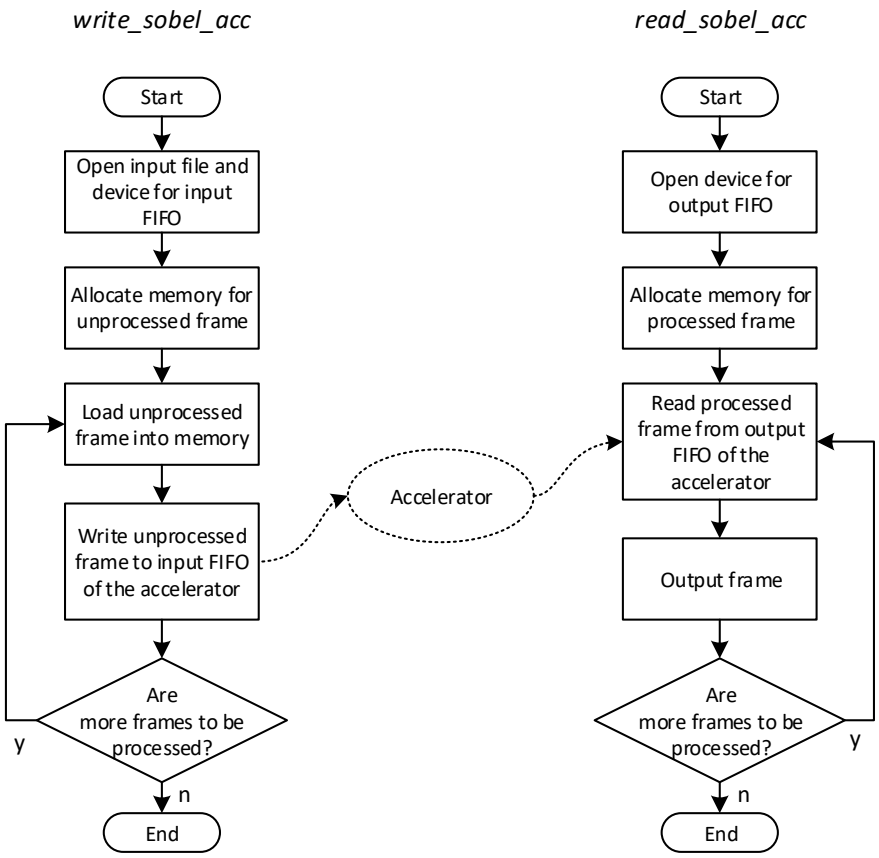


Figure 10 Sobel filtering using the hardware accelerator

First have a look at the software part of the application, which is given in the files *write_sobel_acc.c* and *read_sobel_acc.c*, both of which are in the folder *exercise_4/Sobel_HW*. (The executable versions are already available on the Zedboard Linux and will be appropriately called from scripts described below.)

You should be able to identify the flow shown in Figure 10 within the two source code files.

Now, it is your task to implement the sobel processing unit and the pipeline control as depicted in Figure 9. On a terminal first, go to directory *exercise_4/Sobel_Accelerator/* and start the development environment by entering ***vivado*** on the command line. Select the menu item "Tools →Run Tcl Script…", then go to the directory *exercise_4/Sobel_Accelerator* in the file selector box, which is then displayed, and pick the file *Sobel_Project.tcl*.

When this file is read into the Project Navigator a script is run to create the project and load all the source files to the project. If this script has already has been read in once, the project is displayed under "Recent projects" on the right in the Vivado window after startup. In this case click the associated entry to reload the project.

You should then see the Vivado GUI as depicted in Figure 11.

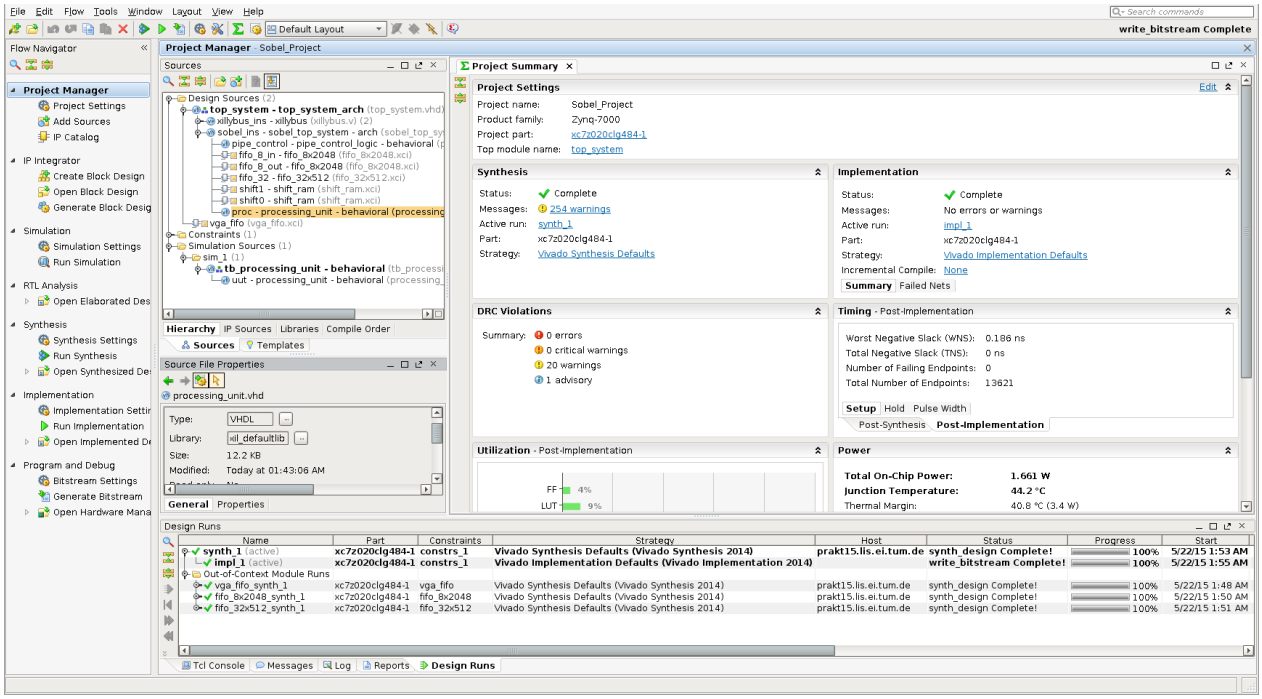


Figure 11 Vivado GUI with loaded project for the Sobel accelerator

In the sources sub-window, you see the hierarchical structure of the overall system. Go down the hierarchy until the Processing unit is visible (see Figure 11). Double click it to open the associated VHDL file *processing_unit.vhd* in the editor.

In the architecture of the *processing_unit* you will find many comments that subdivide the code and explain how the module is working. Many hints are given for writing the missing VHDL code to get an operational accelerator.

In general, the data path of the module consists of a pipeline which is working only if the input *enable* (driven be the Pipeline Control) is '1'. The pipeline should be finally made up of the following stages:

- Three register stages to hold the 3x3 matrix of pixel values for processing. They are generated with the process "*input_regs*".
- Some combinatorial logic that calculates the directional gradients ***Gx*** and ***Gy*** based on the pixel values stored in the registers for the 3x3 matrix.
- The values of ***Gx*** and ***Gy*** are stored in registers to limit the length of the combinatorial path to the output FIFO. Otherwise it would be not possible to operate the circuit with the target frequency of 100 MHz.
- Based on ***Gx*** and ***Gy*** the overall gradient is calculated, and the output value is set appropriately depending its relation to the threshold. This should be realized solely as combinatorial logic.

In addition to the pipeline there are several further parts, which generate control signals for it and also the write signal to the output FIFO. The most important are

- two counters that count the rows and columns of the incoming frame,
- a logic that decodes the two dip switches to be interpreted as threshold value (actually the square of the threshold is set by the switches), and
- a logic that makes sure that the processing pipeline is completely filled up via the external shift registers and the valid results are available that can be written to the o/p FIFO.

As an additional functionality it should be possible to change between processed and unprocessed frame at system runtime via the dip switch SW0 (corresponding to the input ports GPIO(0)) on the Zedboard. Further, the dip switches SW1 and SW2 (corresponding to the ports GPIO(1) and GPIO(2)) of the Zedboard can be used to change the Sobel threshold values used by the *processing_unit* also at run time. (Actually *threshold_square* is changed with these switches. This helps to avoid calculating the square root to get the overall Sobel gradient. Instead the hardware works with the squares of gradient and threshold. For this have a look at the corresponding code.) Threshold values corresponding to the logic are shown in Figure 12. The input port GPIO(7) (Corresponds to the BTNC push button on the Zedboard) is treated as a secondary reset for the processing unit, because the primary reset button on the Zedboard results into rebooting of the Xillinux on the Zedboard.

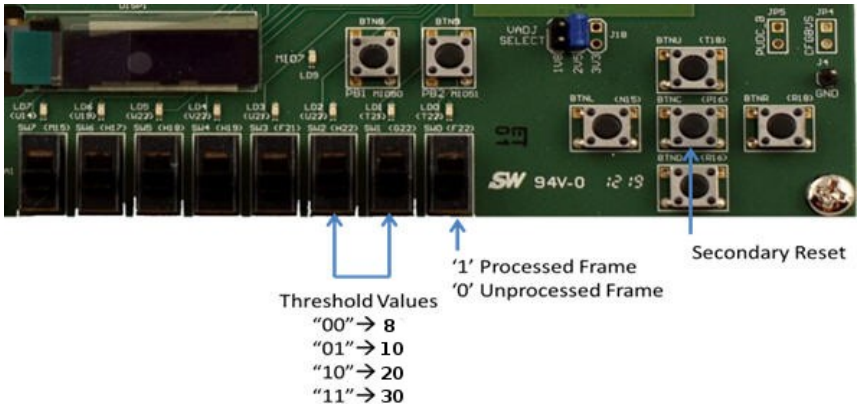


Figure 12 DIP switch settings

Finally, there is also a hardware profiler that counts the number of clock cycles needed for the processing of one frame, beginning with the processing of the upper left to the lower right corner pixel. The counter value encompasses also the clock cycles during which the pipeline is stalled as a consequence of an empty input FIFO or a full output FIFO. (This condition is to be monitored by the pipeline control, which has to set *enable* to '0' in such situations.) The information on frame number and the corresponding number of clock cycles required for processing it is written by the hardware profiler to the profiler FIFO as shown in Figure 9. The *start_prof* command available on the Zedboard Linux reads this information and displays it in realtime. To this we will come later when we analyze the application.

Now, have a closer look at the corresponding sections of the VHDL code in *processing_unit.vhd*. There, you will identify that major parts of the actual Sobel filtering have to be complemented by you (Two sections with the comment "WRITE YOUR CODE HERE"). Don't worry, comments at the appropriate positions give several hints to guide you in writing the code.

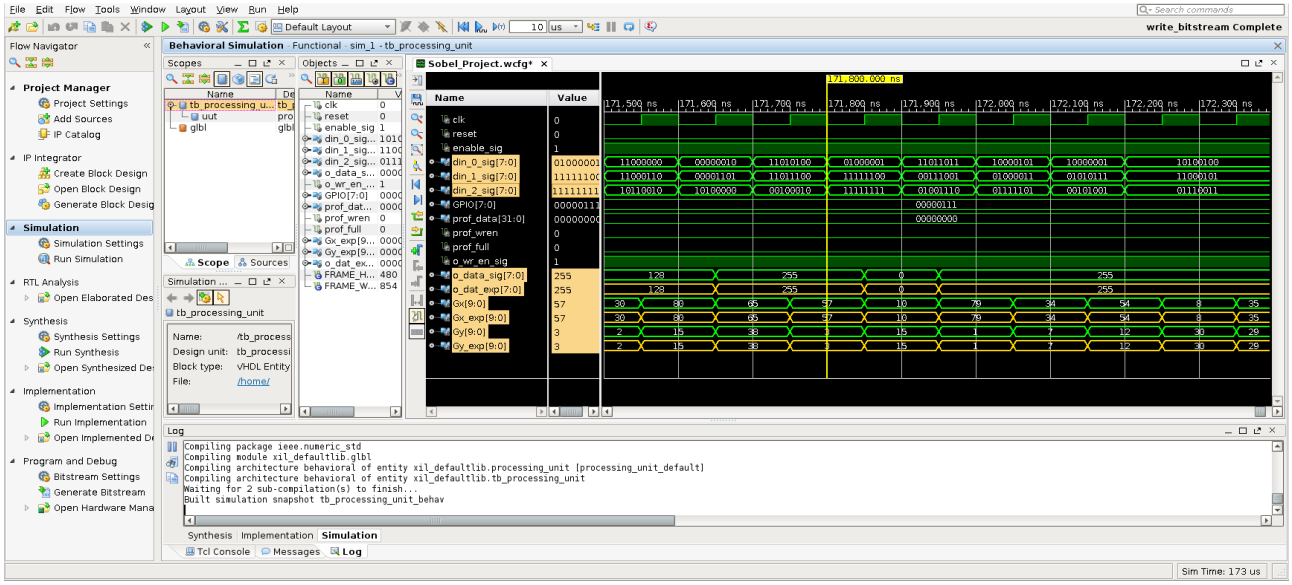
One additional, general hint: Be careful when mixing & and +/- operators in signal assignments. They are of the same precedence and are therefore evaluated from the left. Maybe brackets have to be used at appropriate positions in such an expression to get the desired result.

Once you are done with writing your VHDL code please be sure to save the file *processing_unit.vhd*.

The project contains also a test bench that allows verifying the functionality of your *processing_unit* module. The simulation (only of the *processing_unit* without the other blocks of the accelerator) can be carried out by clicking on "*Run Simulation -> Run Behavioral Simulation*" under the Flow Navigator.

The test bench used for the simulation (*tb_processing_unit.vhd* in the *src* folder) stimulates the *processing_unit* with a given sequence of input data (after resetting it and waiting till internally the Sobel filtering actually starts). In the waveform window the most relevant signals are recorded, and addition expected values for the directional gradients ***Gx*** and ***Gy*** as well as the resulting pixel values of the edge detected frames (signal names with suffix *_exp*).

The waveforms generated by your module should comply with the *_exp* signals (displayed in yellow). (Your signals could be shifted by a few clock cycles if you generated more pipeline stages than expected.)



If you are done with the *processing_unit* open also the file for the pipe control (*pipe_control_logic.vhd*) and add the required assignment to the *enable* output port.

Now you can generate a bit file for FPGA by clicking on "*Program and Debug -> Generate Bitstream*" under the Flow Navigator (it takes 5 to 10 minutes for generating the bit file).

> If it takes too long for generating the bit stream have also a look at the "*Design Runs*" part of the transcript window at the bottom of the Vivado GUI. There you can observe the status of Vivado. If Vivado does not proceed cancel the process via the button on the right top of the main window and restart the bitstream generation process. Sometimes it is also helpful to exit Vivado and start it again. If doesn't help then it is advisable to remove the temporary files from the previous runs (*vivado_pidxxx**, vivado_temp, *.log, *.jou, .Xil in the directory where you started vivado).

Be sure to check whether there are any warnings (or errors) related to the *processing_unit* or the pipeline control logic and to correct your code.

Once the bitstream is generated successfully, exit vivado and make sure the Zedboard adjacent to your workstation is switched on. On the terminal change to directory *exercise_4/Sobel_Accelerator* and load your bit file onto the Zedboard by running ***load_my_bit*** script and follow the instructions.

The script will transfer the bit file and reboot the Linux on the Zedboard. On reboot the programmable logic resources of the device will be programmed with your bit file. After reboot of the Zedboard, reconnect from your workstation using the command ***login_to_fpga*** and follow the instructions.

To verify the functionality of your *processing_unit* execute the command ***run_hw.*** This takes care to call the programs *write_sobel_acc* and *read_sobel_acc* appropriately to establish the data flow according to Figure 10 and send the processed data to *mplayer*. Play around with the dip switches to change the threshold or to select displaying the unprocessed frame. (Note that in this case you will get only a grayscale video as the U and V components have been set to 128 in *read_sobel_acc* before outputting the resulting frame.)

Analysis

When executing ***run_hw*** you may have noticed that in this case also a frame duration is output on the console. This value is even significantly higher than in the software version of the sobel application! How can this be?

The value is coming from the *write_sobel_acc* program, which measures the complete duration of reading a video frame from the input file till it is completely written to the accelerator. The measured value therefore does not necessarily reflect the actual duration of the processing within the accelerator. As we have the hardware profiler let us look at its measurements.

Therefore, repeat the execution with the hardware accelerated Sobel application. However, ***before*** starting the application execute the command ***start_prof***, which will open an extra window for displaying the profiling data. For each frame, the number of clock cycles required for processing in the accelerator and the overall mean will be output. (One cycle means 10 ns, corresponding to a clock frequency of 100 MHz.)

When executing *run_hw* again you should see that the number of clock cycles measured by the profiler corresponds quite well to what is output from the software. Obviously, the duration of the processing in the accelerator is really taking significantly longer than in the pure software version.

However, you may have noticed that the video displayed by *mplayer* in the hardware accelerated version was a bit faster / more smoothly playing than in the software variant, which seems to contradict the measurements we did up to now.

Therefore, let us do a more global analysis by using the ***time*** command of Linux to measure the overall runtime of the two application versions.

Some background on the ***time*** command and the interpretation of its output:

When executing any program with preceding "*time"* (i.e. ***time program***) the operating system measures the execution time of *program* and outputs the following three-time values after program finishes:

- *real* gives the wall clock time it took from start till end of program.
- *user* is the time the processor actually needed for executing the program application code.
- *system* tells how much time was consumed to execute system calls for program (e.g. file IO).

Thus, the sum of user time and system time makes up the time needed on the processor to execute *program*. The difference to the wall clock time comes from other applications or system programs being run on the processor by the operating system interleaved with *program* in time-sliced manner.

Program may be any command including command line parameters.

Now measure the time needed for both variants of the Sobel edge detection (i.e. execute ***time run_sw*** and ***time run_hw***.). Wait till the mplayer has finished playing the complete video. Make note of the time measurements in both the cases.

You should see a significantly lower value for the real time needed for the hardware variant compared to the software variant. (We will only use the user time values in the following.)

This confirms that the hardware accelerated version is really a bit faster than the software only variant. However, this does not explain the discrepancies in the duration of the Sobel processing with the pure software and the accelerated versions as measured before.

As it is known how many frames were processed, determine the duration per frame for both versions respectively based on the wall clock (i.e. real) time values you measured before.

You should see that the duration per frame for the hardware accelerated version corresponds quite well to the values measured before (direct output from the program itself and the measurement from the hardware profiler).

On the other hand, there is a big difference between the mean time for one frame in the software only version and the duration of the Sobel calculation in SW. However, this can be explained when considering the different phases of the processing in the two variants as displayed qualitatively in Figure 13.

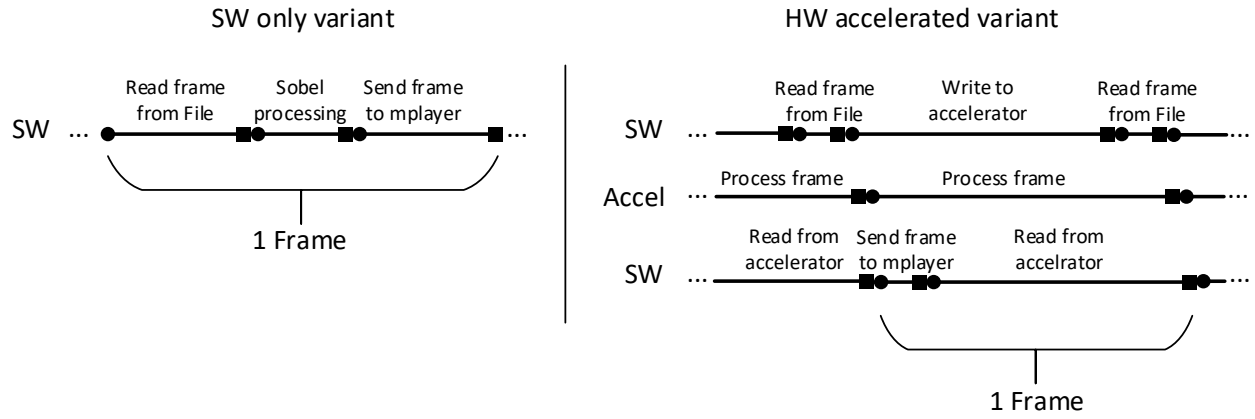


Figure 13 Analysis of Sobel variants

The major principle difference of both variants is the pure sequential execution of all functions in the software only program and the parallel execution in the accelerated version. Even if the accelerator needs longer for processing and data have to be written and read to / from the accelerator this results in a faster overall execution as all can be done in an overlapping manner by the accelerator and the processor cores.

This is still only a qualitative explanation of the speed advantage of the hardware variant. Therefore, we also want to have a closer look at the duration of the Sobel processing in hardware. Remember, this was around 15 to 20 million cycles (output on the *start_prof* window). When considering the actual hardware architecture (see Figure 9) you can see that the accelerator is designed to process one pixel in every clock cycle. As a frame consists of 854x480 pixels the best-case processing would require only around **410 k (!)** clock cycles. This means that out of several millions of clock cycles the accelerator was working only for 410 k, corresponding to a low single digit percentage.

From that we can clearly conclude that the data transport to and from the accelerator is way too slow to optimally profit from the accelerator. On the other hand, even with such a low usage rate we reached a performance gain.

Let us now do a further experiment. Up to now we did not mention the effect of the *mplayer* program, which is run on the processor in parallel (in time-sliced manner, controlled by Linux) and consumes processor cycles as well. This might delay the other programs, in the software case the *sobel_sw* program, in the hardware version our *write_sobel_acc* and *write_sobel_acc* programs.

To analyze this, we have provided two further scripts ***run_sw_null*** and ***run_hw_null***, which do not send the processed frames to *mplayer* for display but discard them instead by sending them to */dev/null*. Thus, the processor is relieved from displaying the results of the filtering. It only has to read frames from the video file, Sobel filter it in the software case or make the transfers to/from the accelerator in the hardware case.

Run now these scripts and analyze the following issues:

- Determine the execution time of ***run_sw_null*** and ***run_hw_null*** and also record the measured durations output by the software and the hardware profiler.
- By how much is the overall execution time reduced in both variants?
- Is the output execution time per frame for the software-based Sobel filtering different than before? What can be concluded from the result? Think about an explanation.
- Calculate the usage factor of the accelerator. Why is it much higher than with the running *mplayer*?
- What can we conclude from the fact that usage of the accelerator is still significantly below 100%?

Considering all results gained in the experiments in this section the following major lessons should have been learned:

- The effectiveness of an accelerator not only depends on its effective implementation but to a very significant extent how well it can be utilized within the overall system. Especially the communication i.e. the data exchange with it can make up a critical bottleneck.
- The overall acceleration of an application is limited if other components like *mplayer* in our case dominate the duration of overall functionality.

Shutdown the Linux on Zedboard by running ***sudo halt***. Wait till your ssh session is disconnected then **switch off the Zedboard**.

As the third and final deliverable of the lab you have to show your running software and hardware solutions by executing *run_sw* and *run_hw*. Details will be announced in time via Moodle.

References

1. I.Yasri, N.H.Hamid, V.V.Yap, “Performance Analysis of FPGA Based Sobel Edge Detection Operator”, IEEE International Conference on Electronic Design, pp. 1 – 4, 2008.

2. R. C. Gonzalez, R. E. Woods, “Digital Image Processing”, Prentice Hall, 2003.

3. O. R. Vincent, O. Folorunso, “A Descriptive Algorithm for Sobel Image Edge Detection”, Proceedings of Informing Science & IT Education Conference, pp. 97-107, 2009.

4. Xillybus, www.xillybus.com