02203 - Design of Digital Systems (Fall 2023)

Edge detection design project

v.7.2

Abstract

This document describes the edge detection design project which is part of course 02203 “Design of Digital Systems” at DTU. The project will result in a FPGA realization of an edge detection hardware accelerator for monochrome video. Edge detection is an important aspect in analyzing a video scene and can be used in for example surveillance systems. The edge-detector is expected to identify abrupt intensity changes in a streaming video which occur at the boundaries of objects. Further analysis of the edges may identify the type of object, e.g., a car or a human.

Using the VHDL language students in the course develops an executable specification of the edge unit and later a synthesizable VHDL model. The latter is synthesized with Xilinx Vivado and implemented on a Nexys4DDR FPGA-board.

Contents

1 Introduction

In this project you will design an edge detection hardware (HW) accelerator for monochrome images. The images are represented in PMG-format and the size of an image frame is 352x288 pixels, i.e., 288 scanlines each with 352 pixels. The intensity of a pixel is represented by an 8-bit number that provide an integer range of 0 to 255, where 0 and 255 refers to black and white, respectively. The pixels are stored successively in a single-port external RAM that uses 32-bit words. Consequently each *read-transaction* or *write-transaction* issued by the HW-accelerator may access four pixels. The original image and the processed image are stored in separate non-overlapping areas in the RAM. The HW-accelerator is triggered by a signal *start* from the CPU and raises a signal *finish* as soon as the last pixel in a frame is processed. The architecture of the system is depicted in Figure 1. The number of frames that your circuit can process per second will depend on your implementation.

You will be given the entity and the architecture body of the HW-accelerator as well as a testbench and a model of the RAM.

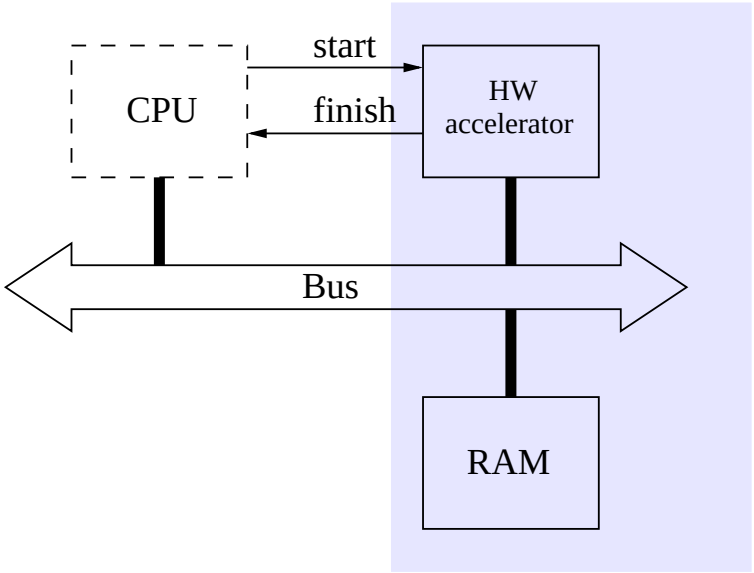


Figure 1: System architecture.

2 Edge detection

The algorithm used for *edge detection* is referred to as the Sobel operator. Edge detection is accomplished by computation of the derivatives of the intensity signal in the *x*- and *y*-direction. Thereafter, the minima and maxima in the derivatives need to be found, indicating where intensity changes most rapidly.

The Sobel method approximates the derivatives in each direction by a process called *convolution*. In other words convolution is performed by an addition of the current pixel **and** weighted (filtered) intensity of the 8 surrounding neighbor pixels. The coefficients of the weighting function are represented by convolution masks G_x and G_y for the derivatives in the *x*- and *y*-directions, respectively, as shown in Figure 2.

G_x=

-1	0	+1
-2	0	+2
-1	0	+1

G_y=

+1	+2	+1
0	0	0
-1	-2	-1

Figure 2: Sobel convolution masks.

The derivative image is computed by centering each pixel with the convolution masks. This needs to be done over successive pixels in the original frame. The coefficients in each mask are multiplied with the underlying intensity and the nine (six) products are added to form two partial derivatives value D_x and D_y, as

$$D_x(n) = G_x * S(n), \tag{1}$$

and

$$D_y(n) = G_y * S(n), \tag{2}$$

where * and n denote the convolution operator and pixel number in a scan line, respectively. The minima and maxima in the magnitude are computed by an approximation as

$$|D(n)| = |D_x(n)| + |D_y(n)|. \tag{3}$$

Finally, the value computed in (3) is stored in the RAM that holds a processed frame, and the Edge detector may continue with the next frame.

The details of the Sobel calculation are illustrated in the following. Figure 3 shows the pixels in a 3×3 pixel fragment of a picture.

s_{11}	s_{12}	s_{13}
s_{21}	s_{22}	s_{23}
s_{31}	s_{32}	s_{33}

Figure 3: Matrix showing nine pixels

The derivatives in the x and y directions for pixel s_{22} in the result picture are approximated in equations 4 and 5. In equation 4 the difference (or change) between the third and the first column is calculated and the pixels in the middle of the column are weighted stronger. The same happens for the first and the third line in equation 5. The value calculated in equation 6 is adapted to the used number range and stored in the memory area for the processed picture corresponding to the pixel s_{22}. It is obvious that this value is big for high intensity changes and small for small intensity changes.

$$D_x(n) = s_{13} - s_{11} + 2(s_{23} - s_{21}) + s_{33} - s_{31} \tag{4}$$

$$D_y(n) = s_{11} - s_{31} + 2(s_{12} - s_{32}) + s_{13} - s_{33} \tag{5}$$

$$|D(n)| = |D_x(n)| + |D_y(n)| \tag{6}$$

3 The HW-accelerator and its interfaces

The HW-accelerator is integrated into a system including also a CPU, a BUS and a RAM module, as shown in Figure 1. The HW-accelerator and the RAM memory will be implemented in the FPGA-chip on the Nexys4DDR board.

3.1 The CPU

In a realistic scenario a CPU would control the HW-accelerator as shown in Figure 1. However, in this project its functionality will be emulated by a push button and a LED on the FPGA board. The push button triggers the accelerator to start and the LED indicates that processing of a frame has finished.

3.2 The HW-accelerator

The HW-accelerator processes 8-bit monochrome CIF video. The bottleneck for the frame rate is expected to be the access to the external memory. The HW-accelerator is triggered by the *start* signal and it raises the *finish* after the last pixel of the last scanline has been processed. This communication is performed with a 4-phase handshake protocol (similar to the SLT-modules we have considered previously).

3.3 The Memory

The memory is a synchronous RAM with a word-size of 32 bits. Figure 4 shows the organization of an image and how the source image and the processed image are stored in separate areas in the RAM. The upper left pixel in an image is pixel (0,0) and images are stored row by row.

We provide two VHDL models of the memory; one for simulation and one for synthesis. From the point of view of the HW-accelerators there is no difference between the two models; they implement the same interface. The only difference is how images are

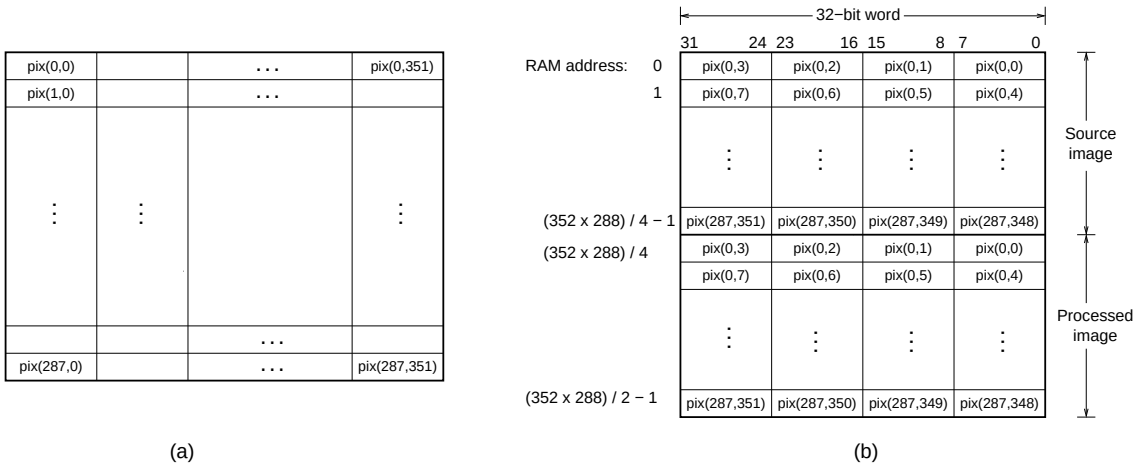
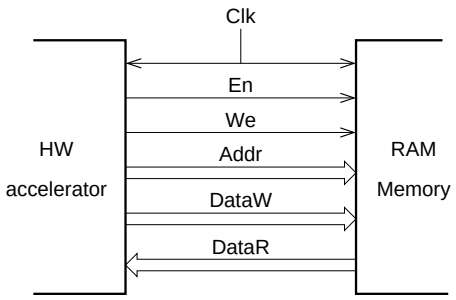


Figure 4: (a) A 352 x 288 pixel image and (b) illustration of how the source image and the processed image are stored in the RAM.

loaded into and read out of the simulation model and the real RAM on the board. These procedures are explained later.

3.4 The Bus Protocol

The interface between the HW accelerator and the memory follow the protocol depicted in Figure 5 and 6, where the former shows the signals and the latter the protocol. The interface is synchronous and only two transactions are provided: *Write Word* and *Read Word*.



En	We	Meaning
0	X	No transaction
1	0	Read transaction (load word)
1	1	Write transaction (store word)

Figure 5: The memory interface.

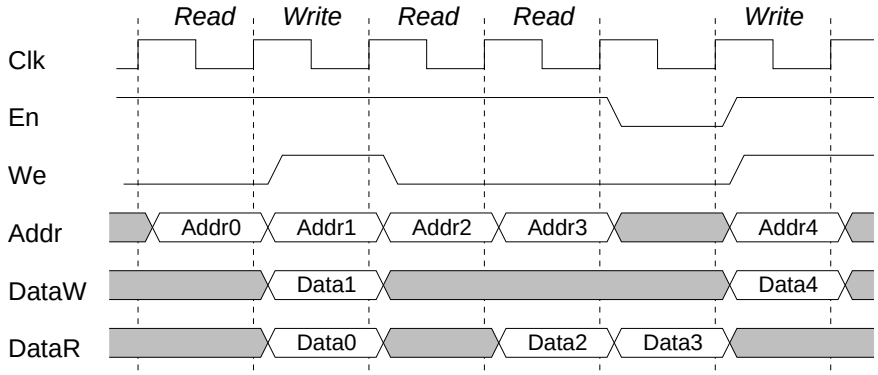


Figure 6: Timing diagram showing read and write transactions.

Using the signal *En* a bus master indicates on a clock cycle basis if it is active or not. *En* is active high, and when it is active the signal *We* indicate if the bus master is performing a read (*We*=0) or a write transaction (*We*=1). The signals En and We are valid from just after the rising edge of the clock and throughout the clock period. Similarly the master must provide a stable address (*Addr*) throughout the entire clock period. During a write transaction the master must provide stable data (*DataW*) for the entire clock period and during a read transaction a slave *must* provide stable data (*DataR*) in the *following* clock cycle. The RAM that we use implements this using an output register and read data is valid immediately after the rising edge of the clock and throughout the entire clock period that follows.

In a real system there may be more masters (CPU, HW-accelerator, etc.) and more slaves (memory modules, IO-units etc.). In this case the bus in Figure 1 should be seen

as a component with multiple ports to which masters and slaves are attached using the interface and the protocol shown in Figures 5 and 6. In this project there is only one master (the HW-accelerator) and one slave (the RAM) and hence they can be connected directly as shown in Figure 5.

4 Design and implementation

4.1 Task 0: Getting started

As the edge detector HW accelerator is a quite complex design and as the design process include procedures for loading and viewing images in simulation as well as on the FPGA board one way to get started is to solve a similar but simpler problem. This can be a HW-accelerator that simply inverts the pixels in the image turning white into black and black into white, e.g., $pix\_result <= 255 - pix\_source$. This can be done by repeatedly reading four pixels, inverting them and writing them back to the memory. This is a much simpler design but it exercises how to access the memory and how to use the image viewing tools. In addition it involves a simple ASMD-chart description and a corresponding FSMD-implementation.

We want you to start designing and implementing this simpler pixel-inversion circuit and in parallel work on task 1; the paper-and-pencil design of your HW-accelerator. For task 0 you can use the same files as we provide for tasks 2 and 3, and you should follow the same steps as described in tasks 2 and 3.

4.2 Task 1: Design of the HW-accelerator

Before coding any VHDL you need a design. This is Task 1. You need to understand the problem and to consider possible implementations. You need to think about how much data your HW-accelerator will buffer internally, how pixels are accessed from the memory, how many times the same pixel is read from memory during the processing of an image frame. In this process you may also try to estimate bounds on the time it takes to process an image. A lower bound can be established by calculating the time it takes your design to read from and write to the memory. You may be able to think of other bounds and estimates that characterize your design. In order to keep the size of the design manageable you may ignore the boundary conditions and simply produce an image that is smaller than the original (missing the left and right columns of pixels and the upper and lower rows of pixels).

Draw a block diagram showing the datapath you have designed and develop an ASMD-chart specification of your design.

4.3 Task 2: Simulating the HW-accelerator in Xilinx Vivado

When you have a design, a sketch of a data path, and an ASMD description of your design it is time to start coding your design in VHDL. We provide you with: (i) a testbench to be used in Xilinx Vivado, (ii) a simulation model (entity and architecture) of the memory and (iii) an entity declaration of the HW-accelerator and an empty architecture body. Your task is to code the architecture body of the HW-accelerator in VHDL; i.e., to describe

your design in VHDL. You can check the correctness of your design by simulating the testbench and comparing the processed image against a reference picture provided by us.

The testbench takes a filename as parameter. This file contains a source image in ASCII format; When you simulate the design using Xilinx Vivado, the source image is first loaded into the VHDL model of the memory, then the simulation is performed, and finally the processed image is written into a PGM-file ready for viewing.

Among the files in the task2 folder is a source image file, *pic1.pgm*. When you run the simulation a result image, *pic1.pgm_result.pgm* is produced. The result image will be written to the simulation folder relative to your Vivado project file (ending with *.xpr*). Specifically, assuming a project name of *MyProject*, the path will be:

proj/MyProject/MyProject.sim/sim_1/behav/xsim

If you want to use your own source image this is possible as well. All the pgm-files can be viewed using the program IrfanWiew.

4.4 Task 3: Synthesizing the HW-accelerator using Xilinx Vivado

The next step is to synthesize your design test your design on the FPGA-board. To do this we need a new top level entity that instantiate a number of components in addition to your edge detection hardware accelerator. These additional components are: (1) a debouncer for the signal *Req*, (2) a clock divider that can provide a clock with a lower frequency than 100 MHz if this is necessary, (3) a memory, (4) a controller, and (5) a UART. In addition to these VHDL-files you will also make use of a small serial interface program that executes on your PC. All these files are in the task3 folder.

The memory we use in task 3 has two ports; the one towards your hardware accelerator (subsection 3.4) and another port towards the controller. Via the UART, the controller "talks" to a serial interface program you run on your PC. This infrastructure allows you to download images from your PC to the memory on the FPGA-board, and it allows you to upload processed images from the memory on the FPGA-board to your PC. This connection uses the same USB-cable as you use for powering and programming the FPGA board. The serial interface program is controlled using a simple GUI shown in Figure 7. Before using the upload or download functionality you first need to setup the serial connection. For details on using the serial interface program click on the Help-button (bottom left).

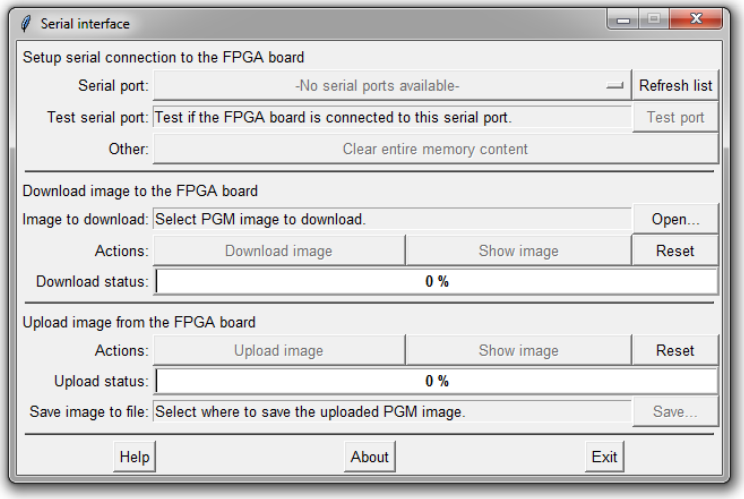


Figure 7: The GUI of the serial interface program.

5 Optional tasks

For those who want to explore more we offer the following ideas.

5.1 Task 4a: Boundary conditions

A simple technique to compute the values at the edges and the corners is mirroring. This may be done by considering the pixel information of the first and the last scanline/column, respectively, as presented in Figure 8. The dotted framed pixels outside the image boundary carry information that is copied form the pixels with a solid frame. To calculate the first pixel in the first scanline you copy the pixel information from the original frame, e.g., to compute the upper left pixel according to equation 3, pixel (0,0) from the original frame needs copied three times as illustrated in the upper left corner.

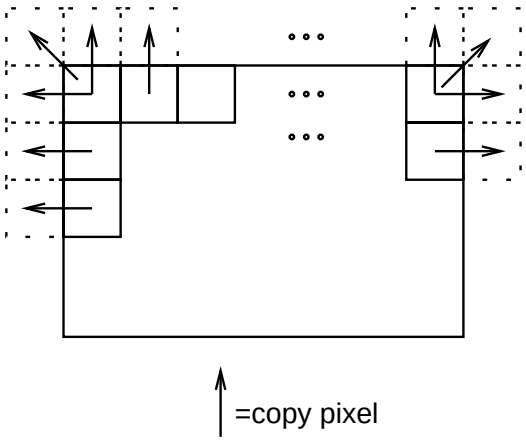


Figure 8: Boundary conditions.

5.2 Task 4b: Using Block RAM or Distributed RAM resources in the FPGA

By buffering whole rows of pixels in the HW accelerator some speedup may be achieved, as pixels are only read once. Such a design may use Block RAM or Distributed RAM resources in the FPGA.

6 Overview of design files

The file *Lab2-EdgeDetector.zip* available in filesharing in DTU Inside contains all the files you need to complete the project. The file contains four directories: task2, task3, serial_interface and other_images. Below you find a description of the files in these directories.

Task 0

Use the files provided for tasks 2 and 3.

Task 1

This is entirely paper-and-pencil design. No files are provided.

Task 2

types.vhd
Defines common types for the all modules.

clock.vhd
A 50 MHz clock generator. Only for simulation.

memory2.vhd
A VHDL *model* of the memory including procedures for reading and writing PGM-files from/to local hard drive. Used for simulation only. Images can be viewed using IrfanView.

acc2.vhd
The HW-accelerator. The entity and an empty architecture body.

test2.vhd
A test bench instantiating a clock generator, the HW-accelerator and the memory. Complete with clock, reset, start and finish signals. The name of the file containing the source image is given as a parameter.

pic1.pgm
A 352x288 source image in PGM-format. Can be viewed using IrfanView.

Task 3

types.vhd
Defines common types for the all modules.

top.vhd
A top-level entity connecting all the components.

clock_divider.vhd
Divides the 100 MHz on-board clock and delivers as default a 50 MHz clock for use in the design.

debounce.vhd
: The push-buttons on the board are not debounced. A digital circuit for debouncing the start signal is needed.

memory3.vhd
: The memory to be implemented on the FPGA board. One port towards the accelerator following the specification in section 3.4 and a second port towards the UART controller.

uart.vhd
: The UART used for serial communication between the controller and PC.

Nexys4DDR_edge.xdc
: XDC-file specifying the pinout of the entire design including UART etc.

acc2.vhd
: Your entity and architecture from task 2.

Serial Interface

serial_interface.py
: Source code for the serial interface program

serial_interface.exe
: The executable code for the serial interface.

HELP.txt
: How to use the tool.

LICENSE.txt
: License.

README.txt
: Readme.

Other images

A few more images ...