Embedded Systems

Lab4 – Camera, LCD & VGA conceptual design



Ecole Polytechnique Fédérale de Lausanne

Camera – TRDB-5DM

[Introduction 3](#_Toc502943320)

[Overview 3](#_Toc502943321)

[Avalon Slave 4](#_Toc502943322)

[Register Map 4](#_Toc502943323)

[Status: 4](#_Toc502943324)

[Control: 4](#_Toc502943325)

[Buffer Addr 4](#_Toc502943326)

[Frame length 4](#_Toc502943327)

[Changes from Lab3 5](#_Toc502943328)

[Note on FSM schemas and drawing conventions 6](#_Toc502943329)

[Avalon Master – DMA module 7](#_Toc502943330)

[Block diagram and FSM 7](#_Toc502943331)

[Changes from Lab3 8](#_Toc502943332)

[Timing diagrams 9](#_Toc502943333)

[Frame Format in Memory 11](#_Toc502943334)

[FIFO Module 11](#_Toc502943335)

[Camera Acquisition module 12](#_Toc502943336)

[Overview 12](#_Toc502943337)

[Schema 12](#_Toc502943338)

[Changes from Lab3 13](#_Toc502943339)

[Assumptions on the camera input signals and their timings 13](#_Toc502943340)

[Output signals 14](#_Toc502943341)

[Sampler 14](#_Toc502943342)

[Schema 14](#_Toc502943343)

[Timing diagram 15](#_Toc502943344)

[Bayer decoder 16](#_Toc502943345)

[Schema 16](#_Toc502943346)

[Timing diagram 17](#_Toc502943347)

[Demosaicing and downsampling in the READ\_R\_G2\_DECODE state: 18](#_Toc502943348)

[Utilization 19](#_Toc502943349)

[TRDB5-D5M configuration 19](#_Toc502943350)

[QSYS, top level, pinning, clocks 19](#_Toc502943351)

[Example software/poc 20](#_Toc502943352)

[Conclusion - Results 21](#_Toc502943353)

[Going further 23](#_Toc502943354)

Introduction

The goal of the project is to design and implement a FPGA-based system able to acquire frames from a camera and display them on a LCD screen. In the following report, we are going to focus on the camera acquisition part of the project, and will describe a camera controller IP component that will grab frames from the camera and write them in memory.

# C:\Users\LPI\AppData\Local\Microsoft\Windows\INetCache\Content.Word\overview.pngOverview

Figure 1: High level overview of the full system

Figure 2: High level view of our camera controller IP component

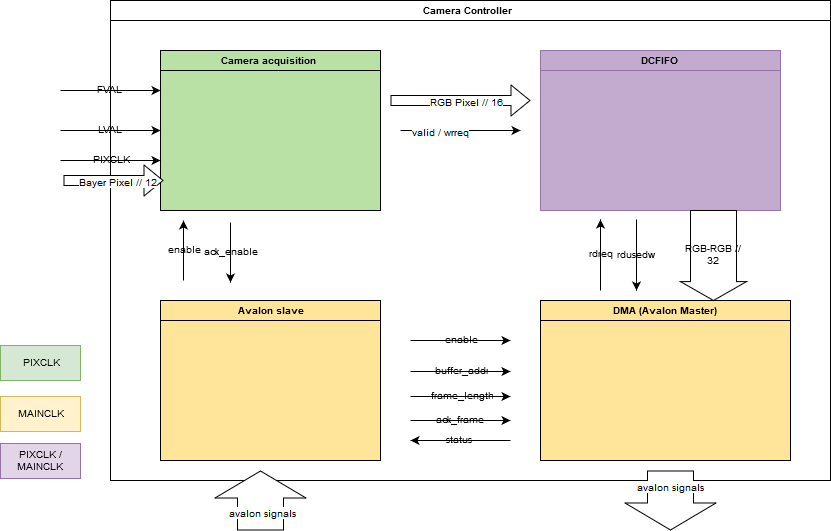


Figure 2: High level view of our camera controller IP component

As we can see in Figure 2 to implement our controller we split the logic into four interconnected submodules, an Avalon slave for configuration and control purposes, an Avalon Master for the DMA, a camera acquisition module and finally between the latter two a dual clock FIFO memory to “synchronize” their different transfers rates.

We will introduce each block of the one by one in details.

Avalon Slave

To configure/use our module, we offer a set of four 32bits registers available through our memory mapped Avalon slave:

# Register Map

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Addr / offset** | **Register Name** | **R/W** | **Description/Register Bits** | | | | | | | |
| 31 | **…** | 5 | 4 | 3 | 2 | 1..0 | |
| 0 | Status | R | 0 | | | | | | status | |
| 1 | Buffer\_addr | RW | Memory write start address | | | | | | | |
| 2 | Frame\_length | RW | Length in bytes | | | | | | | |
| 3 | Control | W | unused/reserved | | | | | | Ack\_frame | Enable |

## Status:

Read only register used to read the current state of our camera among the following 3 valid states:

**0 : IDLE**, the controller does nothing and is waiting a “grab frame” command

**1: GRABBING\_FRAME**, a “grab frame” command is being serviced, the controller is acquiring a frame from the camera and writing it to memory.

**2: FRAME\_WRITTEN**, the controller finished grabbing a frame and is waiting on a “ack frame” command to return to IDLE.

## Control:

Write only register used to issue one of the following commands to our controller:

**Enable:** “grab frame” command, if the controller is IDLE, setting this bit to 1 make it start grabbing a frame.

**Ack\_frame:** “ack frame”, if the controller is done writing a frame to memory and waiting upon an ack, setting this bit to 1 make it return to its IDLE state.

## Buffer Addr

Read/write register that contains the starting byte address where the DMA module will write the frame in memory.

## Frame length

Read/write register that contains the length in bytes of the frame.

Note that you should not modify these last two registers when the controller is in the GRABBING\_FRAME state, while modifying the former might have no incidence at all, updating the later can trap the controller forcing the need for a reset.

# Changes from Lab3

* We removed the DMA burstlength register and replaced it by a constant set to 16 to simplify the VHDL. Making the burstlength a true user settable parameter would have involved too many changes while our main goal was to have a first working system.
* No more interruptions, instead the status is meant to be polled, and is visible in the status register.
* We encode the status with 2 bits instead of 3 (and renamed them)
* We added an “ack frame” command allowing us to implement a kind of handshake between the software and the hardware. (see also explanation in the DMA part of the report).
* Since the enable signal is given by the Avalon slave module running at 50 MHz (MAINCLK) to the camera acquisition module which runs at 10 MHz, we have two different clock systems between the transmitter and the receiver, thus a handshake process must be used. To do so, we added an ack\_enable signal going from the camera acquisition module to the Avalon slave. Previously the enable signal was held only 1 MAINCLK cycle, now it is released only upon reception of an ack from the camera acquisition module. This way we make sure that the enable signal is properly read by the camera acquisition module. See figures below.

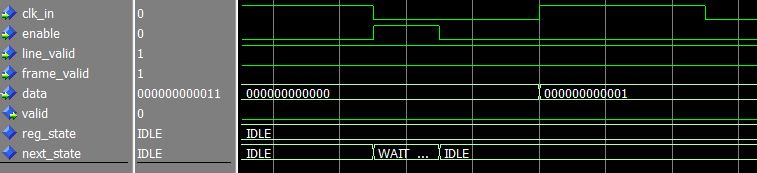
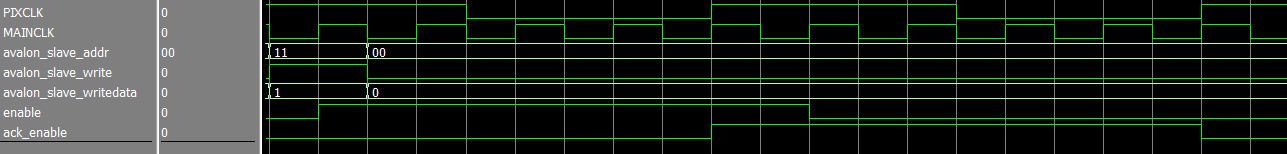


Figure 3: handshake between the avalon slave and camera acquisition modules

Figure 4: signals of the camera acquisition module illustrating the design bug, the module could miss the enable signal, hence we need a handshake

Note on FSM schemas and drawing conventions

Drawings such as the one shown in denote a register or clocked D Flip-Flop that update its output *reg\_name* to be *next\_name* at the rising edges of the clk.

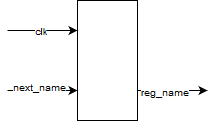


Figure 5: Clocked D flipflop

In the following FSM descriptions of the sub modules all state transitions are done on the rising edge of the clk (to the exception of the nReset transition) and decided asynchronously according to the conditions stated above the horizontal line of their label. Moreover, the actions described below the horizontal line of the labels are also asynchronous.

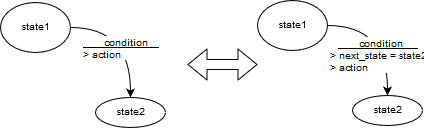
In other words, even if it is not clearly stated, the state is stored in a register such as depicted in and all the transitions update the next\_state signal as part of their actions such as depicted in .

Figure 6: Example of FSM transition

Actions of the form next\_name = newValue are only shown when they are relevant (but to be formal all “next\_name” signal should be present in all action of all transitions), it is implied that if nothing is said about a next\_name signal it is either set to the value of its reg\_name counterpart (the register hold its value) or set to a default value when more relevant (example the registers linked to an output signal).

These FSM descriptions are meant to be implemented as is using the two-process convention for FSM implementation in VHDL, one synchronous process to update the registers (including state) and one asynchronous to decide at any time the next value of the registers (including state and outputs) as a function of the inputs and the current state.

If not totally clear, see the timing diagrams that depict the desired behavior.

Avalon Master – DMA module

# Block diagram and FSM

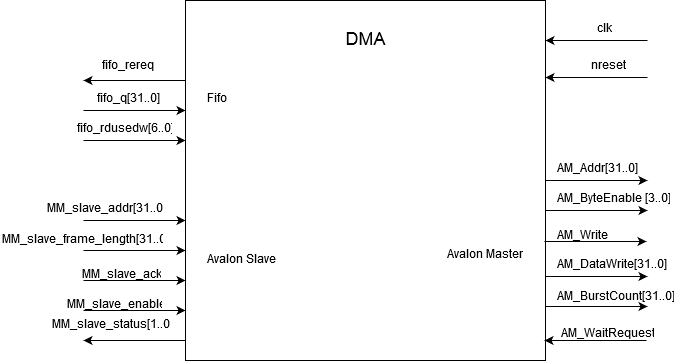


Figure 7: Block Diagram of the DMA module



Figure 8: FSM of the DMA module

# Changes from Lab3

* We noticed that we missed a fifo\_rdreq on the transition from wait\_data to write\_data, therefore we were writing to the memory two times the first output of the FIFO and were missing the last one.
* We added a kind of handshake convention between the software and the hardware. We added a new state WAIT\_ACK waiting for the ack signal from the Avalon slave. This is to prevent the state to change immediately from WRITE\_DATA to IDLE and this way allowing the software to notice that we are indeed done **and** that something was written.

# Timing diagrams

On the last Burst, shown in Figure 7, the state register goes to WAIT\_ACK after the burst. The status register change from “grabbing\_frame” (01) to “frame\_written” (10).

Notice that these timing diagrams were shot using the sensor emulator as input and this is why the data on the bus doesn’t change, but at each clock tick this is a new 32 bit word that goes out of the FIFO (see the rdusedw signal)



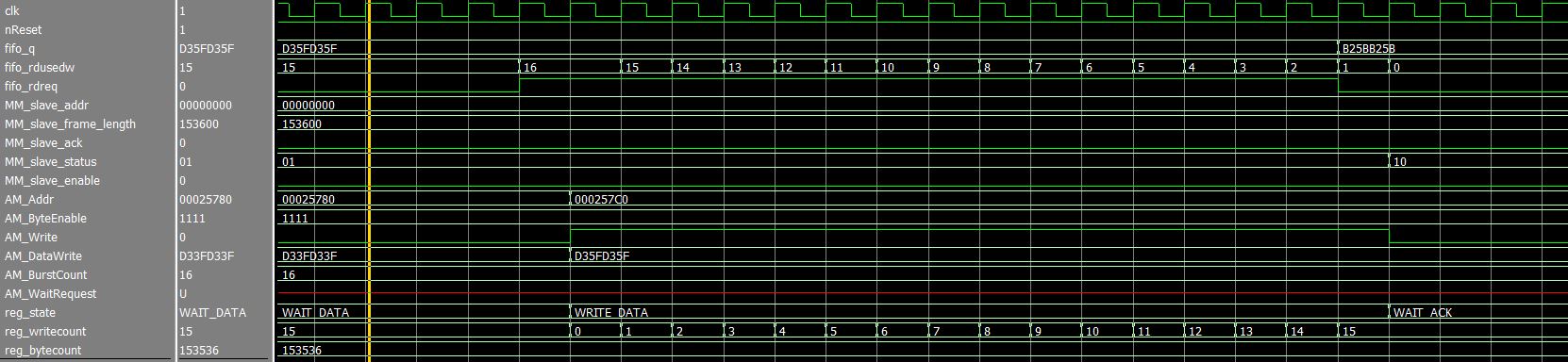


Figure 9: Timing diagram of the last burst

Figure 10: Timing diagram of the first burst



Figure 11: Timing diagram showing enable signal

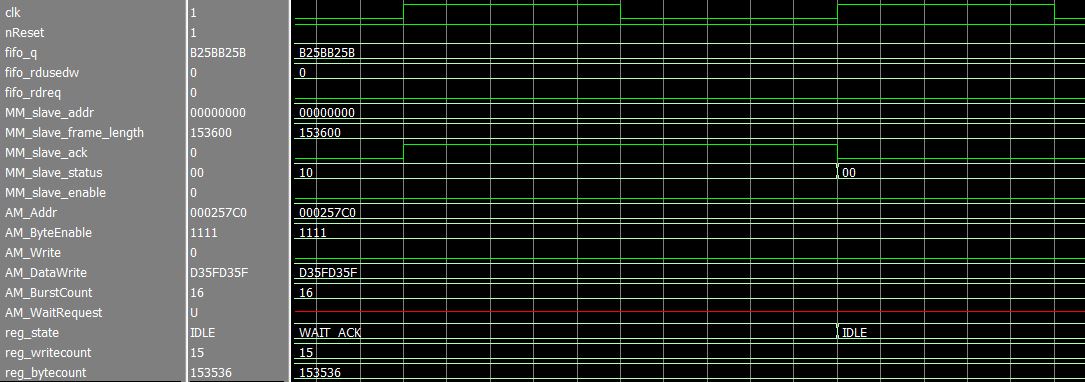
In the status changes from “Idle” to “grabbing\_frame” once the enable is triggered by the Avalon slave module. The state of the DMA FSM changes as well from IDLE to WAIT\_DATA.

Figure 12: Timing diagram showing the ack frame signal

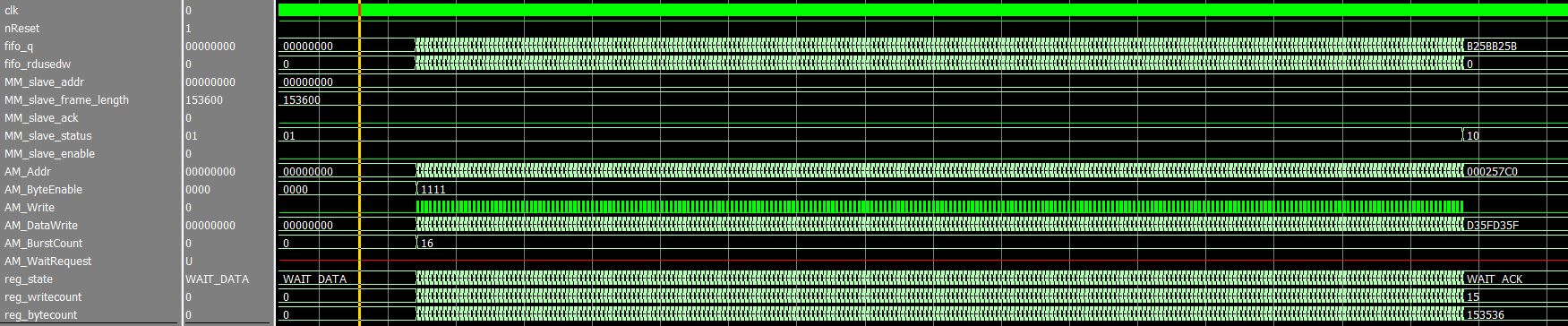
In the status changes from “frame\_written” to “Idle” once the Ack\_frame is triggered by the Avalon slave module. The state of the DMA FSM changes as well from WAIT\_ACK to IDLE.

Figure 13:Timing diagram of a full frame transfer

# Frame Format in Memory

The layout of the memory is the following one, as implied before, the start address and length of the DMA transfers are user provided through the Avalon slave Buffer\_addr and Frame\_length registers, see later for a typical use case.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Byte 3 | Byte 2 | | | Byte 1 | Byte 0 | | |  |
| R4 - R0 PIX2 | G5 - G0 PIX2 | B4 - B0 PIX2 | R4 - R0 PIX1 | G5 - G0 PIX1 | B4 - B0 PIX1 | Addr. 0 |
| R4 - R0 PIX4 | G5 - G0 PIX4 | B4 - B0 PIX4 | R4 - R0 PIX3 | G5 - G0 PIX3 | B4 - B0 PIX3 | Addr. 4 |
| R4 - R0 PIX6 | G5 - G0 PIX6 | B4 - B0 PIX6 | R4 - R0 PIX5 | G5 - G0 PIX5 | B4 - B0 PIX5 | Addr. 8 |
| R4 - R0 PIX8 | G5 - G0 PIX8 | B4 - B0 PIX8 | R4 - R0 PIX7 | G5 - G0 PIX7 | B4 - B0 PIX7 | Addr. 12 |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | - |
| - | - | - | - | - | - | Addr. 153596 |
| R4 - R0 PIX2 | G5 - G0 PIX2 | B4 - B0 PIX2 | R4 - R0 PIX1 | G5 - G0 PIX1 | B4 - B0 PIX1 | Addr. 153600 |
| R4 - R0 PIX4 | G5 - G0 PIX4 | B4 - B0 PIX4 | R4 - R0 PIX3 | G5 - G0 PIX3 | B4 - B0 PIX3 |  |
| R4 - R0 PIX6 | G5 - G0 PIX6 | B4 - B0 PIX6 | R4 - R0 PIX5 | G5 - G0 PIX5 | B4 - B0 PIX5 |  |
| R4 - R0 PIX8 | G5 - G0 PIX8 | B4 - B0 PIX8 | R4 - R0 PIX7 | G5 - G0 PIX7 | B4 - B0 PIX7 |  |
| - | - | - | - | - | - |  |
| - | - | - | - | - | - |  |

Figure 14: Example Memory Mapping

Here in Figure 14 two frames of 150KiB (320\*240\*16 bits) are written consecutively.

FIFO Module

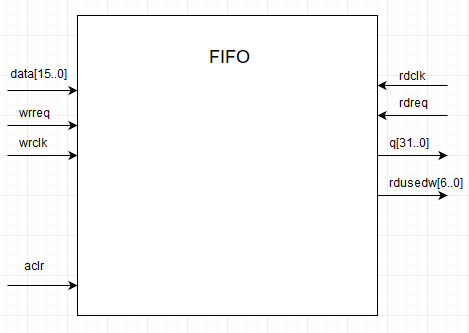


Figure 15: FIFO schematic

The system is divided into two areas running with two different clocks. We have a data producer running at 10 MHz (camera acquisition module), which is abstracted by the dual clock FIFO module, to match with the rest of the system running at 50 MHz. Since we are using it in show-ahead mode on the reading side, we can compare the process with the one seen in class. The rdreq signal is used as an acknowledge signal to warn the FIFO when it can update its output with the next data to transfer. The rdusedw[6..0] shows the number of words of 32 bits contained in the FIFO. The DMA module can start a transfer as soon as 16 words of 32 bits (the size of a burst) are stored in the FIFO. Notice that we fill the FIFO with words of 16 bits while we read it with words of 32 bits. Moreover, the read frequency is five times as fast as the write frequency and the FIFO is made large enough to avoid any overflow (can contain up to 8 DMA bursts).

Camera Acquisition module

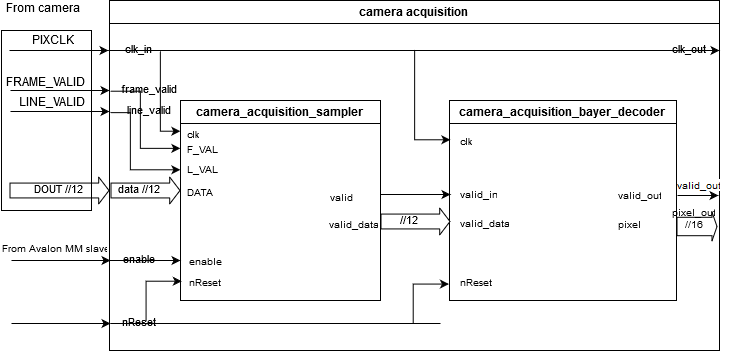
# Overview

The “camera acquisition” module is the part of the design that takes care of processing the output signals of our camera (terasic TRDB-D5M) to provide the rest of the system with a higher-level parallel interface that outputs a “stream” of 16-bit (RGB 565) pixels alongside a valid flag.

In other words it will implement a CAMIF[[1]](#footnote-1) block that is clocked with the PIXCLK of the camera and monitors its LINE\_VALID, FRAME\_VALID and DOUT[11..0] lines to acquire the sensors’ pixels (Bayer pattern) from the camera and interpolate them into image pixels.

The camera acquisition module “listens” for the enable trigger to start processing a frame from the camera and does not do anything the rest of the time. The module ensures that frames are processed in their entirety by waiting for the start of the next frame if the trigger is asserted in the middle of a frame output[[2]](#footnote-2).

## Schema



Ack\_enable

Figure 16: high level diagram of module

## Changes from Lab3

* As already stated, we added an ack\_enable signal to ensure that the enable signal is properly read by the camera acquisition module. See Avalon slave’s changes for more details.
* We had to change multiple times the pixel creation formula (how the four 12bit sensors’ pixels are interpolated into one RGB565 pixel) to have meaningful colors, see the conclusion for a subset of the different results we obtained and the current state of our project.

## Assumptions on the camera input signals and their timings

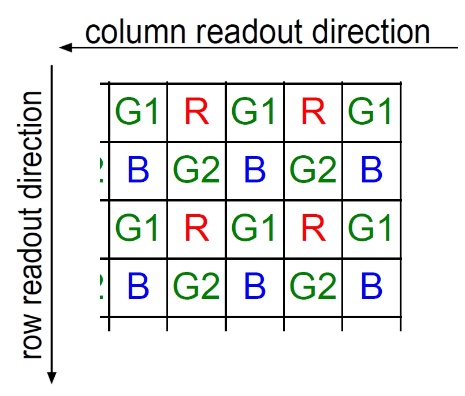
* Our camera acquisition module assume that the camera runs in continuous mode, i.e. it continuously samples and outputs frames.
* The camera outputs the sensors’ pixels using the following readout sequence and color pattern (i.e **no** dark rows/columns are part of the active image):

Figure 17: Bayer pattern and readout direction

Top left corner of frame

* Each PIXCLK cycle, when both FRAME\_VALID and LINE\_VALID are asserted one valid 12-bit sensors’ pixel outputs on the DOUT[11..0] lines[[3]](#footnote-3).
* Every time FRAME\_VALID is low LINE\_VALID is low too.
* There is at least one PIXCLK cycle between the output of two rows of a frame, during which LINE\_VALID goes low.
* Each frame have an even number of rows each with an even width not exceeding 1024[[4]](#footnote-4).

**TL;DR:** we assume that the camera behaves like the default configuration of our TRDB5-D5M camera (except for the frame dimension).

Every camera/sensor that fulfills these requirements should interface without problem with our design.

For further details, see the timing diagrams of the sampler and Bayer decoder submodules.

## Output signals

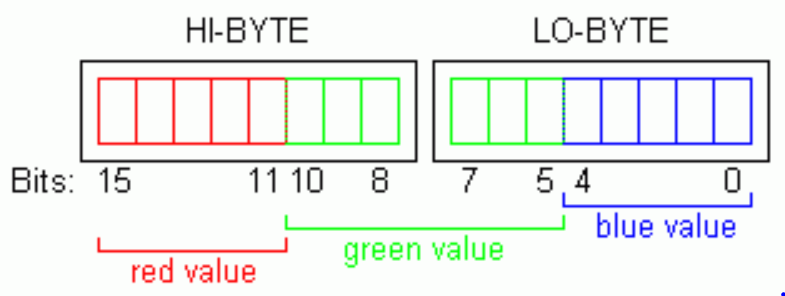
Each clk\_out[[5]](#footnote-5) cycles the camera acquisition module outputs a 16-bit image pixel (RGB 565) on the pixel out[15..0] lines, if the valid\_out signal is asserted, the pixel is valid

Figure 18: Pixel format

# Sampler

## Schema

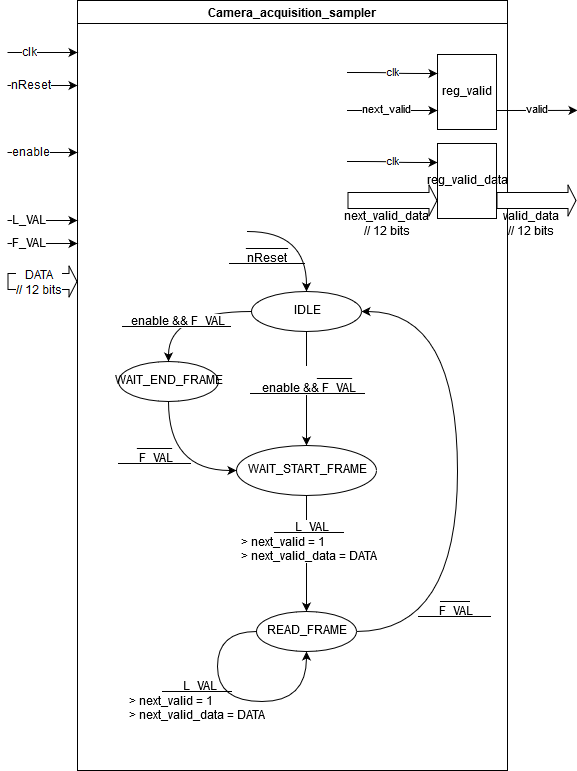


Figure 19: FSM / Behavioral description of the submodule

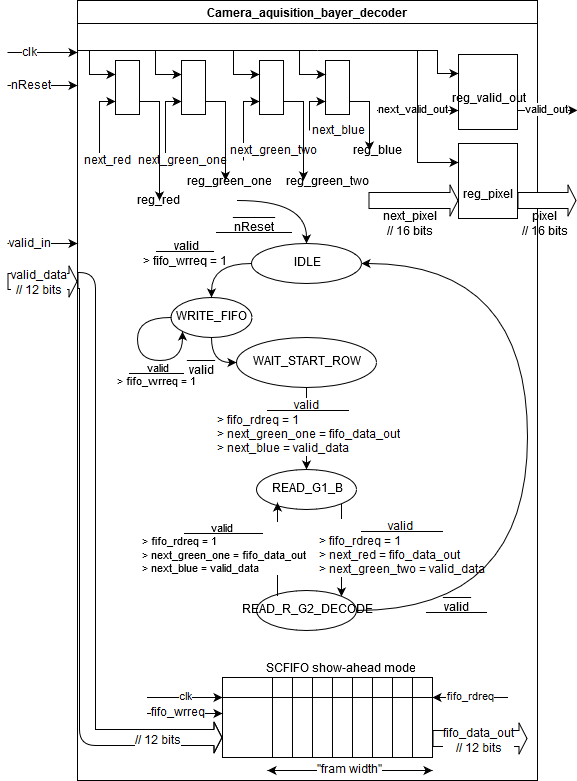
## C:\Users\LPI\AppData\Local\Microsoft\Windows\INetCache\Content.Word\timing_camera_acquisition_sampler_2.pngC:\Users\LPI\AppData\Local\Microsoft\Windows\INetCache\Content.Word\timing_camera_acquisition_sampler_1.pngTiming diagram

Figure 20: example timing diagram of the sampler submodule when enable is asserted during the gap between two frames (during vertical blanking)

Figure 21: example timing diagram of the sampler submodule when enable is asserted during the middle of a frame (here wlog and by bad luck at the beginning of a frame)

# Bayer decoder

## Schema



> next\_valid\_out = 1  
> next\_pixel = f(reg\_red/green\_one/green\_two/blue)

Figure 22: FSM / Behavioral description of the Bayer decoder submodule

## C:\Users\LPI\AppData\Local\Microsoft\Windows\INetCache\Content.Word\timing_camera_acquisition_bayer_decoder.pngTiming diagram

Figure 23: example timing diagram of the bayer decoder submodule  
(note that there is a (not important) drawing artifact on the reg\_green\_one and reg\_blue lines toward the end where a gapper is missing)

## Demosaicing and downsampling in the READ\_R\_G2\_DECODE state:

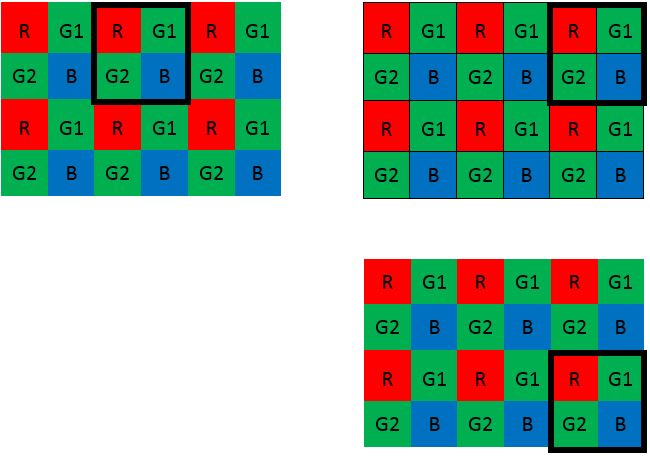


Figure 24: “demosaicing and downsampling readout direction”

Each even row (rows numbered from 0) are buffered in the FIFO (see Figure 22) while the pixels are computed during the receptions of the odd rows using the previously buffered row and current input. (hence, we keep the same readout directions and effectively divide the width and height of our frames by 2, See Figure 24 above and the previous timing diagrams for further details.)

The actual pixel decoding is done in the READ\_R\_G2\_DECODE state, where all the reg\_red/green\_one/green\_two/blue registers are retaining valid[[6]](#footnote-6) sensors’ pixels.

For the red and blue components, we need to go from a 12-bit color description to a 5-bit color description hence we take the 5 most significant bits of the sensors’ pixel. For the green component we have two 12-bit sensors’ pixel and we need to come up with a single 6-bit color description. To do so we average them (paying attention to the carry bit during the addition) before taking the 6 most significant bits of the result.

Now that the image pixel is decoded we can set the next\_valid\_out signal to 1 and the next\_pixel signal to its value and the result will be seen on the output on the next rising edge of the clock.

Note: Our module is designed to accept frames of 640\*480 sensor pixels following the Bayer pattern shown in Figure 17 and down sample them to 320\*240 image pixels (RGB565) but the only real limitation on the input frame dimension is the size of the FIFO used to do the debayerisation that should be large enough to hold an entire row. Currently the FIFO has a hardcoded width of 1024 12bits words hence our system should work for input width up to 1024. Another requirement is that the frames need to have an even number of rows and columns.

Utilization

# TRDB5-D5M configuration

We need to have the camera output 640x480 Bayer frames (our demosaicing module will reduce further the resolution to 320x240).

Essential registers to configure in order to achieve the basic functionality and meet the camera acquisition assumptions (if need to change the default value) (for description and further details see camera datasheet):

* Row Start 🡺 54 (1st row of active image) 64 (4X binning ask for multiples of 16 or 8 or—not very clear and contradictory => 64 should not harm)
* Column Start 🡺 16 (1st col of active image, + 4X binning wants multiple of 16)
* Row Size 🡺 1919 (to keep the full FOV, together with 4X binning)
* Column Size 🡺 2559 (to keep the full FOV, together with 4X binning)
* Output Control
  + Synchronize Changes 🡺 set before each full configuration and cleared after
* Read Mode 2
  + Row BLC 🡺 0 (this way no dark columns are part of the readout seq.)
* Row Address Mode (4X binning)
  + Row Bin 🡺 3
  + Row Skip 🡺 3
* Column Address Mode (4X binning)
  + Column Bin 🡺 3
  + Column Skip 🡺 3
* Black\_level\_calibration
  + Manual\_BLC 🡺 1 (this way no dark rows are part of the readout seq.)

All register write are done through the two wire I2C serial bus via software and using the provided I2C controller. All the configuration is done in one batch enclosed between the two Synchronize changes writes. Furthermore, before each configuration we ask for a soft reset of the camera to ensure that all important registers are holding their default values.

# QSYS, top level, pinning, clocks

Nothing special was done, we used the provided template and added one PLL for the clocks, the given I2C controller IP and our custom IP and wired them all together, to be sure to avoid any memory caching and licensing issues we used the NIOSII/e instead of the NIOS/f.

As already mentioned we chose a 10 MHz clock for the camera and a 50 MHz clock for the rest of the system.

# Example software/poc

We provide code for a typical utilization including camera configuration and basic testing of the state of the system.

In the given code, we make use of the altera\_hostfs package of the BSP to dump written frames to our filesystem, hence it is meant to be run in debug mode.

When merging the designs with the LCD controller team we have the following scenario for the main loop:

To avoid any problem overwriting a frame that wasn’t totally displayed, we use two[[7]](#footnote-7) different areas. (like in Figure 15).

* First, we start grabbing a frame into the first area of the memory.
* When we are warned that a complete frame is ready, we warn the LCD controller that a frame is available at that location and that it can start displaying it.
* We then continue grabbing the next frame at another location,
* When the next frame is ready in memory, if the LCD controller is done displaying the previous frame, fine we can give it the new location, exchange the memory addresses and loop, if not we wait (but we could tune a little by adding another memory area, see7). Our camera controller is blocked until we release the ack frame signal by software.

See the register map of both the camera and LCD controllers to know what actual registers to read/write to implement the wanted behavior.

Note that we expect that the LCD controller offers means to know when it is done displaying a frame and that, in the case of it being faster than our camera controller, it is able to wait and keep the previous frame displayed until we instruct it to display a new one.

This way, the software has complete control on the whole system including the frame rate (constrained by the max framerate of the camera that depends on its configuration, the memory transfer rates and the display rate of the LCD) and knows exactly the state of the transfer. [[8]](#footnote-8)

Conclusion - Results

Our system works well in isolation but wasn’t, at the time of writing this report, tested concurrently with an LCD controller implementation.

We lost some time figuring out why the memory seemed to be updated at multiple place (trusting the memory view of the debugger in eclipse) while it turned out this wasn’t the case or it didn’t matter at all, but the question would merit some more investigation.

But putting aside the missing handshake design bug we described before we didn’t have to make fundamental changes to our design and mainly lost time trying to improve the images we got and figuring out why they were bad. Keep reading for a subset of the images we obtained and the things we tried (not exhaustive) in chronological order.

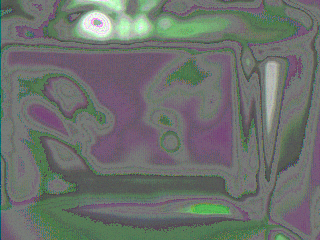
First results using the debayerisation formula designed in lab3 (basically we “orred” two by two the most significant bits):

Figure 25: first results, left: picture of a laptop, (the upper right corner of the screen was a red towel), right: the terasic box featuring their dragon logo.

Then we changed our debayerisation formula to keep only the most significant bits of each channels and only one green channel. We obtained results looking like the following:

Figure 26: fruit basket featuring oranges on the top and bananas on the bottom

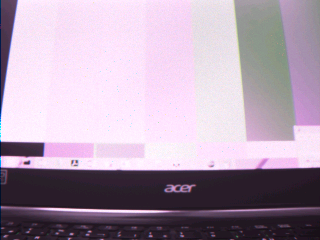
Now we tried to average the green channels and obtained the following results:

Figure 27: laptop displaying the SMPTE color test pattern, green channels averaged (paying attention to the carry bit)

One thing we could notice is that red is green and green/white/luminous colors are rosish. It turned out we were computing the pixel out of a “shifted Bayer pattern” (Figure 29), to quickly verify the hypothesis and hack around the bug we asked the camera to mirror the columns (Figure 31). But before that we were not sure if it could be a problem of endianness or camera calibration and we tried to process the binary files with various quick python scripts and/or ffmpeg commands. During our tests, we noticed too that we were getting slightly better (less blurred) pictures with the camera of another group.

Figure 28: strange column (leftmost), debayerisation bug

Figure 29 : circled in red, the "shifted Bayer pattern", explaining why red is green, green is rosish etc.

Figure 30: green channels averaged (naïvely) without paying attention to the carry bit

A possible explanation that would agree with the fact that we don’t observe the bug in our modelsim testbench[[9]](#footnote-9) (see Figure 32) would be that we configure the camera with incorrect column\_start and row\_start values but we didn’t manage to resolve the matter even with different values. What is even more strange is that the first column we obtain is not similar to the other (speaking of colors) nor black nor random but is still directly correlated to the adjacent one, see what I am saying on the figure to the right.

Figure 31: column mirroring hack

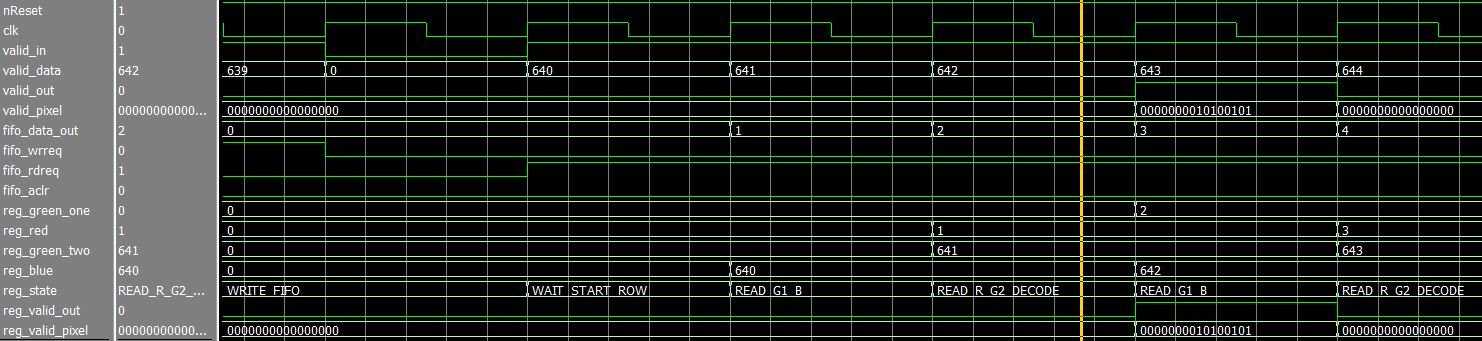
Finally, we still don’t know what happens and we haven’t investigated the matter further[[10]](#footnote-10). We have kept the mirror column hack to have good colors implying that the images we obtain are mirrored.

Figure 32 : Bayer decoder module signals, during testbench execution, the registers used to do the debayerisation are loaded with their correct values.

# Going further

The system developed in this report remains a first solution with simple configuration to process a Bayer CFA of 640\*480 and convert it into a raw 320\*240 RGB565 frame.

Our goal was to have something working at first hence we kept it simple, but here is a list of modifications which could have been made to enhance the system, we didn’t try any of them because until recently our frames were not very convincing and we still have some interrogations on the colors we obtain (see previous page):

* We could make the implementation more generic and the IP component customizable by the user at the system creation time.
  + different frame sizes. To do that, we should ensure that the FIFO used for the debayerisation is large enough to contain one full line of the frame.
  + the size of the DMA burst could be configurable by the user through a register, again we should pay attention to the size of the FIFO used between the CAMIF and the DMA module.
  + More pixel interpolation options, and supported pixels format, way to choose output resolution if for example the design is used with a VGA controller
  + Add an interruption based mechanism and an option to choose between it and the current polling based mechanism to warn the LCD part that a frame is available in memory
* Add other statuses and debug information showing for example the number of byte transferred in the memory, the states of the FIFOS the actual size of the frames that are coming from the sensor.
* Add some more protection, for example prevent the address and length registers being updated while the controller is grabbing a frame or cache them in a local register of the DMA module. Monitor the state of the FIFOs and assert they are correctly emptied and filled, no overflows, error recovery procedures etc.
* Better resource management, currently the camera is continuously outputting frames even if we don’t grab them.
* More sound C library, calibration of the camera, etc.

Together with these improvements we could have tried to see how our system behaves under other scenarios, especially with a higher clock frequency for the camera, but as we didn’t have the opportunity to even test it with an LCD controller, it sounded like a bit futile or premature.

1. Camera Interface, block that provides a standard output from the sensor interfaces. [↑](#footnote-ref-1)
2. See timing diagram and FSM of the sampler submodule. [↑](#footnote-ref-2)
3. All launched with the rising edge of the PIXCLK. [↑](#footnote-ref-3)
4. To not overflow the FIFO used to do the debayerisation, see below in the Bayer decoder part [↑](#footnote-ref-4)
5. Same as clk\_in, same as PIXCLK, only for convenience purpose (if any) by doing this our camera acquisition module behaves like an enhanced/convenient camera, to push things further we could have kept the LINE\_VALID and FRAME\_VALID conventions. [↑](#footnote-ref-5)
6. belonging to the same pixel according to Figure 22 [↑](#footnote-ref-6)
7. But we can use more and implement kind of a rotating FIFO buffer of frames with our cam controller as producer and the LCD controller as consumer. [↑](#footnote-ref-7)
8. According to the datasheet of the camera, at 10MHz with our current settings the framerate is roughly 8.4 fps but the final/actual frame rate is controlled in software by the enable triggers and constrained by the states of the transfers and/or the chosen algorithm. [↑](#footnote-ref-8)
9. we use the provided sensor emulator module; hence we can verify that the registers used to do the debayerisation are loaded with correct values. [↑](#footnote-ref-9)
10. .. and aren’t very satisfied. [↑](#footnote-ref-10)