CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Bachelor's Thesis

VHDL Teaching Module for TRDB_D5M Camera

Martin Čižmár

August 2019

/ Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Abstrakt / Abstract

Cieľom bakalárskej práce je implementácia knižnice v jazyku VHDL, ktorá môže byť použitá na zachytenie obrazu na úrovni hardvéru. Kamerový modul TRDB_D5M je využitý, ako zdroj obrazu. Naviac, je navrhnutá úloha na spracovanie obrazu, na riešenie ktore, môžu študenti použiť dodanú knižnicu. Inštrukcie k implementácií sú dodané taktiež.

Preklad titulu: Výukový VHDL modul pre TRDB_D5M kamerový modul

The aim of the bachelor thesis is the implementation of a library written in the VHDL language, used for capturing an image at the hardware level. The TRDB_D5M camera module provides the image. Furthermore, an image processing task is devised, that the LSP students can solve using the library. Implementation instructions are supplied too.

/ Contents

Chapter 1
Preface

At first, the task to design and implement a library for taking and processing picture seemed feasible and easily doable. However, that was only because the author did not have enough practical knowledge to spot the hidden complexities. So, as expected, the complications started arising during implementation, the most prominent being the inclusion of SDRAM and FIFOs.

The library is composed of components implemented using VHDL-2008 dialect, and the author strived to test and verify all the designs in a repeatable manner. It contains modules helping with:

- Reading data from the sensor
- Configuring the sensor
- Converting pixel format
- Framebuffer usage
- SDRAM usage
- VGA display
- An implementation of the image processing task

As a side note, the authors' secondary objective was using as many open-source components and tools, as possible, during the development.

Chapter 2
Motivation

The author has a deep interest in processor architectures and even successfully designed a very simple microcontroller during the LSP school subject. However, the complexity of the imaging library is incomparable in scope and the depth of knowledge required. At the start of the thesis, the author did not have an in-depth understanding of FPGAs, digital design, hardware description languages, or testbenches. The digital design is an interesting subject, as it allows the programmer, to tap into a parallel nature of the hardware. In authors eyes, the most significant problem during design phase is a lack of proper, free, and open-source tooling. So, the author would like to acknowledge and thank the developers of these tools:

- **GHDL**[1] (free, open-source)
  GHDL is a functional VHDL simulator, supporting up to VHDL-2008 language standard. It was used to simulate, test, and verify all the library designs. During the thesis, a few bugs manifested in the simulations, but the developers were quick to fix them. Moreover, it is quick and outputs the waveforms in a custom format, understanding, and correctly displaying even the more advanced VHDL data types (code blocks, records, type enums).
- **GTKWave**[2] (free, open-source)
  GTKWave is a waveform viewer. It supports both digital and analog value dumps and also supports the custom format used by the GHDL simulator. The viewer was used extensively during testing and verification. The GTKWave also supports custom signal decoding filters, and this functionality was used during the SDRAM controller implementation to decode its commands.
- **Sigasi**[3] (commercial, free educational licence, closed-source)
  Sigasi is a VHDL IDE based on the Eclipse project. The editor is commercial but free for educational purposes. It helped immensely during project implementation, and the author honestly cannot believe how anybody can design hardware without essential functionality, that software developers take for granted. The Sigasi IDE provides:

  - syntax highlighting
  - advanced type analysis and autocomplete
  - design refactoring
  - intelligent autocomplete
  - simulation tools integration

Furthermore, these open-source VHDL libraries are used:

- **OSVVM** (Open Source VHDL Verification Method)
  OSVVM is a library implementing utilities to help with the implementation of a testbench. It was mainly used to provide better logging and alerts, to implement an SDRAM model memory backend, and to help with synchronization of some testbenches.

- **PoC** (Pile of Cores)

  PoC is an open-source IP core library, published and maintained by Faculty of Computer Science, Technische Universität Dresden, Germany. It provides a clock-independent FIFO implementation used in the data controller.

Chapter 3
Board descriptions

3.1 The DE2 board

The DE2 is a development and education board aimed at digital hardware design teaching using FPGAs. The supplied documentation covers almost all the used auxiliary board chips. Terasic also provides examples of some possible projects, written in Verilog. By default, after power-on, the board starts up in in a factory mode but can be re-programmed using the built-in JTAG interface.

The heart of the board is a Cyclone II 2C35 FPGA chip manufactured by Altera (currently Intel). It is a low-end, low-cost, FPGA model, introduced to the market in the year 2008. It contains[4]:

- 33 216 LEs (logical elements)
  A logic element is the smallest unit of logic in the Cyclone II architecture. It contains a four-input lookup table, a register, a carry chain connection and it can drive any interconnect (row, column, and others).
- 105 M4K RAM blocks (with a total capacity of 483 840 bits)
  The memory blocks are distributed around the internal FPGA fabric and provide support for FIFOs, shift registers, and other types of data storage.
- 35 embedded multipliers
  The multipliers can operate at up to 250 Mhz, supporting one 18-bit multiplier or two independent 9-bit multipliers.
- 4 PLLs (phase-locked loop)

The DE2 board contains a lot of additional supplementary hardware and chips to enable rapid prototyping. For example[5]:

- SRAM
- SDRAM
- Flash Memory
- SD Card Socket
- Toggle/Pushbutton switches
- 50/27 MHz oscillators
- Audio CODEC
- VGA output
- NTSC/PAL TV decoder
- Ethernet controller
- USB Slave/Host controller
- Serial ports
- IrDA transceiver
- Two 40-pin expansion headers

3.2 The DE2-115 board

The DE2-115 is a development board using a high-performance Altera Cyclone IV E FPGA core. It is best suited for task that require better DSP capabilities. Same, as the DE2 board, it supports various peripherals, and contains a lot of inbuilt hardware, ready to be programmed.[6]

3.3 The TRDB-D5M camera module

The TRDB-D5M is an addon card for FPGA boards sold by Terasic, containing a CMOS imaging sensor plus supporting circuitry. The author believes that the actual chip is made by Micron (model number `MT9P001I12STC`).[7]

The primary silicon, as stated, is a CMOS imaging sensor, with a 2752 pixels wide, 2004 pixels tall sensor array. It consists of three types of pixels: Dark, Active, and Active boundary, where the Active boundary and Dark pixels surround the central Active ones. Boundary pixels can be used to avoid fringing effects when doing color processing to achieve full resolution of 2592 x 1944. Dark pixels are typically only used for internal black level calibration, nevertheless can be used too, if configured.

The image data is read out serially, starting from the upper-right corner facing the sensor, line after line in a Bayer pattern format. Thus, each pixel consists of four sub-pixels: Green1, Green2, Blue, and Red. Furthermore, vertical and horizontal blanking periods surround active pixel output, occurring after each line or frame, respectively.

The TRDB-D5M is runtime configurable using an I2C-compatible bus.

Chapter 4
The library description

Library implementation consists of several loosely-coupled entities, divided into these categories:

- Pixel processing pipeline
- I2C communication
- SDRAM controller
- VGA controller
- Data arbitration
- Packages
- Testbenches & models

The pixel processing pipeline is designed in a way, that each stage feeds the current state and data to the next one, as early as possible. Because of that, we do not need to use much memory for pixel caching and concern ourselves with timing delays. Internally, a single image is represented using a record of three 8-bit unsigned signals, one for each of Red, Green, Blue colors.

While it seemed, that FPGAs' internal memory or the external SRAM could be used for as framebuffer, that is not the case. Both have insufficient storage to cache a full image in VGA (640 x 480) resolution, at 8-bit color depth. A minimum of 640 * 480 * 24 bit = 7 372 800 bits = 900kB is required, while the SRAM can store 512 kB and FPGA memory blocks can store approximately 60kB. While the DE2-115 board has an SRAM with higher capacity, it was not considered, to stay compatible.

4.0.1 Pixel processing pipeline

The first pipeline stage is demosaicing (*ccd_demosaic.vhd*), which converts the pixels from Bayer format to a more conventional 8-bit RGB format suitable for storage and further processing. Next, an image convolution (img_convolution.vhd) implementation (to be implemented by the student) modifies the image according to the parametrized kernel and passes pixels to the data controller.

4.0.2 I2C communication

Configuration controller (*i2c_ccd_config.vhd*) is in charge of sensor configuration, working in tandem with i2c master controller implementation (*i2c_ctrl.vhd*). Where i2c_ccd_config handles configuration parameters sequencing, and i2c_ctrl is in charge of sending the requested data using the i2c bus.

4.0.3 SDRAM controller

A burst controller with help from memory initialization controller manages access to SDRAM. After reset, only the initialization controller controls the memory bus, and after successful initialization of memory, the burst controller is ready to handle full-page writes, full-page reads or memory refreshes.

An SDRAM library implementation simplifies interfacing with memory, provides auxiliary functions, synthesis-time computed timing constants, and abstracts away command encoding and decoding.

4.0.4 VGA controller

VGA controller implementation is simple and straightforward. The only reason for custom design is specific data path requirements.

4.0.5 Data controller

The data controller (*data_ctrl.vhd*) provides the FIFO arbitration. It:

- Buffers the incoming pixels from CCD
- Buffers the outgoing pixels to VGA
- Manages the SDRAM framebuffer

A pair of FIFOs and an SDRAM burst controller primarily accomplish the task. The main objective of data_ctrl is then to manage access to these resources and to make sure that FIFOs do not get filled, or run out of pixels, while periodically refreshing the memory.

4.0.6 Packages

The modules use various constants, functions, and type definitions from their respective packages. The moderate use of precisely constrained types helped catch many `off-by-one` errors during design testing. The use of protected types and shared variables further eased testbench construction. Notably, in the case of SDRAM model, the validity of written data can be reliably tested by faking data in memory cells.

4.0.7 Testbenches & models

Nearly all designs are tested using automated testbench, with help from SDRAM, CMOS sensor, and i2c slave models. Furthermore, an open-source OSVVM (Open Source VHDL Verification Methodology) library helped significantly, as it further eased testbench construction. It provided useful utilities such as random data generation, configurable logging and output, memory models, and test alerts.

Chapter 5
A camera processing task definition

The proposed task is a 2D image convolution using a 3x3 kernel and optional prescale.[8]

Image convolution is a technique frequently used in digital image processing for sharpening, blurring, edge detection, and for other various effects. To compute the value of a convolved pixel, we first flip the convolution kernel vertically and horizontally.

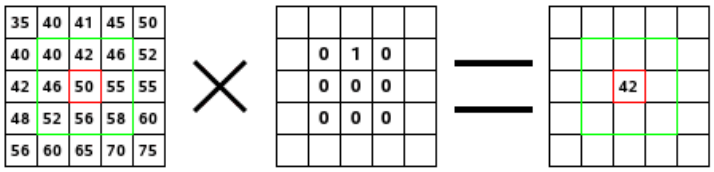

Figure 5.1. A convolution process example.

Next, the kernel starts in a top left image corner and moves over each pixel sequentially. At each position, it overlaps a few pixels on the image. To compute the value of the pixel in the center of the convolution matrix, we multiply pixel values by the corresponding kernel weights.

Next, we sum up the matrix values and optionally divide or multiply the result by a prescale amount. Finally, the number is saturated (`clamped`) to fit the pixel data width. The saturation rules are as follows (assuming pixel value is an unsigned number):

- If pixel value $\geq$ *MAX_UNSIGNED*, the result is *MAX_UNSIGNED*
- If pixel value ≤ 0, the result is 0
- Otherwise, the result is unchanged

The image kernel usually requires pixels outside of the image boundaries. Handling of the edges is up to the implementation, and several options exist:

- Extend
  The edge values are extended to cover the whole kernel.
- Wrap
  Pixel data is taken from the opposing edge.
- Crop
  Values on the edge are skipped, reducing the output dimensions.
- Kernel Crop
  Any pixel that extends over the edge is not used. The normalizing factor is adjusted for the fact.

The task, and the library implementation, uses a Crop method to deal with edge artifacts.

Chapter 6
Library implementation

6.1 CCD controller

The image sensor output is largely stateless, signaling only a vertical or horizontal blanking. For use in the library, more information is required. A CCD controller implementation (*ccd_ctrl.vhd*) provides just that, by saving the actual state in registers. The internal logic is relatively simple, consisting of three registered counters: height, width, pixel.

While the pixel counter could be used to compute relative pixel coordinates in the frame (because we know image dimensions beforehand), modulo operation using numbers other than the power of 2 is hard to implement in logic efficiently. Each new valid pixel increments the counters, and a frame end resets them.

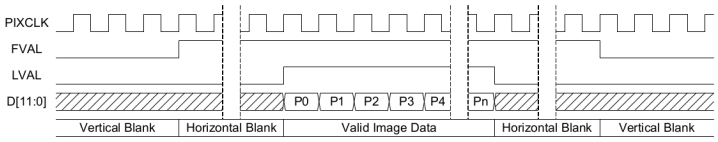

Figure 6.1. A sensor output timing diagram.

The controller determines the validity of a pixel by checking *FRAME_VALID* and *LINE_VALID* outputs from the sensor. Both signals are active high. *FRAME_VALID* determines whether the current pixel is part of a vertical blanking interval, while *LINE_VALID* determines whether the pixel is part of a horizontal blanking interval. Decoding is as follows:[7]

- *FRAME_VALID* = '1' and *LINE_VALID* = '1' - pixel is valid
- *FRAME_VALID* = '1' and *LINE_VALID* = '0' - horizontal blanking
- *FRAME_VALID* = '0' and *LINE_VALID* = '1' - vertical blanking
- *FRAME_VALID* = '1' and *LINE_VALID* = '1' - vertical & horizontal blanking

The controller provides these pieces of information (all are active high):

- *pixelValidOut* - current pixel is part of active image
- *frameEndStrobeOut* - a strobe signalizing end of the frame (all pixels of the current frame have been output)
- *hBlankOut* - sensor is in a horizontal blanking interval
- *vBlankOut* - sensor is in a vertical blanking interval
- *heightOut* - relative y coordinate of pixel in a frame (zero-based)
- *widthOut* - relative x coordinate of pixel in a frame (zero-based)
- *pixelCounterOut* - relative order of pixel in a frame (zero-based)

The pixel data width is truncated from 12 bits to 8 bits, discarding lowest 4 bits, as this makes processing easier and occupies less space. Furthermore, VGA output can only handle up to 8 bits (DE2-115) or 10 bits (DE2) of color information.

The *ccd_pkg.vhd* package contains a lot of convenient constants and type definitions concerning CCD data and signals.

The functionality of the module is validated using automated testbench implemented in *ccd_ctrl_tb.vhd*, with the help of CCD model.

6.2 Demosaicing description & implementation

The first step in the pixel processing pipeline is demosaicing (debayering). As stated, pixels from an image sensor are output in a Bayer pixel format, where each pixel consists of four subpixels: Red, Green1, Green2, and Blue. To reconstruct full color from these pixel samples, we have to employ somehow compute the other two missing colors for each pixel. As this process is well-understood, there exist many algorithms. However, because we do not require superior quality, and we can tolerate some edge artifacts, we use simple interpolation.

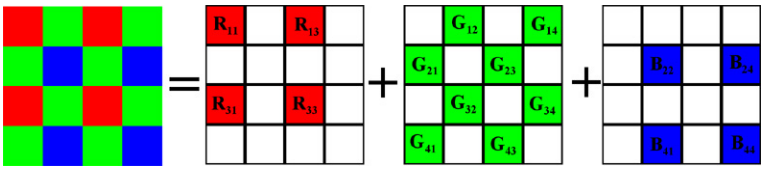

Figure 6.2. An example of subpixels in a Bayer pattern.

In simple interpolation, a color of current Bayer pixel is output as-is, whereas the other two color components are computed by averaging over nearest neighbors in a 3x3 grid. The neighboring pixels in a Bayer pattern are non-uniform; not all pixel types are surrounded by the same colors, and we have to handle each combination explicitly.

Furthermore, to keep track of the current pixels' color, we have to know its relative location in a frame. The CCD controller provides the state information, and the color of a pixel is computed using its relative (zero-based) height and width within a frame. In the default (non-mirrored) mode, the decoding is as follows:

- width is EVEN; height is EVEN - Green1
- width is EVEN; height is ODD - Blue
- width is ODD; height is EVEN - Red
- width is ODD; height is ODD - Green2

As the sensor outputs pixels serially, a shift register implemented in *pixel_shiftreg.vhd* is used to store pixel and its neighborhood temporarily. The shift register outputs a pixel and its neighborhood using a three by three 2D array. Pixels located on the edge of the frame are used only during demosaicing and are not output; thus, the shift register only incurs a one cycle delay during processing. As was stated in the CCD controller description, a modulo operation is costly. So, a pair of registers keep track of the height and width on the output of the shift register.

A 2 stage pipeline performs the color averaging. In the worst (performance-wise) case, we have to average over four values, for each of the colors. To achieve high performance and meet the timing requirements; a pipelined binary adder tree is used, as the Cyclone II LEs are composed of 4-input lookup tables. In the first stage data values are widened, to prevent overflow and added in pairs of two. In the second stage, values are finally added together and averaged. As pixel counts are powers of 2, the division is synthesized as a simple bit shift.

The functionality of *ccd_demosaic.vhd* is verified and tested using testbench implemented in *ccd_demosaic_tb.vhd*. During testing, we fill the mocked frame output array

(exposed as a shared variable in *ccd_ctrl_pkg.vhd*) with random data, at the beginning of each frame. Correct algorithm implementation is checked by exploiting the direct access to the mocked frame array and comparing it with the output of the tested module. Furthermore, we test whether the other signals behave according to the functional specification.

6.3 Image convolution description & implementation

A file *img_convolution.vhd* provides a possible implementation of the image convolution, using a 3 x 3 kernel and optional prescale. The flipped convolution kernel and an optional prescale amount are provided as generic parameters in the module. The implementation uses a 4-stage pipeline design, to process incoming pixel according to flipped convolution kernel defined as the generic parameter, as image convolution is a computationally demanding process combining both multiplication and addition. Similar to the demosaicing, image convolution employs a shift register, to store pixel including its neighborhood. However, unlike in the case of demosaicing, the amount of used shift registers is threefold, as each color uses one. The amount of used memory blocks is still manageable, as they occupy 4kB. As in the demosaicing, we keep track of pixel at the output of the shift register.

In the first stage, the pixels are widened to 9 bits, so as not to be truncated while being converted to a signed (two's complement) representation. Next, the datapath is widened to accommodate resulting values further down the pipeline. Precisely, we are multiplying a signed pixel value (9 bits) by a signed kernel coefficient (5 bits), and so, at least 14 bits are required to store the result. As the multiplication result is then progressively summed up during the following stages, the accumulator is widened by one bit for each stage, to prevent over- or underflow. So, the internal data width used internally is 17 bits.

In the next two stages, the results of the multiplication are progressively added up using an adder tree design, to adjust for Cyclone II internal architecture. In the last stage, the last data summed and then divided by the number 2 exponentiated by the prescale amount. The pixel is first saturated, meaning that, if its value is below zero, or above 255 ($2^8 - 1$), it is rounded to fit in the 8 bits.

The design is tested using automated testbench in file *img_convolution_tb.vhd*. The same approach to testing is used, as in the case of convolution. The simulated CCD image array is filled with random values at the start of each frame. Next, the outputs of the implementation are verified against the ones computed using direct access to the image. Standard sanity checks are also used.

6.4 I2C Bus

6.4.1 Controller implementation

A TRDB-D5M chip is configured using the I2C-compatible bus specification. While the bus type is not mentioned explicitly in the documentation, its specification is consistent with that of an I2C bus. To interface with the sensor, a master controller is implemented in *i2c_ctrl.vhd*.

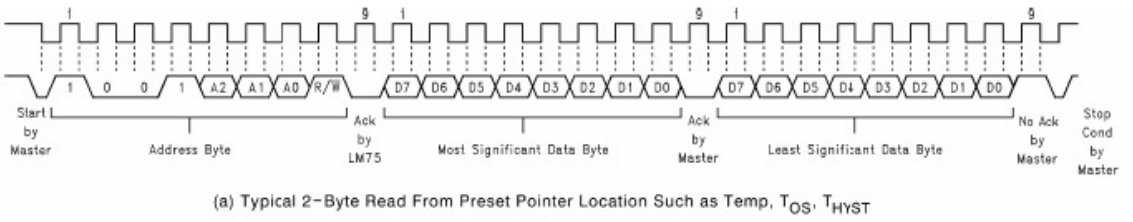

Figure 6.3. An example of data transmission on a I2C bus.

I2C is a synchronous serial bus invented in 1982 by Phillips Semiconductor. It is widely used for communication with low-speed peripherals, or when the number of communication lines is constrained. The data is transferred serially on the SDA line, one bit at a time, synchronized by the SCL clock signal. The I2C is a real multi-master bus, supporting bus arbitration and clock synchronization. However, since only two devices will communicate on the bus (DE2 and TRDB-D5M), a single master mode is used. Thus, clock synchronization and bus arbitration are not used, consequently simplifying the controller design.[9]

As stated, the SDA carries the data sent in byte chunks, and each byte has to be acknowledged by the receiver. Data transmission, consisting of multiple bytes, are delimited by a start bit and a stop bit. The SDA line is pulled up by a resistor (residing on TRDB-D5M board), and both the sensors and the FPGA can pull the data line low. For SDA communication a tri-state buffer is used, only available in IOEs (I/O elements) on the periphery of the FPGA. In the next paragraphs, if SDA is said to be HIGH, that means the bus is not driven.

A start bit, defined as the HIGH-to-LOW transition of the SDA line while the clock line is HIGH, signals a data transmission start. Similarly, a stop bit ends a data burst, defined as a LOW-to-HIGH transition of the SDA line while the clock line is HIGH. During a regular transfer, data on the SDA line must be stable while the clock period is HIGH. Changing the data line is only permitted when the SCLK is LOW. After sending 8 bits, the transmitter releases the data line, and the receiver indicates successful receiving of byte by pulling the data line LOW during the acknowledge clock pulse. During reads, the master acts as a receiver, and slave is the transmitter, while a write switches the roles.

A bus is in an idle state when both data and clock line is HIGH. The master takes control of a bus by issuing a start bit and releases the bus with a stop bit. The master is in charge of generating a clock and start/stop conditions.

After a starting bit, the master sends the first 8 bits, the 7-bit slave address followed by a write/read selection bit. In the case of TRDB-D5M, the read address is 0xBB, and a write address is 0xBA. Afther that, the master sends an 8-bit configuration register address, specifying where to store the transmitted data. Next, two bytes of data are transmitted, as the width of sensor configuration registers is 16 bits. If writing, the master is in charge of transmitting the data, and the slave acknowledges them. If reading, the slave transmits the data, and the master has to acknowledge after each byte.

After the two bytes are sent, the master can end the burst by issuing a stop bit, or instantly begin next one by sending a start bit. During the read, if a slave does not receive a stop bit, the address is auto-incremented, and contents of the next register are sent.

The library implementation of I2C controller is write-only and exposes its functionality using these signals (all active high, registered on the rising edge of the clock):

- *enableStrobeIn* - start a data transfer
- *dataIn* - register data to be sent
- *devAddrIn* - address of I2C device with read/write bit set
- *dataAddrIn* - sensor register address
- *doneStrobeOut* - signalizes end of data transfer
- *errorStrobeOut* - signalizes error during a data transfer
- *sClkOut* - SCL wire output
- *sDataIo* - SDA wire output

Internally, a state machine manages the operations and controls data output. The SCL clock signal is generated using a simple clock division technique, a counter. As the I2C protocol uses relatively low clock frequencies, the clock skew and jitter does not worsen the data transmission.

Implementation correctness is checked against an I2C slave model. The automated testbench is inmplemented in a *i2c_ctrl_tb.vhd* file.

6.4.2 CCD configuration description & implementation

While I2C controllers' purpose is to transmit a single data burst, setting up the sensor requires us to set several registers according to library settings. During the development and testing, the author settled on these parameters:[7]

- Image Width = 644 pixels [addr = 0x04]
- Image Height = 484 pixels [addr = 0x03]

  After reset, the sensor immediately starts outputting pixels in default resolution, that is, using full active array (2592x1944), a lot of data to store, transfer, and process. To illustrate, storing a full pixel array requires 15MB (2592
- 1944
- 8b
- 3 colors) of storage capacity. As the greatest storage available on the DE2 board is SDRAM (8 MB), using the full resolution is not possible. Furthermore, to avoid edge artifacts during demosaicing and convolution, we add one extra edge pixel for both, thus arriving at a final resolution of 644 by 488 pixels. ADD NOTE THAT THEY MUST BE ODD EVEN
- Width start = 990 [addr = 0x02]
- Height start = 838 [addr = 0x01]

  These parameters set the location (zero-based) of image viewport, and we set them to be roughly in the center of the sensor array.
- Row Mirror = false [addr = 0x20]
- Col Mirror = false [addr = 0x20]

  Optionally, the row and column mirroring can selectively or together, so the image from the sensor is flipped horizontally or vertically, respectively.
- Horizontal Blank = 1023 [addr = 0x05]
- Vertical Blank = 130 [addr = 0x06]

  During a horizontal or vertical blank, the sensor does not output valid pixels. Frame time and exposure are also controlled using these values. In the current implementation, blanking values are computed to keep the frame time at 33ms, thus outputting 30 frames per second.
- Test Pattern = 0x00 [addr = 0xA0]

  Instead of the captured image, a test pattern can be output from a sensor for debugging purposes. Several test patterns are available, some of them with config-

urable parameters. Black level value can be set independently for each color if the test pattern is enabled.

Other than these explicit parameters, modifiable by changing constant definitions. A row and column pixel skipping and binning modes are disabled. In the case of pixel binning, multiple Bayer pixel quadruplets can be summed and averaged internally by the sensor, to produce a brighter image with less noise, albeit with reduced resolution. By using pixel skipping the same reduced resolution can be achieved, but the skipped pixels are discarded and do not participate in the resulting image.

The module implemented in file *i2c_ccd_config.vhd* uses the I2C controller to send the configured parameters to the sensor after startup. Its design is simple and uses a state machine to keep track of parameters array pointer and I2C controller state. After the configuration is sent, it is ready to transmit again. The module outputs are as follows (all active high, registered on the rising edge of the clock):

- *enableStrobeIn* - starts the configuration transmission
- *configDoneStrobeOut* - signals completion of data transmission

Rest of the signals are either reset/clock inputs or I2C communication signals.

Correct operation and functionality is tested using automated testbench *i2c_ccd_config_tb.vhd*. I2C slave implementation verifies correctness of module outputs.

6.5 SDRAM

6.5.1 SDRAM description

As previously stated, CCD and VGA clocks are asynchronous. For that reason, we need to employ some method of buffering between data reads and writes. While the SRAM used in the DE2-155 board has sufficient capacity, the author was working with a DE2 board, and the library aims to be compatible with both. Thus, in the case of the DE2 board, as both SRAM and internal FPGA memory blocks have insufficient capacity, external on-board SDRAM is used as framebuffer.

SDRAM, a synchronous dynamic random-access memory, is fast volatile storage using interface coordinated by an external clock. Interestingly, DE2 board documentation contains mention of memory chip made by ISSI (IS42S16400F/IS45S16400F), whereas on the authors' board, the chip is made by Zentel (A3V64S40GTP/GBF). While the SDRAM interface is standardized, there are subtle timing differences between the mentioned memories.

Implementation of SDRAM controller, together with accompanying memory model was probably the most time-consuming part of the library. The valuable documentation for ISSI memory module helped immensely during implementation.

The SDRAM is an evolution of DRAMs. In case of DRAMs, the commands are asynchronous, only delayed by signal travel time through the wires. The SDRAM latches control signals and data on the rising edge of the provided external clock. Furthermore, this enables command and data pipelining. While the latency stays roughly the same, throughput is increased, thus improving performance.

In case of register or SRAM memory, to store one bit of information, usually up to 6 transistors are used, whereas SDRAMs use one capacitor. The primary deficiency of capacitors is that they lose charge over time and have to be refreshed periodically. Moreover, memory reads are `destructive`, meaning data has to be written back after reading.

FUNCTIONAL BLOCK DIAGRAM

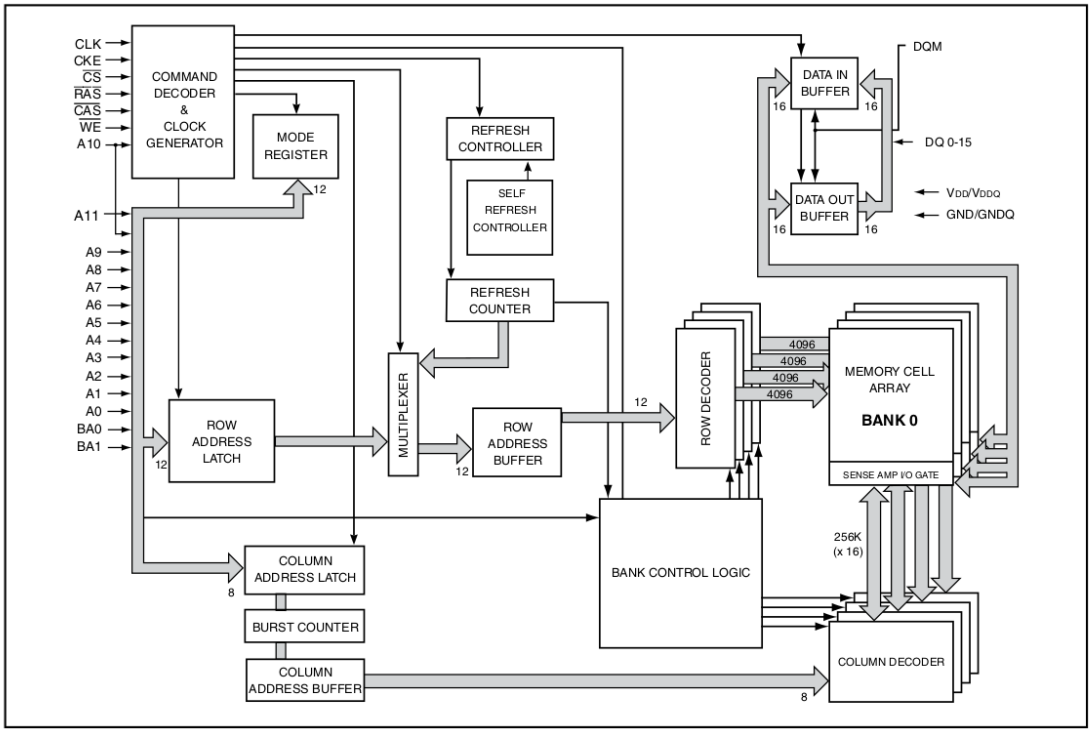

Figure 6.4. A functional diagram of SDRAM.

Internally, data is stored in 2D arrays of capacitors, divided into rows and columns. A column constitutes a smallest addressable entity, commonly containing 8, 16, or 32 bits of data, usually equal to data interface width. Each row contains a number (a power of 2) of columns. A bank comprises a memory array, supplementary row/column decoders, and data registers. It is individually addressable using a bank select signal.

PIN DESCRIPTIONS

A0-A11	Row Address Input
A0-A7	Column Address Input
BA0, BA1	Bank Select Addresses
DQ0 to DQ15	Data I/O
CLK	System Clock Input
CKE	Clock Enable
$\overline{CS}$	Chip Select
$\overline{RAS}$	Row Address Strobe Command
$\overline{CAS}$	Column Address Strobe Command

$\overline{WE}$	Write Enable
LDQM, UDQM	x16 Input/Output Mask
V_{DD}	Power
GND	Ground
V_{DDQ}	Power Supply for I/O Pin
GNDQ	Ground for I/O Pin
NC	No Connection

Figure 6.5. An SDRAM pinout.

- *A0 - A11* - address inputs
- *BA0, BA1* - bank select address
- *CAS, RAS, WE* - device command input (active low)
- *CKE* - clock enable
- *CLK* - master clock input
- *CS* - chip select (active low)
- *DQ0 - DQ15* - data i/o pins
- *LDQM, UDQM* - data i/o byte lower/upper byte masking

Address inputs are used in conjunction with various commands, to select row or column to be used. The bank address is used to select to which bank the current command applies. A CAS, RAS, and WE signals encode SDRAM commands. The CKE input determines whether the CLK input is enabled. When CKE is low, the device will be in either power-down mode, clock suspend mode, or self-refresh mode.

Chip select determines whether the command input is enabled. When CS is LOW, the decoding is enabled and disabled when CS is HIGH.

TRUTH TABLE – COMMANDS AND DQM OPERATION[1]

FUNCTION	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	DQM	ADDR	DQs
COMMAND INHIBIT (NOP)	H	X	X	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X	X	X
ACTIVE (Select bank and activate row)[3]	L	L	H	H	X	Bank/Row	X
READ (Select bank/column, start READ burst)[4]	L	H	L	H	L/H[8]	Bank/Col	X
WRITE (Select bank/column, start WRITE burst)[4]	L	H	L	L	L/H[8]	Bank/Col	Valid
BURST TERMINATE	L	H	H	L	X	X	Active
PRECHARGE (Deactivate row in bank or banks)[5]	L	L	H	L	X	Code	X
AUTO REFRESH or SELF REFRESH[6,7] (Enter self refresh mode)	L	L	L	H	X	X	X
LOAD MODE REGISTER[2]	L	L	L	L	X	Op-Code	X
Write Enable/Output Enable[8]	—	—	—	—	L	—	Active
Write Inhibit/Output High-Z[8]	—	—	—	—	H	—	High-Z

Figure 6.6. SDRAM commands encoding and descriptions.

COMMAND DESCRIPTIONS:

- NOP (No Operation)

  A No Operation command prevents unwanted commands from registering during idle, or wait states.
- Active

  Data in a column cannot be accessed directly. The selected row must first be `Activated,` its capacitors sensed and amplified, latching the decoded bits into a register. When the command is used, the bank address inputs select the row, and regular address inputs select the row to be activated. Until the row is Precharged, it stays open for further data accesses.
- Precharge

  If we want to access another row, the current one must be first Precharged, storing any changes back into capacitors. The bank address selects a target bank. Alternatively, all banks can be simultaneously Precharged by setting A10 (address pin) HIGH. After a row is Precharged and timing is met (tRP - row precharge time), it is in Idle state and must be Activated before starting a burst.
- Write

  Starts a write burst. The address inputs select starting rows, while BA0 and BA1 signals pick the bank. A burst can be optionally Precharged at the end of the burst by automatically if A10 is HIGH. Data is sampled immediately starting at the clock cycle when the command is registered. The memory controller automatically increments column address, and data are sampled until a configured burst length is attained unless the full-page burst is configured. In that case, the writes continue indefinitely, wrapping-around end of the row, until a Burst Terminate command is issued. DQML and DQMH inputs can be used to mask lower or upper data byte, respectively. If the corresponding DQM signal is low, data will be written to memory.

Else if the signal is HIGH, the corresponding data byte will be ignored, and the byte at the current address will remain without change.

- Read

  Starts a read burst, starting from column selected using address inputs, in a bank selected using bank address lines. Same as in case of a write, the burst can be optionally Precharged at the end by setting the A10 line high. Data output is subject to DQM inputs registered two clocks earlier. First valid data is output after a tCAS (column address strobe) delay configured in the mode register.
- Burst Terminate

  Issuing a Burst Terminate command terminates current burst, truncating it. It acts as a NOP if memory is currently not bursting. Note, that a burst with auto precharge enabled cannot be terminated.
- Auto/Self Refresh

  A row is refreshed using the Auto Refresh command. The row to be currently refreshed is generated internally by the memory. For correct functionality, the Refresh command should be issued at least 4096 (number of rows) times every 64 ms (tREF).
- Load Mode Register

  A Load Mode Register command is used to change the memory configuration. The memory data input should contain the new contents of the configuration register. These parameters are configurable using the mode register:

  - Burst Length (1, 2, 4, 8, full page)
  - Burst Type (Sequential, Interleaved)
  - Latency Mode (2, 3)
  - Write Burst Mode (Programmed Burst Length, Single Location Access)

  The tRCD delay denotes a delay between issuing Active command and starting a read/write. After a row is activated and the timing is met, a burst using a selected row can begin.

6.5.2 Controller implementation & description

At first, the author thought, that an SDRAM controller implementation taken from an open-source PoC (Pile of Cores) library would be sufficient, that was not the case. The controller is a generic implementation that has to cover a wide range of use-cases. That means, the memory operations are abstracted away, and most importantly, refreshes are unpredictable and non-deterministic. That could be a problem since a FIFO bubble can de-synchronize the whole pipeline.

The library implementation of the SDRAM controller aims to provide predictable, static scheduling suited for a framebuffer operation. Given that the data traffic patterns are known beforehand, the design can be thus statically optimized. In our case, as the pixels are read and written serially, the data bursts are long and using consecutive addresses. We can exploit this fact by using prefetching, memory bank parallelism, by executing the commands in a pipeline as soon as possible, and by using longest burst lengths possible. While it is true that the SDRAM is randomly accessible, the accesses to differing rows in a bank are penalized by a long delay. Thus, the implemented controller only writes and reads whole rows at a time (256 columns) conforming to the predicted data patterns.

The address encoding is selected so that the consecutive addresses are using different banks, thus exploiting the memory parallelism. The lower two bits encode a bank, while the rest of the address selects a row. If the cmdReadyOut output is true, the controller is ready to execute commands. Available commands are: Write,

Read, and Refresh. When a valid command is issued, the first has to find out how to carry it out. That is the purpose of a plan generator. It takes into account the bank information, and the requested operation, generating a plan, that when executed, puts the memory into the desired state.

Next, the generated plan is scheduled. The scheduler makes sure that the execution of the plan does not violate memory timing requirements, while still being as tight as possible. After that, the controller transitions to an 'ExecutePlan' state, in which the scheduled plan is finally executed. If the execution started a burst, the controller transitions to a 'Burst' state, else it goes back to the 'Idle' state.

While in a Burst state, the controller tries to predict the next address, and Activate it if possible. To guess the following address, we keep track of the number of consecutive bursts of the same type and the last used address. The burst lengths are configurable using a generic parameter, and so is the highest used address, after which, the address wraps around. There are a few cases when the next predicted address cannot row/bank combination cannot be Activated, e.g.:

- it is currently in a burst
- it clashes with the predicted address of another operation

In the first case, the bank is scheduled to be Precharged at the end of the current burst, that is, if not explicitly overridden by a new command. In the second case, a simple technique is used to choose which row gets Activated. Since we keep track of how many consecutive operations of the same kind got executed, we can use this information to activate the address belonging to the operation that is guessed to be requested next. Finally, the controller waits for the burst to end, and then transitions to Idle state, where it awaits next command.

Inputs and outputs description:

- addrIn - the address for the command
- cmdIn - the command to be executed
- dataIn - data input used during writing
- cmdReadyOut - true, if controller is ready to execute commands
- provideNewDataOut - if true, provide new data on the next clock
- newDataOut - if true, valid data are ready to be read in this cycle
- dataOut - memory data output
- memInitializedIn - a signal from mem initialization controller
- memOut - a group of memory output signals
- memDataOut - data to be output to memory
- memDataIn - data output from memory
- memDataOutputEnableOut - controls a tri-state buffer output

As the design is pipelined, the signals provideNewDataOut and newDataOut can briefly overlap. Furthermore, because the data bus is bi-directional, a memDataOutputEnableOut signal is used to control whether the port is driven, or is in high impedance state.

An automated testbench (*sdram_ctrl_tb.vhd*) verifies the functionality of the controller with help from the SDRAM model. The model checks if the timing is correct, and if the command sequence is logically sound. The test uses three types of address generation: same address, consecutive address, random address. The command execution plan is designed to permute through all possible command combinations. Finally, a range of command delays is tested to catch bugs arising during transient states (e.g., when two bursts overlap).

6.5.3 Init controller implementation

The role of an SDRAM initialization controller is to carry out proper initialization routine, as defined in the documentation, and bring the memory to an idle state. While its implementation could be fused with a burst controller, the author did not want to make it more complicated. The documentation contains the exact initialization routine definition, but it consists of roughly these steps:[10]

1. Wait for stable power
2. Start the clock and drive CKE (clock enable) signal HIGH
3. After stable power and clock, wait for 200us
4. Issue Precharge All command
5. After the tRP delay use at least two Auto Refresh commands
6. Use the Load Mode Register to set the mode register contents
7. Assert at least one NOP command after the Load Mode Register is issued (to meet the timing requirements).

The implementation is relatively straightforward, as it only uses a counter and a state machine to meet the timing and to sequence the commands.

The implemented interface is as follows:

- Generic parameters:
  - *MODE_REG* - the data used to set the mode register
  - *INIT_DELAY_CYCLES* - configurable initialization delay
- Signals:
  - *clkStableIn* - input from PLL (phase-locked loop) signalling clock stability
  - *memInitializedOut* - true, if the memory is ready to be used
  - *memOut* - a group of memory outputs
  - *memDataOut* - memory data output
  - *memDataOutputEnable* - signals whether to drive the data bus

An SDRAM model checks the implementation using an automated testbench, implemented in *sdram_init_ctrl_tb.vhd*.

6.5.4 Model description & implementation

During the library design phase, a need to verify the SDRAM controller functionality arose. While the author tried to find a suitable, free model implementation, the only ones found were provided by Micron. Nevertheless, they do not support full-page burst lengths, do not check refresh timings, and lack proper initialization checks. Furthermore, the models are written using an old version of VHDL, are not very informative, do not support mocking of the memory contents for testing, and the author believes that the code style is unmaintainable (too much code repetition). So, the goals of the SDRAM model implementation were:

- to explicitly convey the memory state
- to be able to define the memory contents programmatically from the testbench
- to provide more maintainable and understandable code
- to implement the model in a way that is analogous to the inner implementation of the SDRAM

Two processes are mainly responsible for the functionality. The bankProc process is responsible for timing and logical checks concerning bank states and transitions.

It schedules and executes state changes, additionally exposing the information to the mainCtrl process. It is analogous to a bank control logic inside the SDRAM. If enabled, the debugging logs are printed out during runtime. Next, the mainProc is in charge of bursts and mode register loads.

For now, the model supports only the sequential bursts; moreover, only the full-page burst length has been tested extensively. The simulated memory array is implemented using a MemoryPkg from OSVVM library. The advantage of using the library is that it enables us to dump/load the memory contents to/from a file. Finally, the model also checks if proper initialization routine was carried out and whether the rows are correctly refreshed. The author believes that other students could use the memory model to help them verify memory controllers and to understand SDRAM operation better.

Model inputs description:

- Generic parameters:
  - *LOAD_FROM_FILE* - if true, load memory contents from a file
  - *DUMP_TO_FILE* - if true, dump memory contents to a file at the end of the simulation
  - *INPUT_FILENAME* - name of the file used for memory initialization
  - *OUTPUT_FILENAME* - name of the file used for memory dumping
- Debug signals:
  - *isInitializedOut* - if true, memory has been successfully initialized
  - *simEndedIn* - stops memory operations, and optionally dumps the memory to file if set to true

6.6 VGA controller

The final part of the library is a VGA controller, implementing a basic 640x480 @ 60Hz video mode. It generates the required sync signals, keeps track of timing, and sends the pixel data to the onboard DAC (ADV7123). The VGA is a graphics standard for video display controllers defining the timing, signals, connectors and electrical characteristics used. This module provides rudimentary video output with a resolution of 640x480 pixels (width x height). Before being transmitted over the wire, the DAC converts the digital pixel values to analog voltage (0V - 5V).

The analog color data is then sent to the output along with synchronization clock and horizontal/vertical synchronization signals. Although the clock frequency required for the chosen video mode is 25.175 MHz, we use a 25 MHz clock derived from a 50 MHz input by using a PLL. The rounding of a clock is not uncommon, and modern displays should be able to sync to our rounded clock without problems.

The display timings are as follows:[11]

General timing

Screen refresh rate	60 Hz
Vertical refresh	31.46875 kHz
Pixel freq.	25.175 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	640	25.422045680238
Front porch	16	0.63555114200596
Sync pulse	96	3.8133068520357
Back porch	48	1.9066534260179
Whole line	800	31.777557100298

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	480	15.253227408143
Front porch	10	0.3177557100298
Sync pulse	2	0.063555114200596
Back porch	33	1.048659384309821
Whole frame	525	16.683217477656

The image color values are output serially, row after row, synchronized using a pixel clock, and periods of horizontal and vertical blanking surround the valid image frame. During blanking, pixels must be kept dark (0x00). The horizontal and vertical sync signals are active LOW and must be asserted during a line of frame synchronization periods.

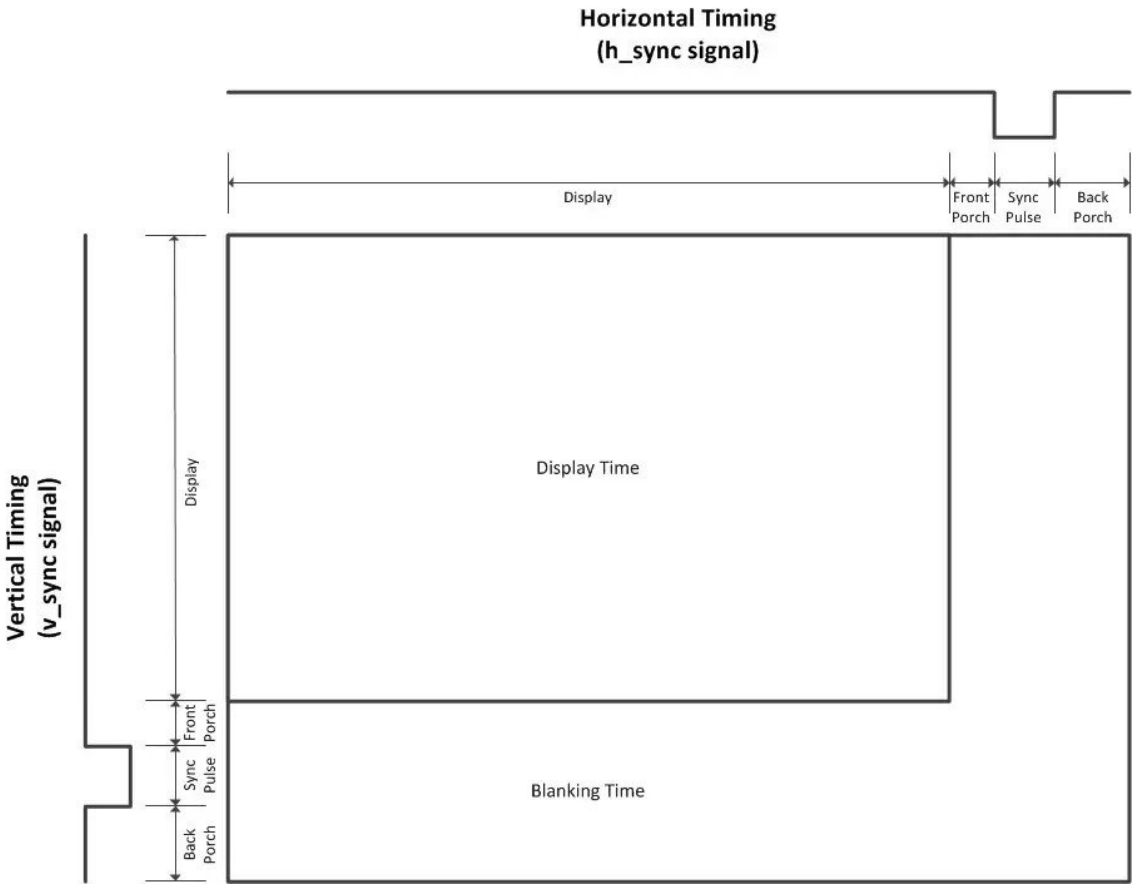

Figure 6.8. VGA blanking diagram.

]

Internally, the outputs are derived using the height and width counters. As the DAC latches digital pixel data on the rising edge of the clock, the synchronization signals are registered, and thus delay by a one clock cycle, to compensate. The DAC additionally uses two more signals, sync, and a blank (both active LOW). They can be used to synchronize a frame without using separate horizontal and vertical sync signals, by manipulating a green color value instead. This functionality is unused, and signal values are set to avoid affecting the output. Signals definitions:

- *pixelDataIn* - pixel data input
- *nextPixelOut* - if true, provide new pixel in the next clock cycle
- *blankNegOut* - a blank DAC output signal
- *syncNegOut* - a sync DAC output signal
- *pixelDataOut* - pixel data output to DAC
- *vSyncNegOut* - VGA vertical synchronization output
- *hSyncNegOut* - VGA horizontal synchronization output

The testbench is located in a file named *vga_ctrl_tb.vhd*. Interestingly, while the controller implementation is comparatively short, containing just 54 lines of code. Whereas a test spans 204 lines.

Chapter 7
Conclusion & closing remarks

In the end, the library is implemented, but not without delays, and it never got tested in hardware. Still, the author believes, that the amount of testing and verification is a good indicator of correctness, and hopes the implementations will be used in a practical design or as a teaching aid.

The first weeks and months were hard, as the author has previously only worked with VHDL briefly and without a more profound knowledge of digital design. However, as the weeks passed, the author started to understand the language, simulation, and the hardware used better. The author must say that, interestingly, the most exciting part of the thesis was an SDRAM and the controller implementation.

Appendix A
Specification

CTU CZECH TECHNICAL UNIVERSITY IN PRAGUE

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Čižmár Martin** Personal ID number: **456988**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

VHDL Teaching Module for TRDB_D5M Camera

Bachelor's thesis title in Czech:

Výukový VHDL modul pro TRDB_D5M kameru

Guidelines:

1. Study the DE2 and DE2-118 development boards and the TRDB_D5M camera module.
2. Create a VHDL library to capture an image at the FPGA hardware level.
3. Design a camera image processing task that LSP students can solve with your library.
4. Create instructions.

Bibliography / sources:

[1] TRDB-D5M, Hardware Specification, Terasic 2009
[2] DE2 a DE2-115 Manual, Terasic 2009

Name and workplace of bachelor's thesis supervisor:

Ing. Richard Šusta, Ph.D., Department of Control Engineering, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **21.02.2019** Deadline for bachelor thesis submission: **14.08.2019**

Assignment valid until: **30.09.2020**

Ing. Richard Šusta, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt

Student's signature

References

[1] *GHDL.*
https://github.com/ghdl/ghdl.

[2] *GTKWave.*
http://gtkwave.sourceforge.net/.

[3] *Sigasi.*
https://www.sigasi.com/.

[4]
. *Cyclone II Device Handbook, Volume 1.*
.

[5]
. *DE2 Development and Education Board, User Manual.*
.

[6]
. *DE2-115 User Manual.*
.

[7]
. *TRDB-D5M, Terasic TRDB-D5M Hardware specification.*
.

[8] *Basics of convolution.*
http://aishack.in/tutorials/convolutions/.

[9]
. *I2C MANUAL.*
.
https://www.nxp.com/docs/en/application-note/AN10216.pdf.

[10]
. *64Mb Synchronous DRAM Specification, A3V64S40GTP/GBF.*

[11] *VGA Timings.*
http://martin.hinner.info/vga/timing.html.