POLITECNICO DI TORINO

Electronics for Embedded Systems

Project of the course

Professor: Claudio Passerone

Data: November 18, 2019

Author: Alessandro Salvato

Contents

1 Introduction

I'm very glad to present by this report, my own project for Electronics for Embedded Systems course. The main idea consists in realizing a camera able to capture photos to be transferred to an host computer for further processing. Before describing in details the environmental structure, I guess it's fundamental to highlight that all the topics of the course have been covered in the following manner:

- **Memory**: memory controller as FSM
- **Programmable logic device**: acts as remote to send the user command
- **Interconnection protocol**: I2C, SCCB and UART
- **Peripheral management**: programming of camera peripheral
- **AD and DA converters**: usage of AD converter to sample the luminance of the environment
- **Power management**: Step-down converter to drive a LED

Each of them owns a proper section below. Let's have a look on the general architecture, listing the main parts of the system. The camera is connected to an ST microcontroller: a STM32F446ZET. The choice on this MCU has due to the fact that it integrates a very useful and widespread engaged peripheral in the video-capturing field: the Digital CaMera Interface (**DCMI**). All images are sent to the host computer by UART protocol, exploiting another embedded peripheral of the microcontroller. Computer reads UART's data plugging a USB-UART adapter. Then a small Python script converts those data into a BMP image. The command is generated by the user pushing a specific button. An 8 buttons keyboard is tied up to the FPGA: an Altera Cyclone IV on DE0-Nano board. The implemented VHDL is composed of the keyboard driver and a fully structural UART peripheral. When a button has clicked, an encoded 8-bit data is sent to the microcontroller, which, by an interrupt, recognizes that the user pushed something. A breadboard has been used to build circuits for AD and Bulk converters. I used STM32 CubeIDE to program the microcontroller , a software Eclipse based. To program the FPGA I exploited both Quartus II when uploading the *sof* file and Xilinx ISE Design Suite when simulating (just for my high familiarity with that). Moreover, as classical laboratory instruments I relied on an oscilloscope provided by GW Instek and a multimeter for resistor measurements. Moreover, I exploited the functionalities offered by GitHub, more specifically the desktop version that I found very easy and immediate to use.

For sure, the following image is more clarifying for the reader.

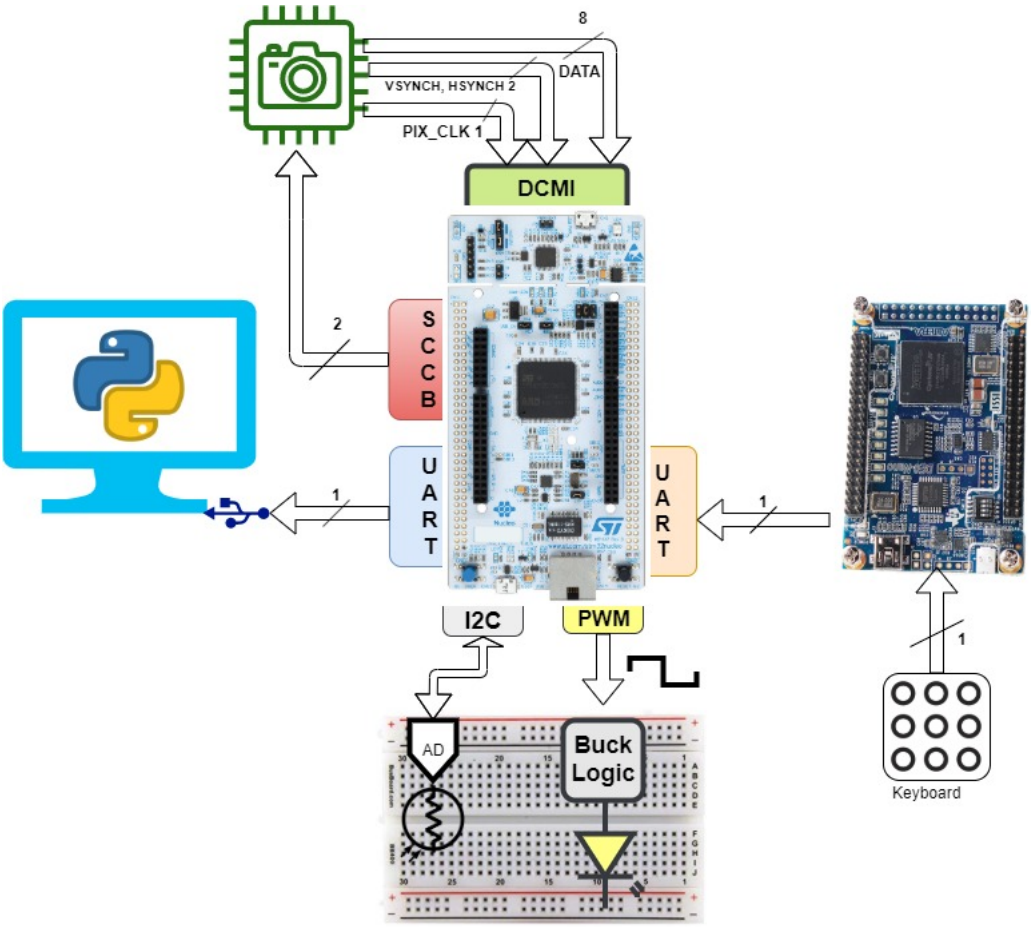

Figure 1: General block schema

Camera has been set to capture images in format QVGA (320x240) RGB 565 for an amount of 153600 bytes. However due to the poorness of memory availability, I capture the half of them. Then the Python script generates on the host a picture 640x120 without colors.

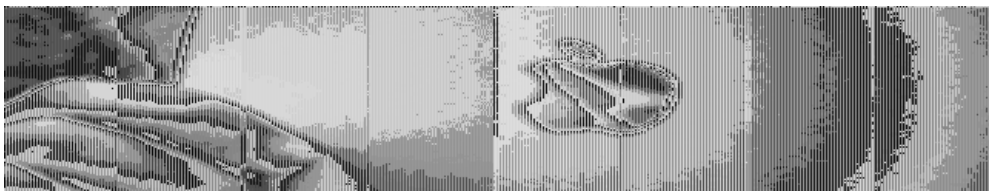

Figure 2: Apple logo captured by camera and then processed by software

Figure 3: Original Apple logo

What actually the system appears.... It's a little bit messy due to the high number of flying wires. Only the camera requires 18 links.

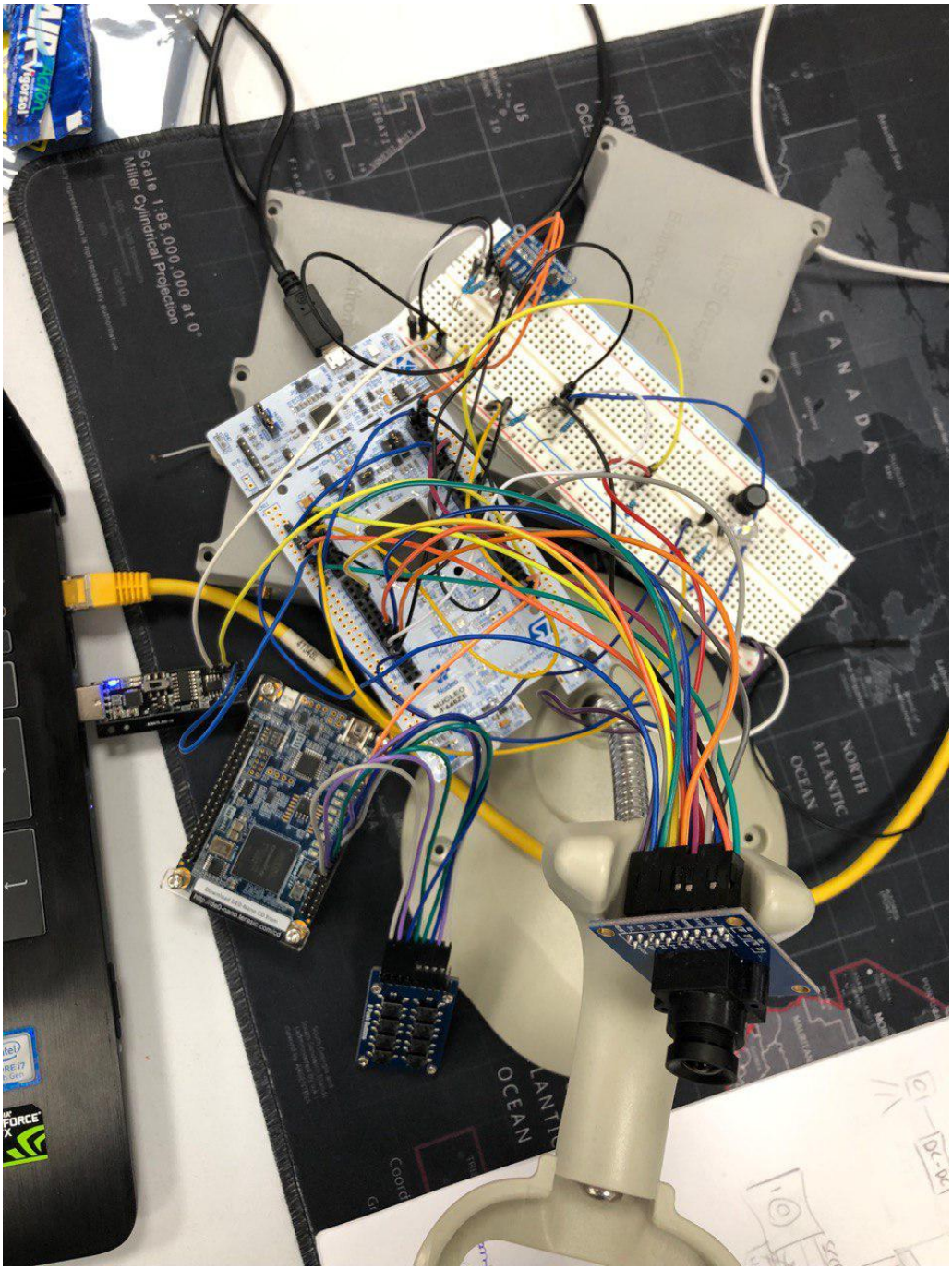

Figure 4: Real system

I'll proceed in the description in the same order I implemented each part of the project.

2 Analog to digital conversion

The analog to digital conversion aims at reading the luminance of the room, sending the sample to microcontroller. The input channel is feed by the voltage passing through a photoresistor. After having measured that voltage in condition of both full and absence of light, the dynamics has approximated at range [0.5 - 3.0] Volt.
The chip is ultra-small, low-Power, 16-Bit analog-to-digital converter with internal reference, provided by Texas Instrument at 3.50$. The microcontroller drives it by means of an I2C interface.

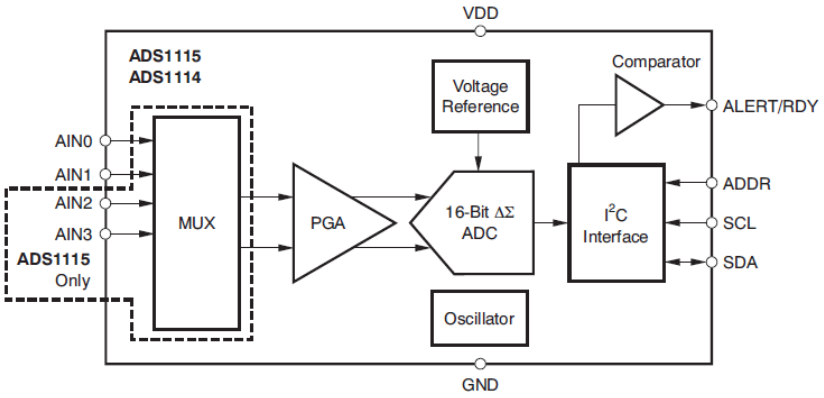

Figure 5: A2D internal architecture, from datasheet

The chip operates either in continuous conversion mode or a single-shot mode that automatically powers down after a conversion and greatly reduces current consumption during idle periods. I decided to work in the single shot one, sending the command directly from the microcontroller.

In general, to write and read 4 bytes must be sent, the string is composed of the **slave address** (0x48), the **register address**, and then in the big-endian format 2 bytes of data. Due to the fact I choose to operate in single shot mode, I have to send every time the command, so the protocol gets as follows: write 4 bytes (including the command), write the slave address and the register address containing the converted data and read 2 byte from it.

Bit(s) name	**Description**
Command	Begin conversion (automatically cleared when completed)
Input multiplexer configuration	Channel P: In0, Channel N: In1 (physically grounded on breadboard)
Programmable gain amplifier configuration	$\pm 2.048V$
Device operating mode	Power-down single shot mode
Data rate	8 sample per second
Comparator mode	not used
Comparator polarity	not used
Latching comparator	not used

Table 1: A2D configuration

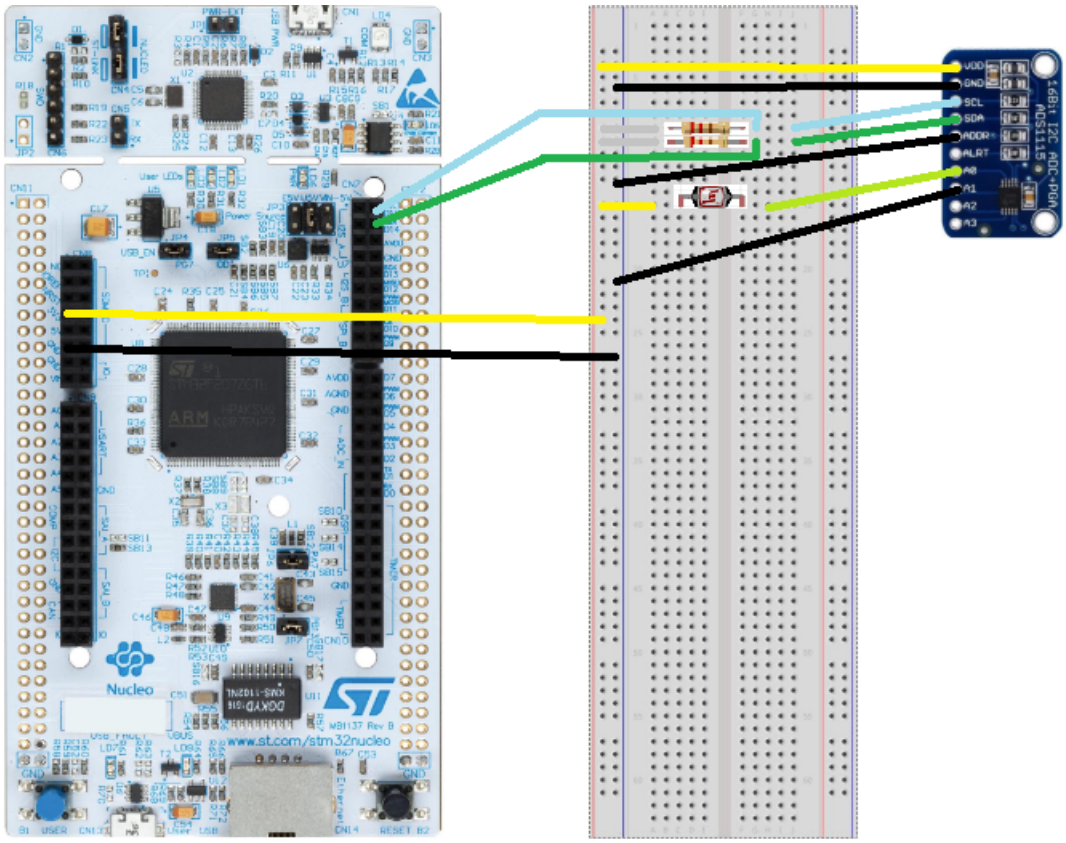

Figure 6: Connection between MCU and A2D

GPIO	Morpho Connector	Description
PB8	D15	I2C1_SCL
PB9	D14	I2C1_SDA

Table 2: A2D: GPIOs involved

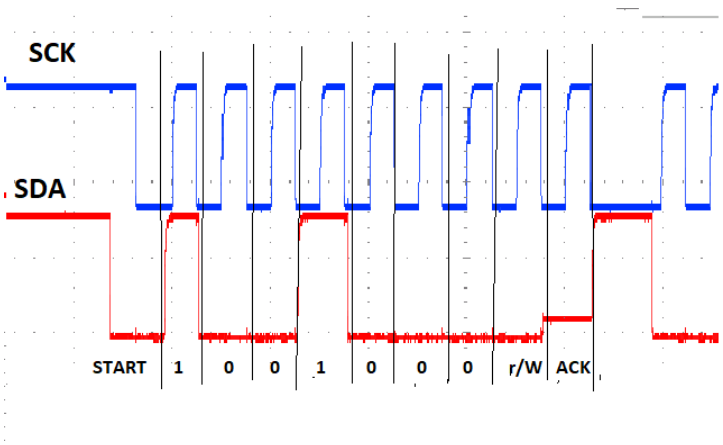

Figure 7: Starting an interrogation sending as first byte the slave address of A2D (0x48)

3 Peripheral programming

The camera is composed of 18 pins, where 2 are dedicated to Vdd and Gnd, 2 are part of the serial protocol called **Serial Camera Control Bus**, 11 are connected to MCU's DCMI peripheral for image acquiring and synchronization, 1 pin is an input for the clock source and the last 2 are for resetting and enabling. Before describing what I did, I would like to spend few words about SCCB and DCMI.

Substantially, SCCB is almost identical to I2C. The only difference regards the role of the ACK bit. This protocol doesn't care it, anyway it's generated low as well. However, from the microcontroller side, I enabled another I2C peripheral, another one than whose interfaces with A2D.

DCMI is a kind of protocol widespread used in multimedia processing. Although it allows also an higher parallelism, in terms of data bits, the camera chip, provided by OmniVision, has constrained at 8, labeled from $D0$ to $D7$. After having configured, the acquisition of the image works as follows. PXCLK is used as internal synchronizator derived from the XCLK, i.e. the system clock generated by a GPIO of the STMF446. VSYNCH and HREF pins are part of DCMI interface as well. VSYNCH triggers the start and the stop of frame capturing, HREF signal indicates the start/end of a line.

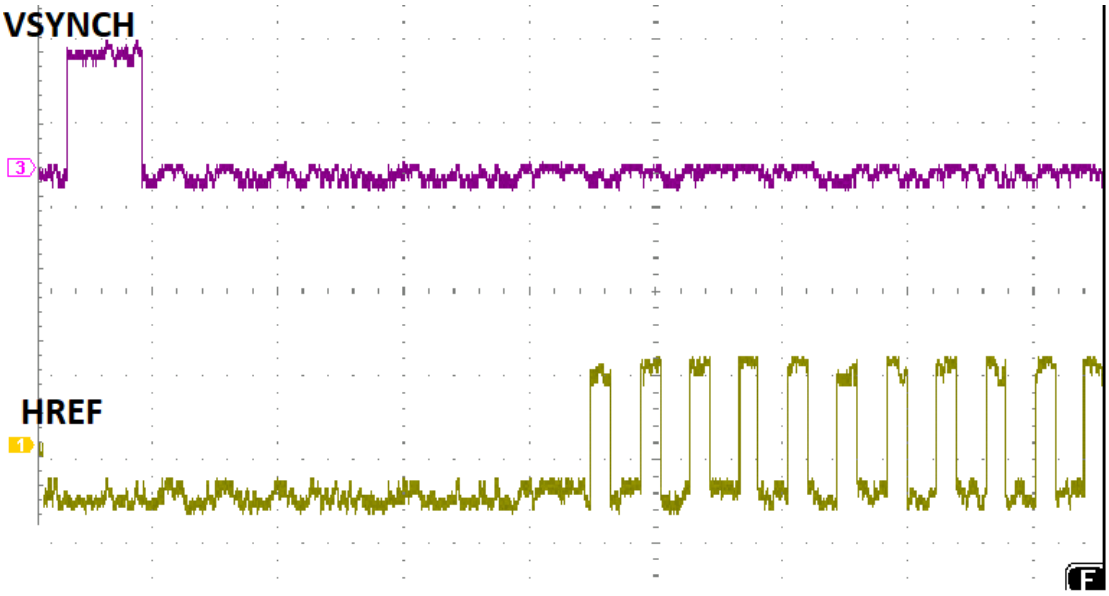

Figure 8: Start of frame capturing (top) and sampling line by line (bottom)

I guess that another relevant aspect of DCMI is how data are then transferred within microcontroller. This peripheral embeds and works as default by DMA. In fact, when setting DCMI, any option related to DMA's disable cannot be modified. So, in the firmware, I had to declare an array 76800 bytes long, to satisfy image sizing. If I hadn't been constrained by RAM capacity (128Kbyte), I would have used a 153000 long one.

I programmed the camera in order to work in single snap shot. The command is user generated on the FPGA side, which sends the command via UART to microcontroller. That UART generates an interrupt when start working; the interrupt launches the start capturing function. When the image transfer has done, I turn the camera off and then I drive the array to the host computer via another UART peripheral of the microcontroller.

The camera embeds a register configuration file of 200 locations around. Not all of them are required to be written to make it work. However I found an example online, where the main ones are highlighted with values to be loaded in them. I modified some of them to make the application compliant with my system. I'm going to list some important configuration settings.

Bit(s) name	Description
(COM7) Color Bar	Enable
(COM7) Output format	RGB and QVGA
(MVFP) Mirror	Normal Image Acquisition (No mirroring)
(CLKRC) Clock prescaler	Prescaler disabled for PXCLK
(COM16) Threshold and de-noise auto adjustment	Both enabled

Table 3: Camera configuration

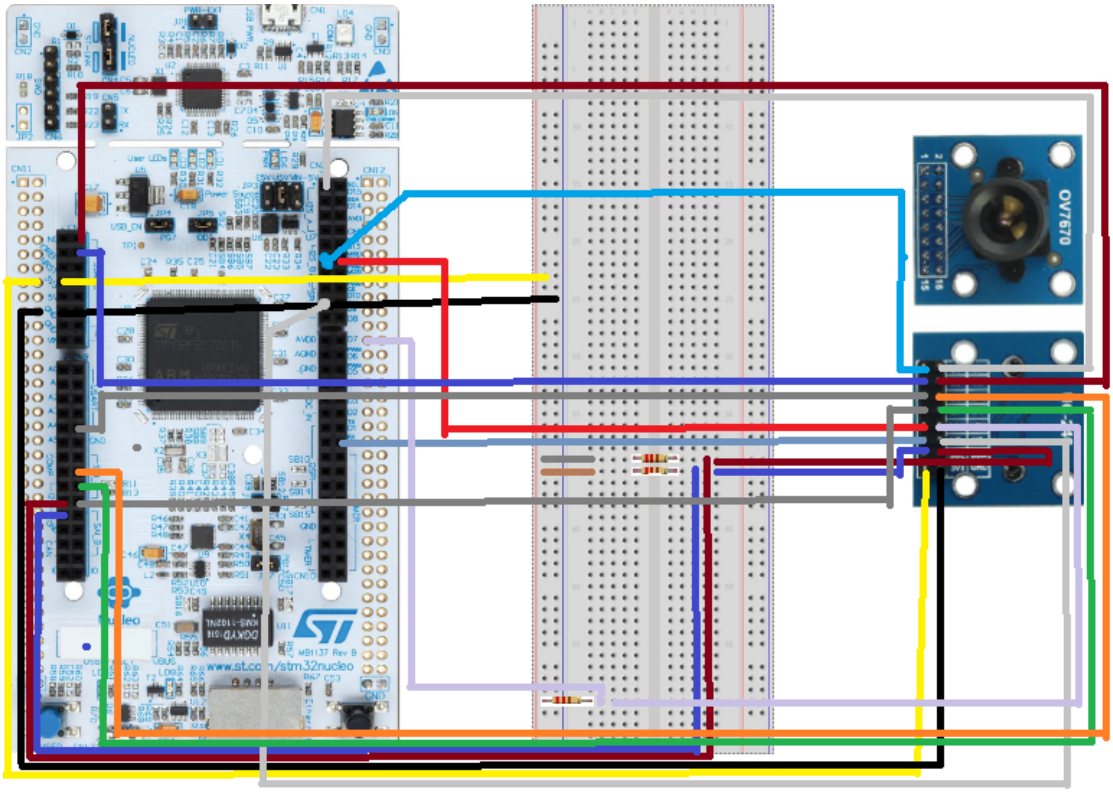

Figure 9: Connection between MCU and camera

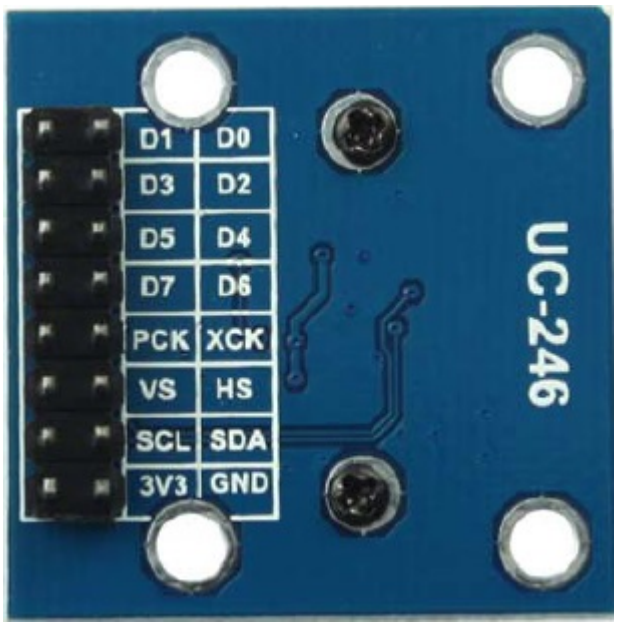

Figure 10: Camera's bottom side

GPIO	Morpho Connector	Description
PF0	D68	I2C2_SDA
PF1	D69	I2C2_SCK
PA8		RCC_MCO1 (25 Mhz) -> XCLK
PA4	D24	DCMI_HSYNC
PG9	D0	DCMI_VSYNC
PA6	D12	DCMI_PXCLK
PC6	D16	DCMI_D0
PC7	D21	DCMI_D1
PC8	D43	DCMI_D2
PC9	D44	DCMI_D3
PE4	D57	DCMI_D4
PD3	D55	DCMI_D5
PE5	D58	DCMI_D6
PE6	D59	DCMI_D7

Table 4: Camera: GPIOs involved

4 Power management

Any commercial camera integrates a flash light, which is turned on in leakage of adequate luminance. I have tried to emulate that behaviour using an high emittance LED. I choose a component able to emit a very cold light: it should be fed with 100 mA at 3V. Having defined the load, then I had to chose the input voltage between two possibilities: 3.3 and 5 volts, both of them directly generated by microcontroller and accessible by means of its own morpho connectors. I thought that the former was too close to the desired output, making the implementation of a converter totally useless. So I decided for 5V as input. Due to the fact that I need of a lower voltage, the straightforward architecture is the Bulk one, whose characteristic is

$$V_{out} = \frac{T_{on} V_{in}}{T_{on} + T_{off}} = \delta V_{in}$$

Another, maybe the most important, feature of a switching regulator is the duty cycle used to drive the NPN transistor's base terminal. First of all I adopted a reasonable period $T = T_{on} + T_{off}$. One of the timers on microcontroller is involved in generating the square wave. It's connected to the Advanced Peripheral Bus (APB) which is synchronized with a clock of 16 Mhz. The timer prescaler has been choosen at 160, so each tick is sensed each 10 μs. Selecting a period of 100 ticks, the overall period T gets $1ms$, making any consideration about duty cycle easier. However setting duty cycle isn't so simple, something cannot be taken out from a simple equation.

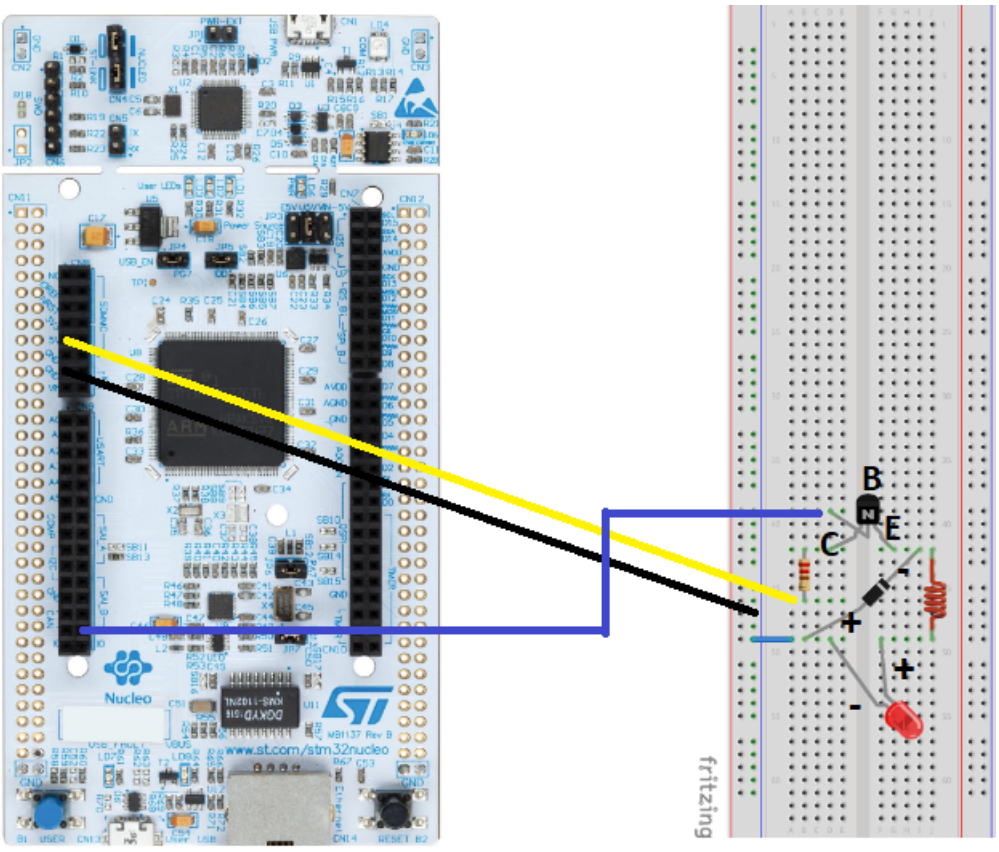

Figure 11: Connection between MCU and Buck circuit

GPIO	Morpho Connector	Description
PF9	D63	TIM14_CH1 (PWM)

Table 5: Buck: GPIOs involved

Before showing measurements captured by oscilloscope, I need to state a couple of things. I put a $1K\Omega$ serially the 5V pin in order to avoid overfeeding. It's useless but I was afraid of burning some component: MCU, A2D, camera, resistors, inductor... since all of them were connected somehow. Then, few words about the inductor: in the circuit I put $470\mu H$. But, due to the length of T_{on} ($750\mu s$) I would need a much higher one to do not enter in Discontinuos Current Mode. I choose that because it is the biggest I own and have immediate availability. However, everything works and waveforms among the nodes of the circuit seems to have a compliant shape.

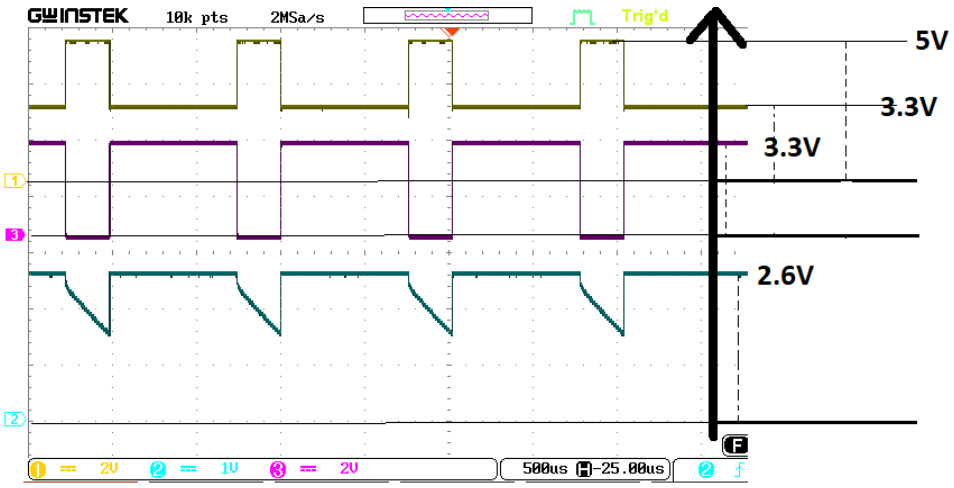

Figure 12: From top to bottom: V_{in}; $PWM signal$; V_{out}

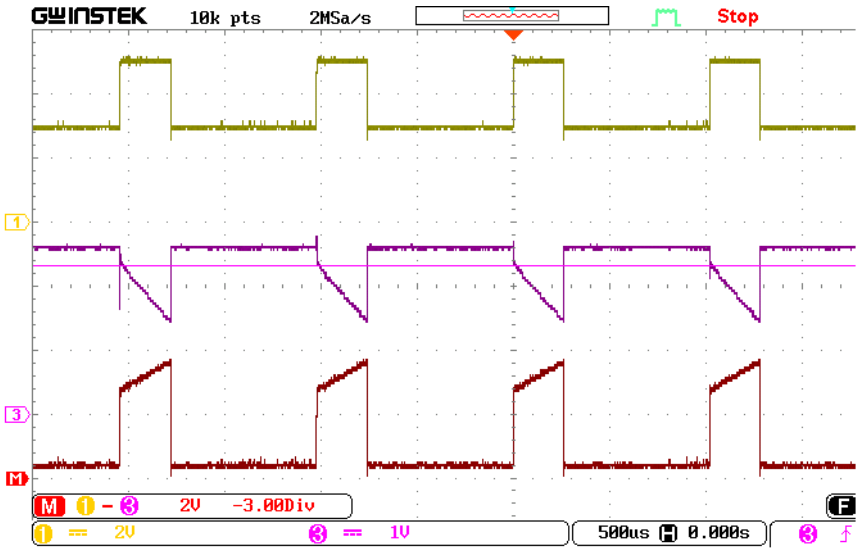

Figure 13: From top to bottom: V_{in}; V_A of the node where are emitter diode's cathode and one inductor pin; V_{out}, $V_{in} - V_A$

5 Programmable logic device

The behaviour that I tried to emulate by means of the FPGA consists in reading the status of a group of buttons and send them to the microcontroller via single wire communication protocol: UART. So, the overall architecture is composed of two main parts:

- The logic of interface for the keyboard
- UART peripheral VHDL-based (only TX channel). It will be described in the next section

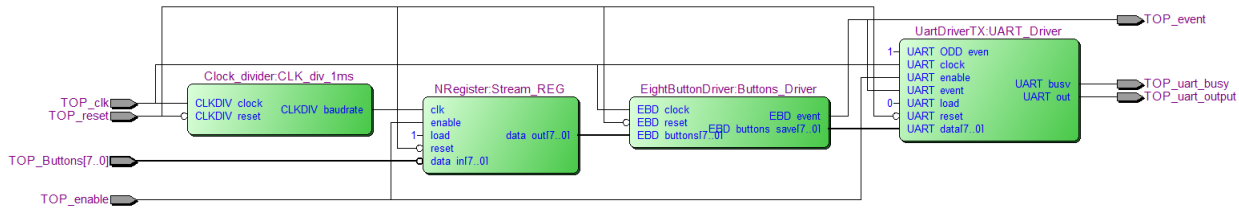

Figure 14: Top level entity view

Clock is internally generated by the FPGA at 50 MHz, however in some point of the architecture I needed to slow down. The clock feeding the output shift register within UART peripheral block runs at 115400 bit/s around. Another clock divider is used to sample buttons each 1 ms. During my tests, I noticed that the architecture worked well in simulation, but scoping the UART output with an oscilloscope probe I saw the mess. This totally uncorrect behaviour was given by the fact that button are mechanical objects generating a bouncing signal when pushed. So I measured the duration of these oscillations, leading me at the conclusion of implementing a "debouncing logic", in my project composed of a register and a clock divisor, the ones are shown in the top level entity view. The input signals got stable after some hundreds of μs, until $800\mu s$ in some cases. That's the reason why I perform input sampling each one millisecond. I'm sure that in this interval oscillations expire.

Clock dividers I implemented are both fed by the 50 Mhz clock. Internally the output of a counter is compared with the one contained within an hardwired register. The comparator performs the equality checking operation. Every time comparison gets true, the counter is reset in next cycle. The output of clock divider is the one of the comparator; in such a way the resulting waveform is a train of pulses having the desired baudrate.

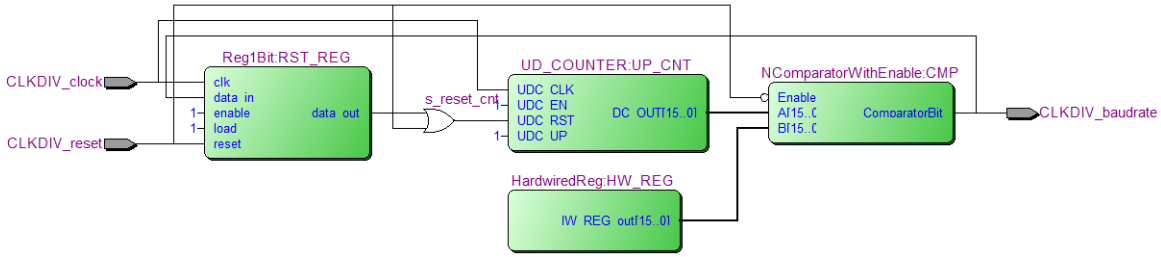

Figure 15: Clock divider architecture

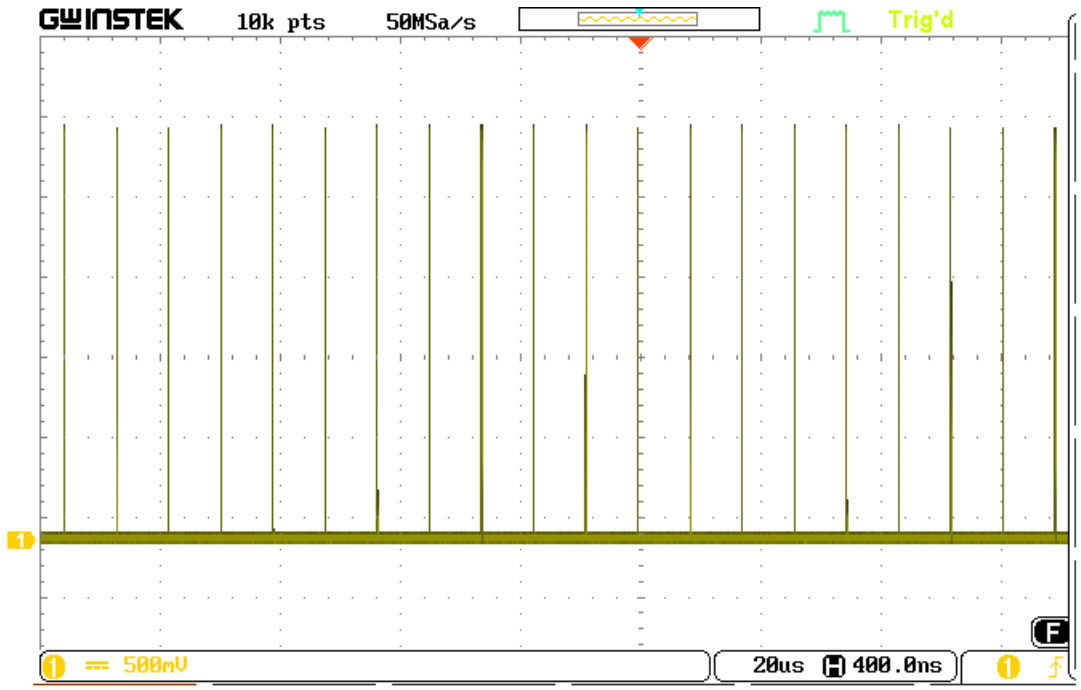

Figure 16: Clock divider output within UART peripheral. Time between two consecutive pulses is 9μs, leading a baudrate of 115200 bit/s. The hardwired value is 434, from the round division between 50Mhz and 115200 Hz

The eight buttons driver just simply recognizes when a button has been pushed and generates an "event", then sent to the UART peripheral to start the transfer. Two registers are on the output, one for buttons, one for the event bit.

I'm going to talk about UART peripheral implementation in the next section, dedicated to **Interconnection protocol**. Another VHDL component, that is not integrated here, is the memory controller for the SDRAM laid on the Altera board; the last section **Memory management** is dedicated to it and some comparisons between waveform simulation and real required access protocol will be shown.

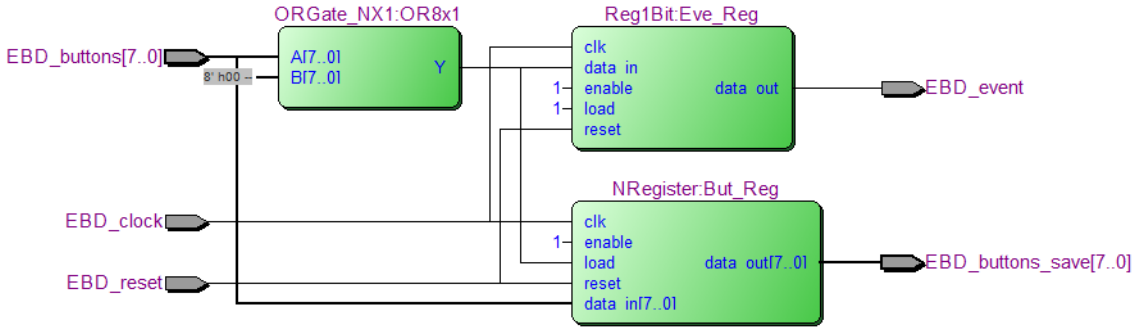

Figure 17: Eight button driver architecture

Node name	Direction	Location
TOP_reset	Input	J15 (Push Button[0])
TOP_enable	Input	M1 (DIP_switch[0])
TOP_clk	Input	R8 (Internal clock source 50 Mhz)
TOP_buttons[0]	Input	D5 (GPIO_09)
TOP_buttons[1]	Input	A6 (GPIO_011)
TOP_buttons[2]	Input	D6 (GPIO_013)
TOP_buttons[3]	Input	C6 (GPIO_015)
TOP_buttons[4]	Input	E6 (GPIO_017)
TOP_buttons[5]	Input	D8 (GPIO_019)
TOP_buttons[6]	Input	F8 (GPIO_021)
TOP_buttons[7]	Input	E9 (GPIO_023)
TOP_uart_output	Output	D3 (GPIO_00)
TOP_uart_busy	Output	C3 (GPIO_01), not used, just for debugging
TOP_event	Output	A3 (GPIO_03), not used, just for debugging

Table 6: FPGA pin assignment

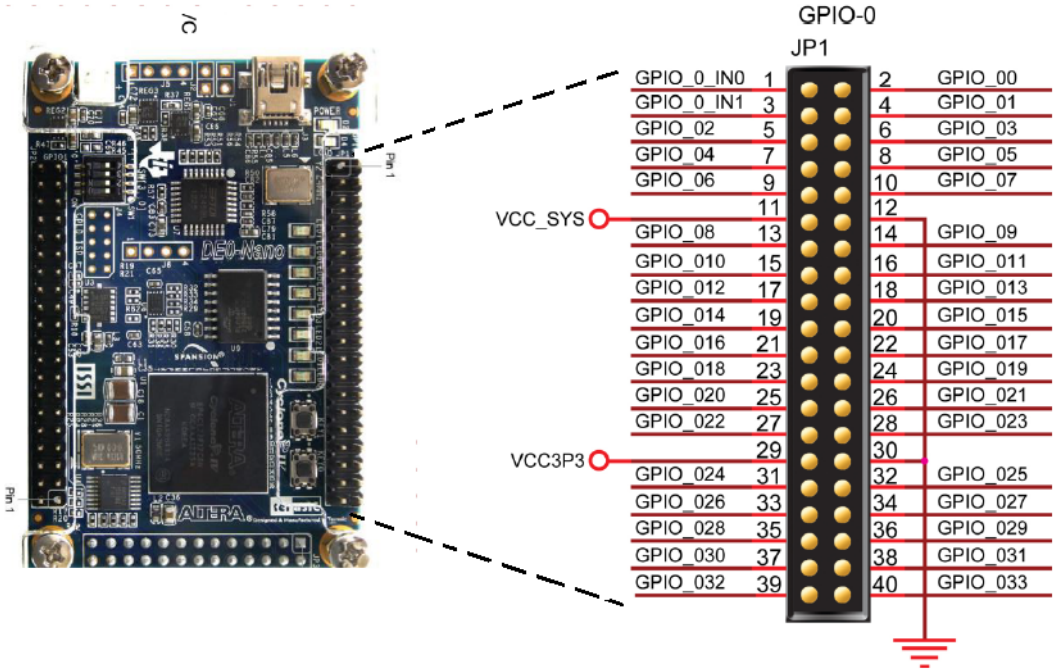

Figure 18: Altera DE0 Nano GPIO pinout

6 Interconnection protocol

The target interconncection protocol is the UART, which is in charged to allow communication from the FPGA and microcontroller. Main settings regard the baudrate (115200 bit/s), the length of data (8), the parity ODD bit and one stop bit. Considering also the start, the UART peripheral gets busy for 11 cycles.

The clock divider generates a train of pulses at 115200 Hz. I talked about clock divider implementation in the previous section. A module called UART synchronizer takes the peripheral busy for 11 cycles. This component receives as input that train of pulses, rather than the FPGA's clock. Conceptually, the synchronizer has a structure very similar to the one of clock divider: a counter, an hardwired register and a comparator. The synchronizer task is to send the shift command to the shift register storing the data to be sent via serial protocol. So, instead of using an equality comparator, I put here an inequality one. Until the value of the counter is less or equal than the value in the hardwired register (10) the UART is being busy. Another difference regards the counter resetting. The UART logic works only in response to an external action, like an interrupt; the train of pulses is generated periodically regardless of buttons status instead. I mean that, after those 11 cycles, the shift command gets disabled but counter is counting up without stopping. A classic counter gets overflow when reaches the all 1s string, turning the peripheral busy in absence of event ($0 \leq 10$). A *saturation counter* keeps at all 1s and has reset to 0 only when an external event has triggered.

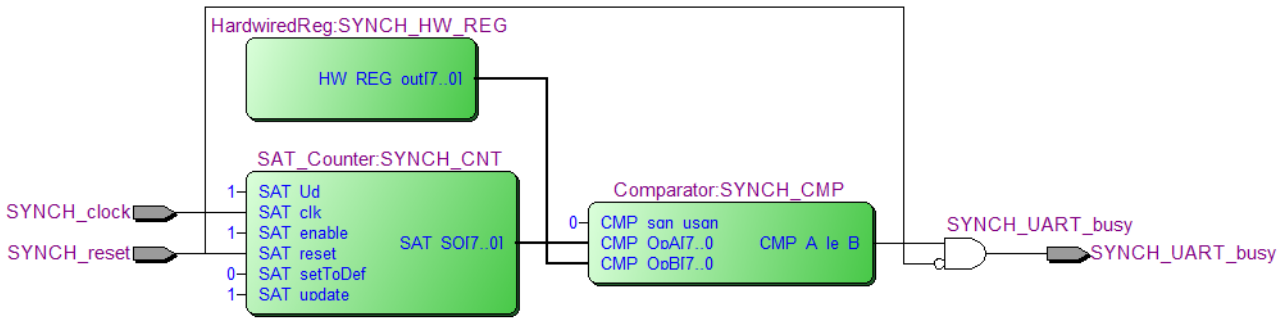

Figure 19: UART synchronizer architecture

I think that a couple of words should be spent on the shift register, which is named UART_Fifo. It is a common shift register parallel input serial output. However, it is between two frequency domains, the 50 Mhz and the 115Khz. The coming data is loaded with former domain, shifting has performed at the latter. The clock signal going in input to each flip flop is the result of an OR between the 115Khz clock and the load command travelling at 50Mhz. Then, a few combinational logic drives multiplexers in order to perform operations of KEEPING, SHIFTING and LOADING.

To prove the correct behaviour of my UART module, before I plotted the output signal onto the oscilloscope, then I used a software serial terminal like Hterm using the same settings I listed above; the picture shows one of eight buttons forming the keyboard pushed one at a time.

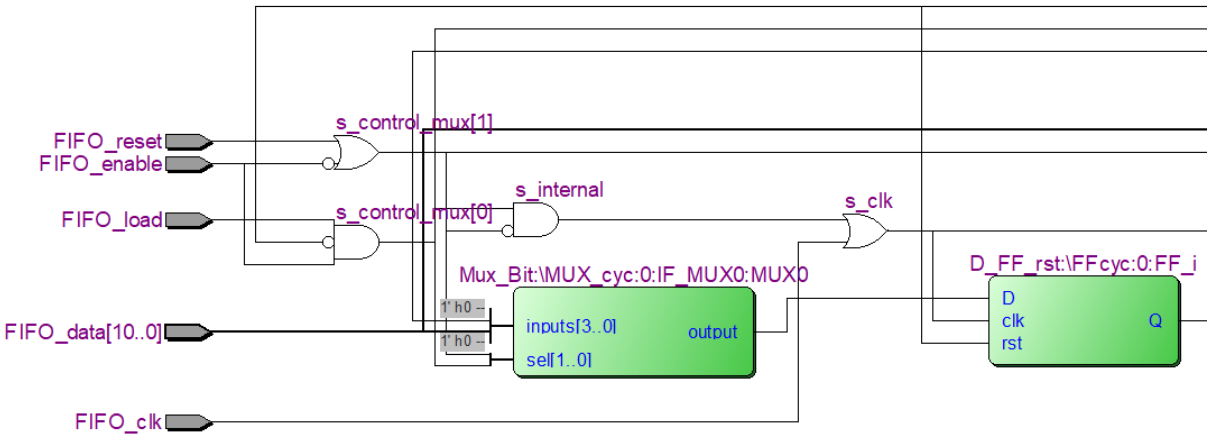

Figure 20: UART Fifo architecture: combinational logic to drive multiplexers and FF clock

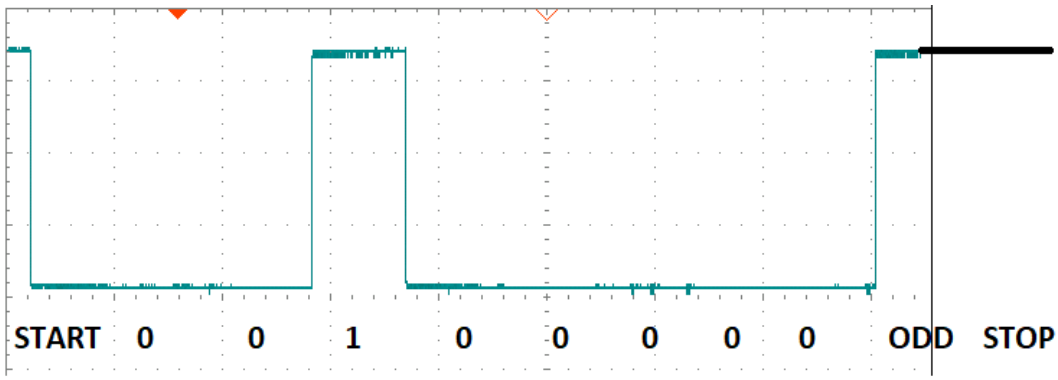

Figure 21: UART output on oscilloscope (10 µs/div)

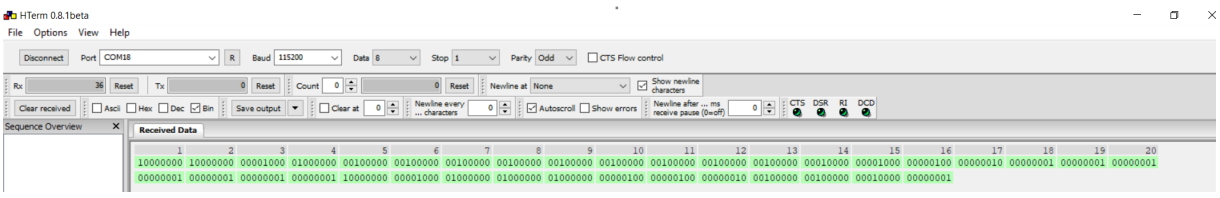

Figure 22: UART output on HTerm (serial terminal)

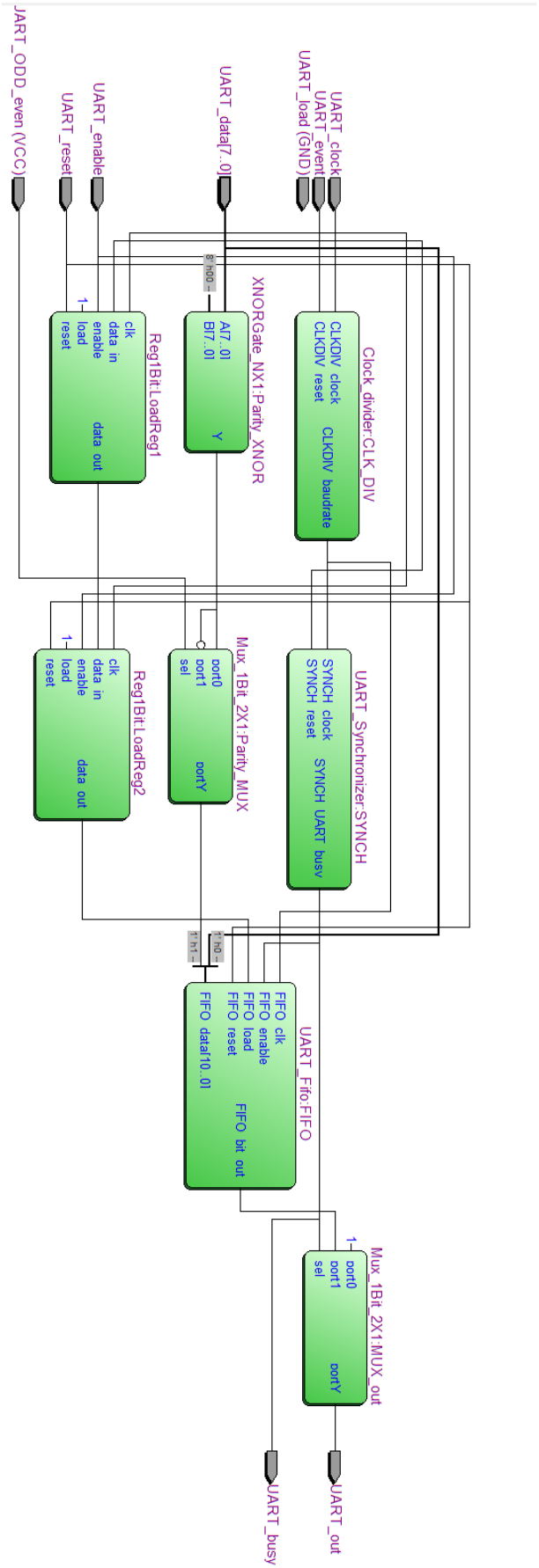

Figure 23: UART module architecture

7 Memory management

My goal for this part was to store the data read from buttons driver in a single memory location and then read it plotting tha value by means of the array of leds on the Altera board. First of all I had to implement the memory controller to run access protocols, but in any case I no reached to synchronize correctly the manager and the memory. In the end, my memory controller just emulates the access protocols, satisfying the waveform shapes for each clock cycles; however any consideration about access time or similar have been discharged.
Two words about the target memory. It's a 256 Mbits SDRAM provided by ISSI. It's fully synchronous, offers the possibility to do burst transfers, does auto-refreshing, allows programmable CAS latency and so on.

As first, I needed to know how FPGA is interfaced with SDRAM.

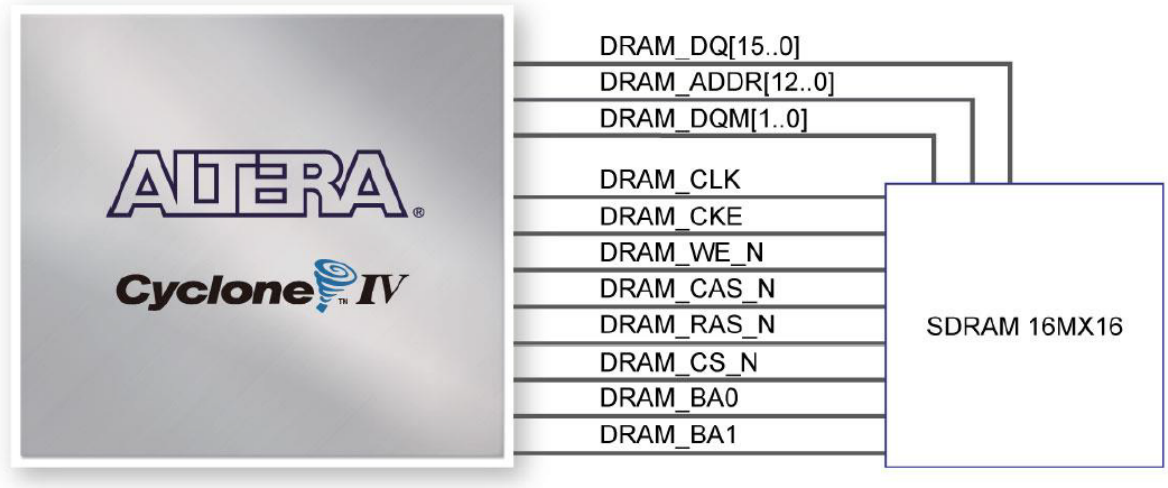

Figure 24: Connections between FPGA chip and SDRAM

There are 13 bits of address Ax and 16 bits input/output for data. The SDRAM is composed of 4 banks 8192 size each; to select one of them 2 input bits called Bank Select Address are used. Then the clock and clock enable are inputs as well. Moreover, there are the chip select, the write enable, the row address strobe and the column address strobe, all of them active low. This SDRAM offers possibility to mask the output, driving the 2 Data Masked bits, for the high and low parts respectively.

My module is a classic finite state machine, where each step is generated on the rising edge of the clock. The FSM is composed of 20 states. Looking at the datasheet, the state diagram is more complex so I decided to build it implementing the basic operations: the initialization, the programming, the request of a read the the request of a write. In between, memory expects to be precharged and refreshed. What I'm going to show in this section is the comparison between waveforms I'm able to generate and what is shown in the datasheet.

Listing 1: VHDL entity of memory manager

```
entity MemoryInterface is
        port(
                MI_buttons : in std_logic_vector(7 downto 0);
                MI_clk    : in std_logic;
                MI_reset : in std_logic;
                MI_enable : in std_logic;
                MI_RW : in std_logic; -- external event, no event reading, event writing
                MI_action : in std_logic; -- active low
                MI_address : in std_logic_vector(14 downto 0);

--              real interface

                MI_address_9_0 : out std_logic_vector(9 downto 0);
                MI_address_10 : out std_logic;
                MI_address_12_11: out std_logic_vector(12 downto 11);
                MI_bank : out std_logic_vector(1 downto 0);
                MI_data : inout std_logic_vector(15 downto 0);
                MI_CAS : out std_logic; -- active low
                MI_RAS : out std_logic; -- active low
                MI_CS : out std_logic; -- active low
                MI_write_enable : out std_logic; -- active low
                MI_clk_enable : out std_logic; -- active high
                MI_memory_clk : out std_logic;
                MI_DQML : out std_logic;
                MI_DQMH : out std_logic
        );
end MemoryInterface;
```

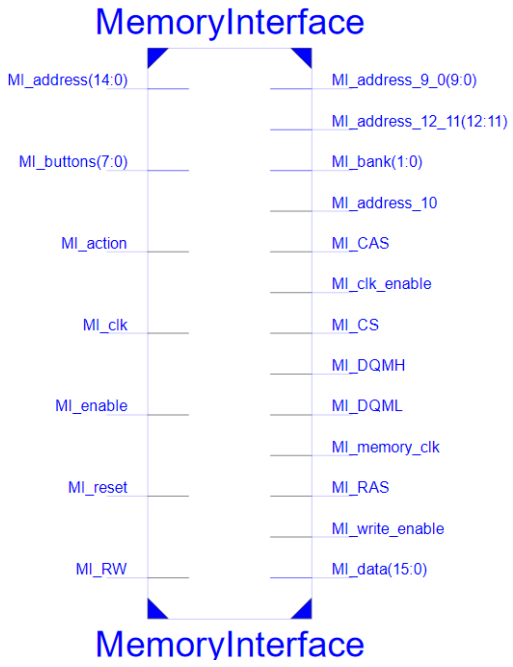

Figure 25: Memory interface pinout

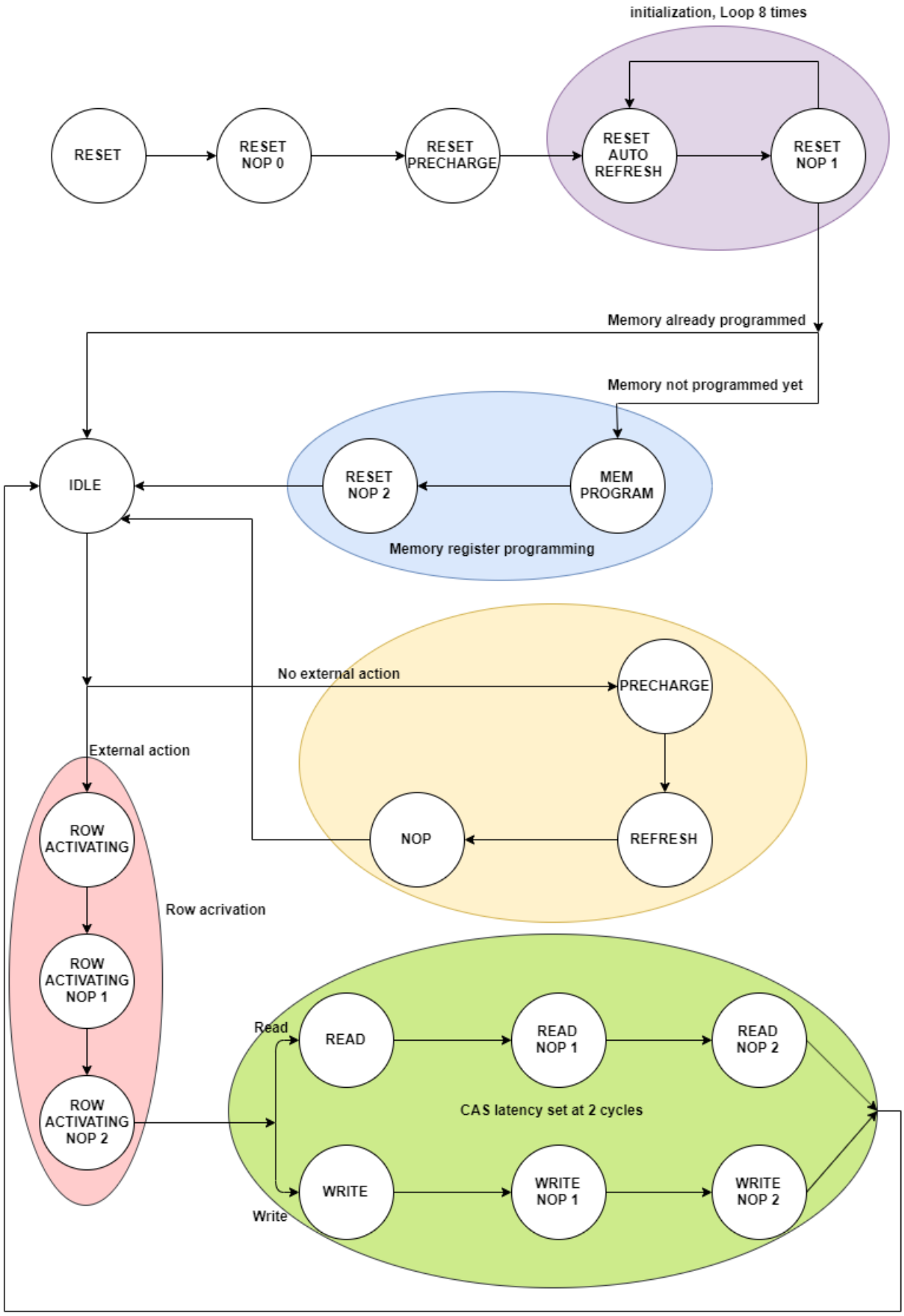

Figure 26: Memory controller state diagram

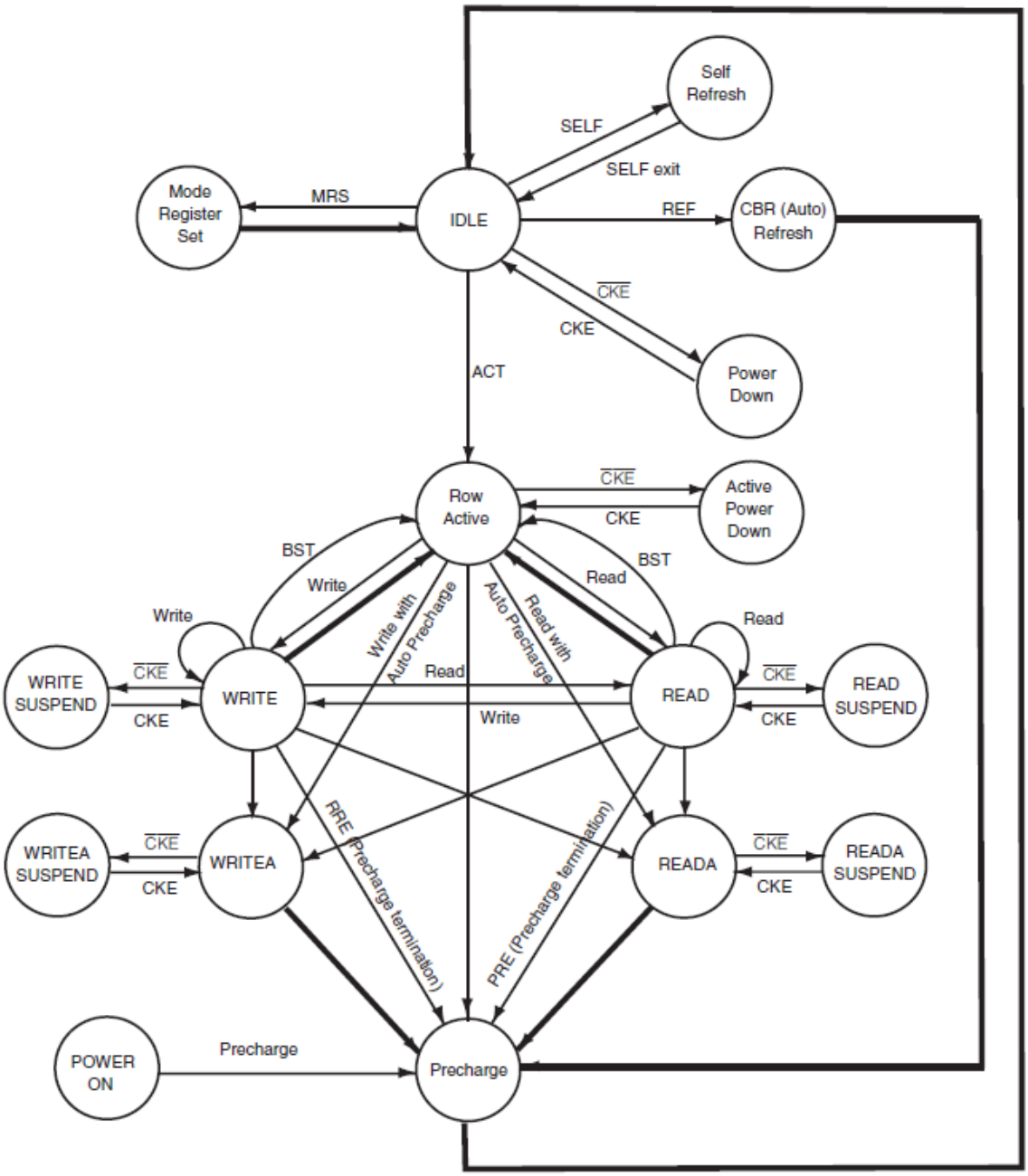

Figure 27: Memory state diagram from datasheet

When reset pin has activated, the SDRAM is once precharged, then for 8 times alternates a cycle of refreshing and one cycle of NOP. The bit 10 of the address, provides memory information about refreshing. If it's '1' all banks are refreshed, otherwise only the one is indicated by MI_bank bits. After initialization, the manager should be program the control register of the memory, which provides some information such as: burst length (1), burst type (sequential), CAS latency (2) and write burst mode (single location access). The state memory load register follows the end of initialization loop.

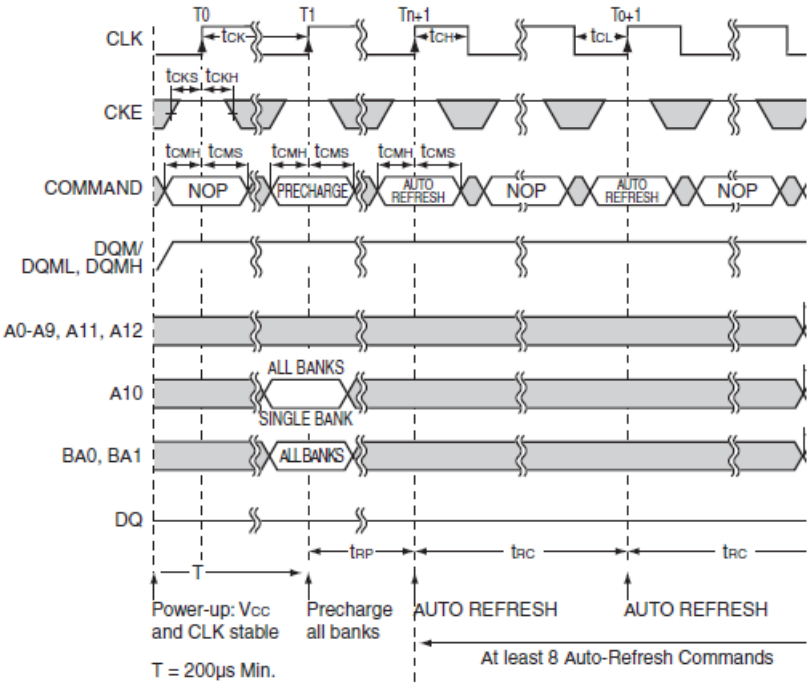

Figure 28: Initialization: datasheet

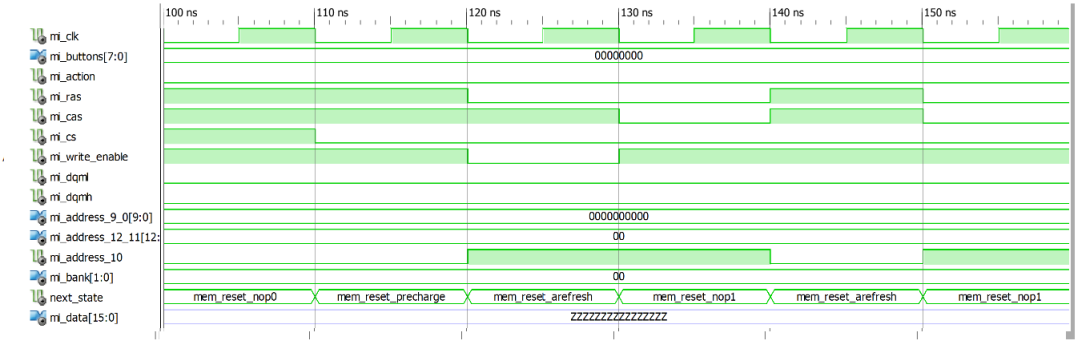

Figure 29: Initialization: simulation

From the idle state, the controller detects that an action has been driven from outside. From idle the state diagram moves toward the activation on the target row, which is passed on address pins and recognized by activating the row address strobe. After two cycles, memory controller sends the data and the column address, activating the CAS in the meantime. Since CAS latency has been set to 2 cycles, memory requires two NOP cycles to accomplish the operation. Small note for the reader: both read and write can be performed with or without auto-precharging, depending on the logic value on the 10th bit of address. Due to the fact that, in datasheet, the time diagrams of write both with auto precharge and without are totally equal, while read diagrams differ from the explicit presence of that (how we will see next), I guess that write with auto precharging diagram is wrong. Precharging command should not be sent from memory controller.

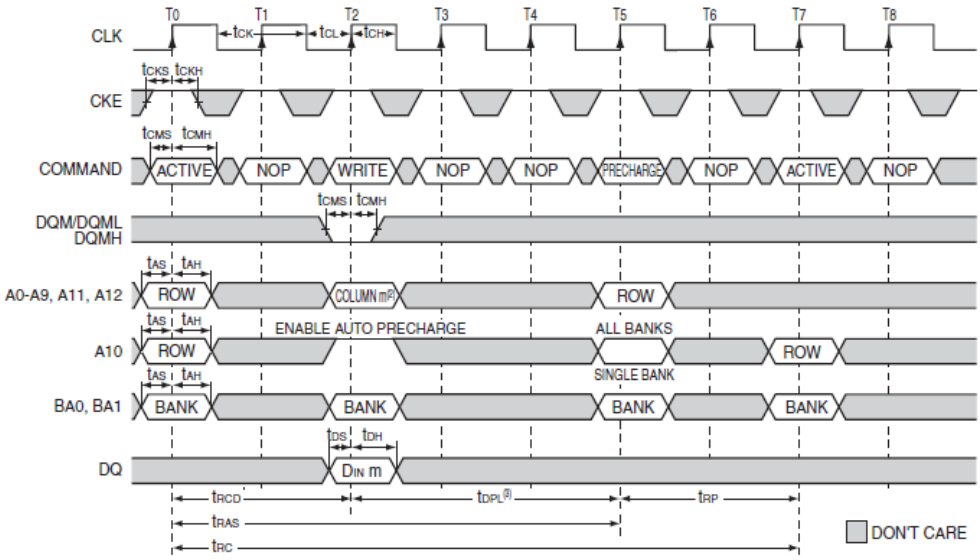

Figure 30: Write: datasheet

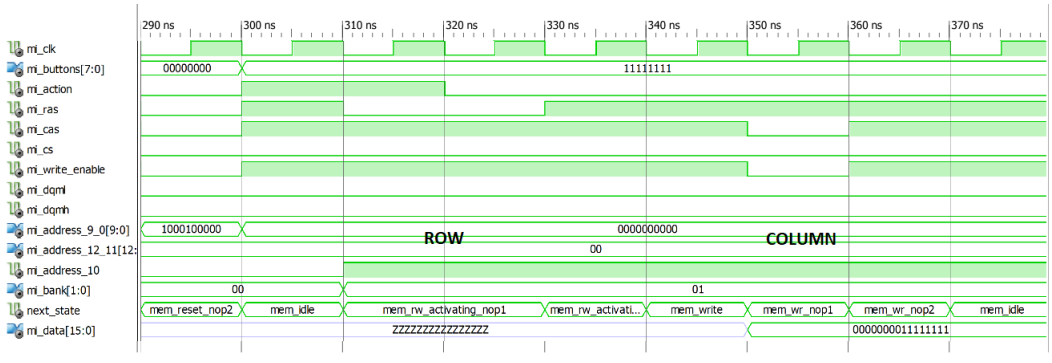

Figure 31: Write: simulation

As you can see from the datasheet diagram, when the operation has been set with auto-precharging, that command is not sent but automatically performed by memory internally.

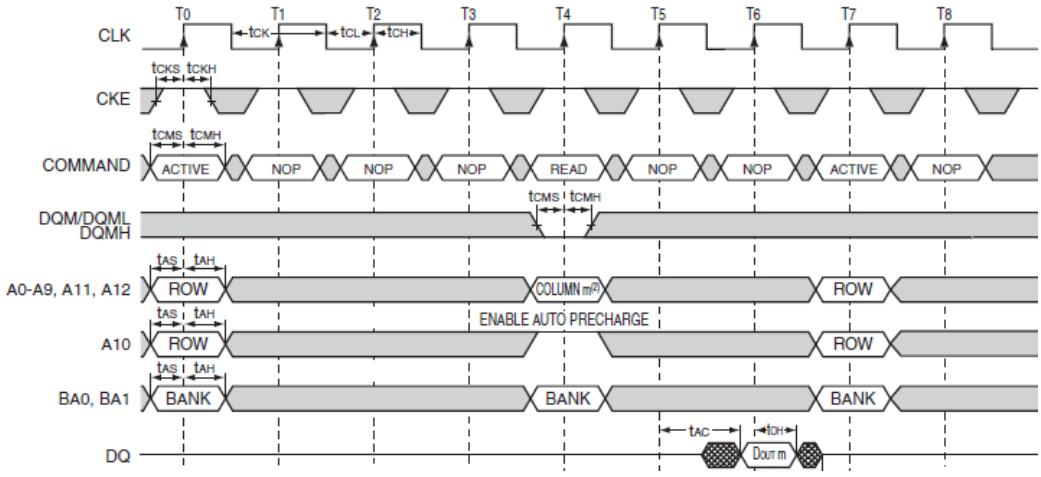

Figure 32: Read: datasheet

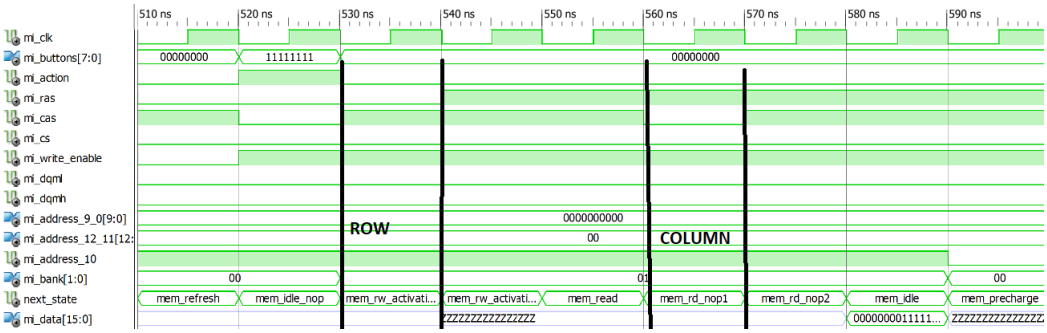

Figure 33: Read: simulation

COMMAND TRUTH TABLE

Function	CKE n – 1	CKE n	$\overline{CS}$	$\overline{RAS}$	$\overline{CAS}$	$\overline{WE}$	BA1	BA0	A10	A12, A11 A9 - A0
Device deselect (DESL)	H	×	H	×	×	×	×	×	×	×
No operation (NOP)	H	×	L	H	H	H	×	×	×	×
Burst stop (BST)	H	×	L	H	H	L	×	×	×	×
Read	H	×	L	H	L	H	V	V	L	V
Read with auto precharge	H	×	L	H	L	H	V	V	H	V
Write	H	×	L	H	L	L	V	V	L	V
Write with auto precharge	H	×	L	H	L	L	V	V	H	V
Bank activate (ACT)	H	×	L	L	H	H	V	V	V	V
Precharge select bank (PRE)	H	×	L	L	H	L	V	V	L	×
Precharge all banks (PALL)	H	×	L	L	H	L	×	×	H	×
CBR Auto-Refresh (REF)	H	H	L	L	L	H	×	×	×	×
Self-Refresh (SELF)	H	L	L	L	L	H	×	×	×	×
Mode register set (MRS)	H	×	L	L	L	L	L	L	L	V

Note: H=V_{IH}, L=V_{IL} x= V_{IH} or V_{IL}, V = Valid Data.

Figure 34: Main commands format

8 Conclusion

I started working on this system on 1st of October and the development ended on 12nd of November, for a global amount of 240 hours of workload. I'd like to highlight the main difficulties I found and how I fixed them, but it's something I'm going to do during the presentation of the project.
Approximately, considering all hardware I bought, not what I burnt, this kind of system costs 150 euros. Anyway, that price will be lower for sure. If I wanted to draw a real circuit PCB, for sure I would use the embedded ADC of the microcontroller; moreover I would choose a microcontroller having a smaller number of pins, so removing a lot of peripherals that I didn't initialize; then, the implementation by PCB would make the breadboard useless.
Another thing that I could improve is the Python script to re-build images. Currently it just generates black and white pictures, a better code would fill them by colors.
A further step consists in resizing components in Buck circuit, in order to optimize consumptions.
Some other pictures follow and end the report.