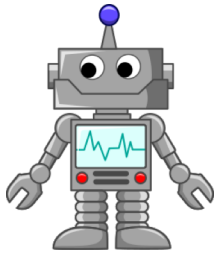
Sistemi Digitali M

07 - Embedded computer vision



Introduction

In many application domains, such as those involving autonomous navigation of vehicles or drones, is crucial to infer cues (eg, depth or flow) from images deploying embedded devices with a limited power budget

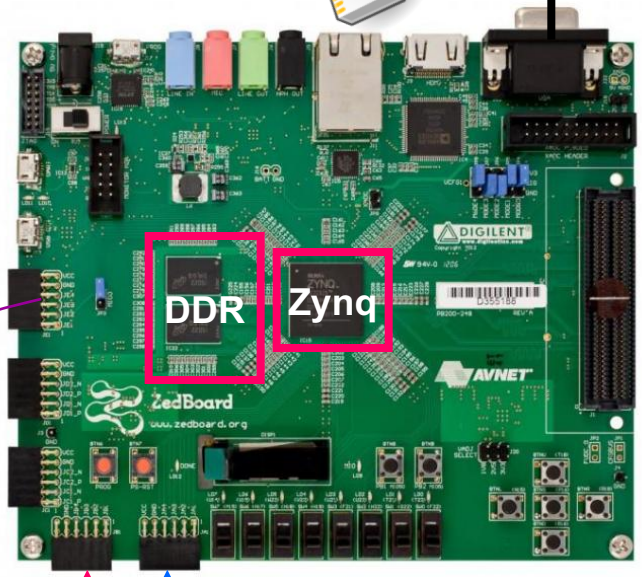
We will describe next a whole computer vision (CV) pipeline, implemented from scratch and totally with high-level synthesis tools, on a Zynq evaluation board

Specifically, we will introduce:

- Imaging device technologies
- A simple, yet effective, CV processing system
- Strategies to implement CV modules on FPGA
- OpenCV app for remote image visualization

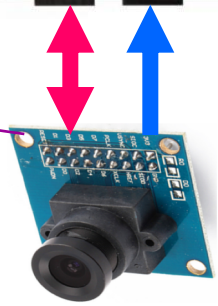
Overview 1/3

Zynq 7000:
Standalone
or Linux Os



VGA

Programming
I2C



1 Gb
Ethernet

OpenCV
App

Overview 2/3

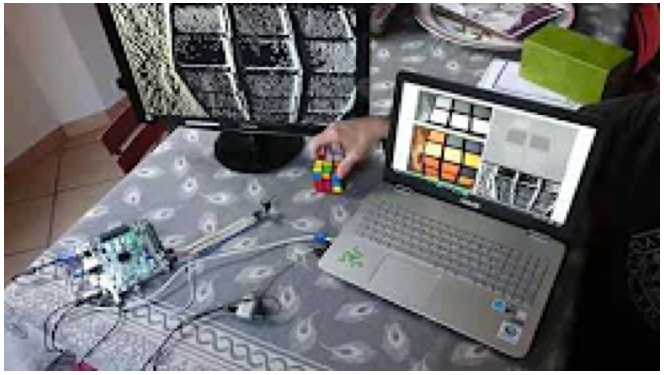
- Whole processing pipeline mapped on the FPGA
- Image sensor OV7670
- Circular frame buffer (DDR memory)
- VGA output
- Optional ethernet streaming, interrupt-driven, from ARM to remote host with OpenCV visualization

The whole project is available on Github at this link:

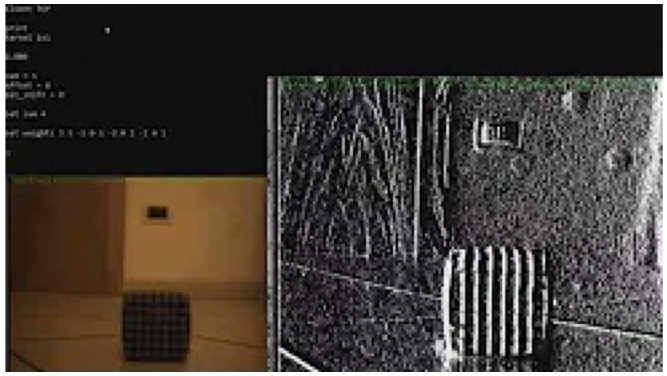
https://github.com/smatt-github/SmartCamera

Overview 3/3

Videos of the overall system



https://www.youtube.com/watch?v=6HEgdZEHpsM

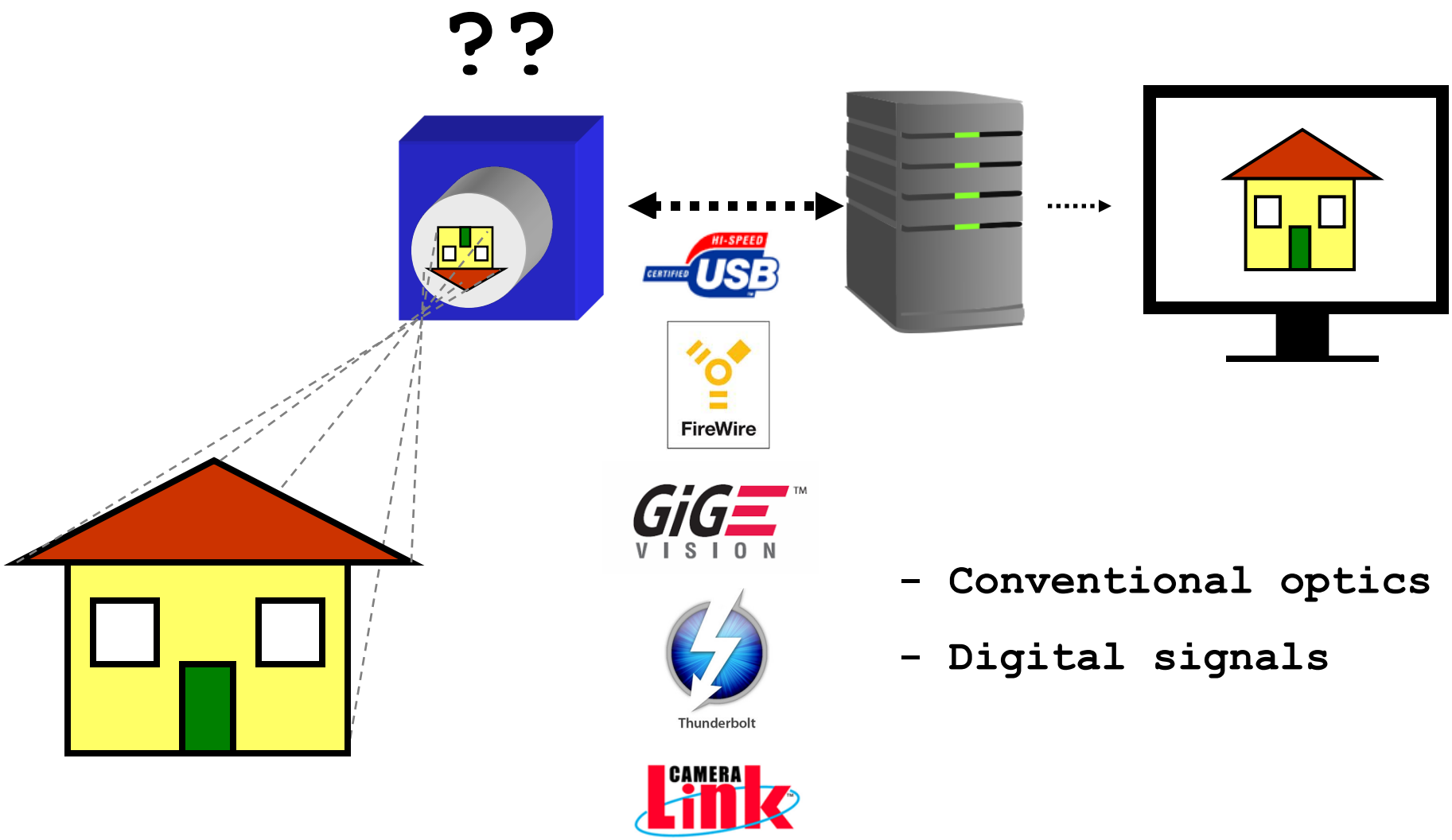


https://www.youtube.com/watch?v=QFxjXWgBBcc

https://www.youtube.com/watch?v=EG3NYqMJvZI

Imaging sensors

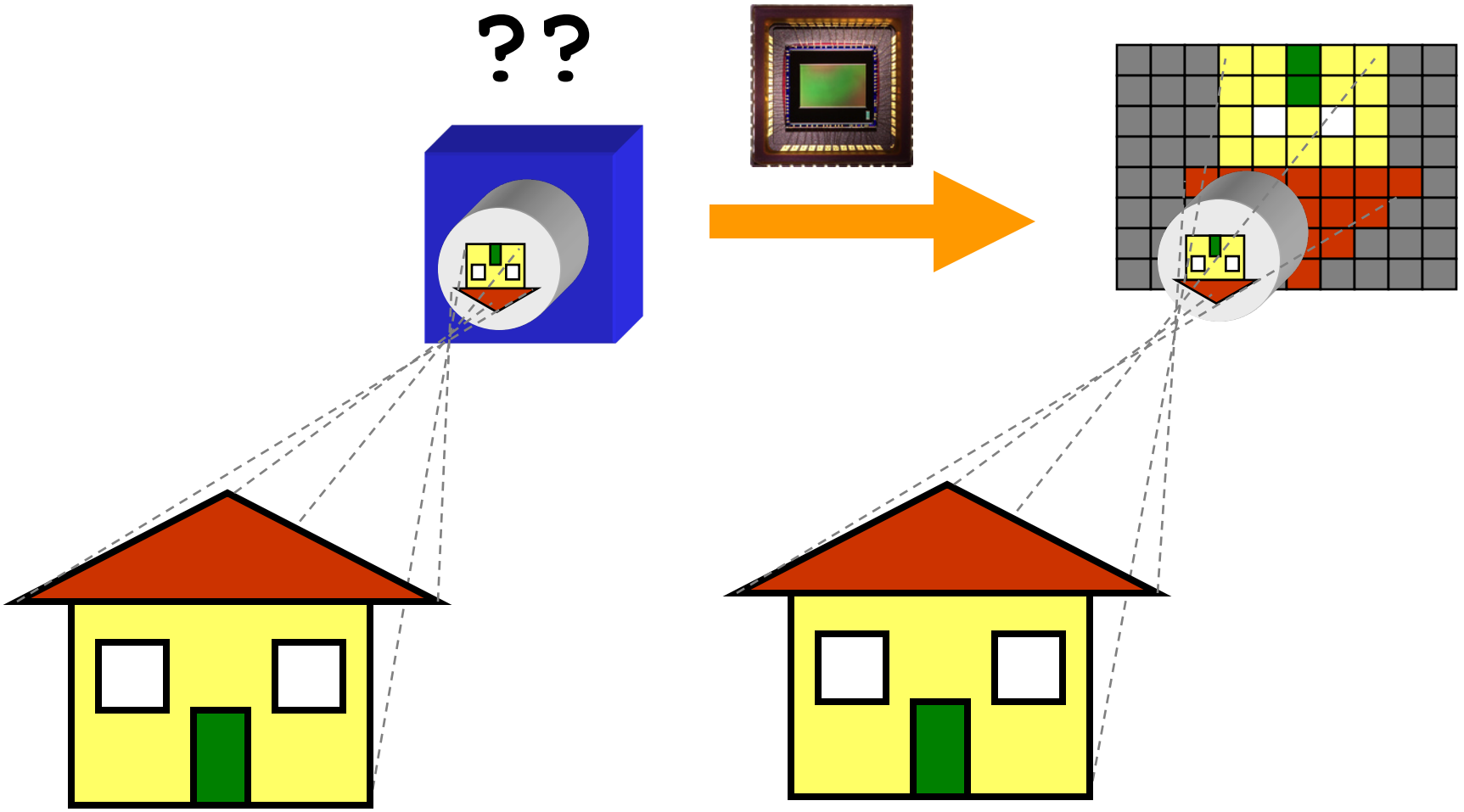
Consumer cameras, webcams, industrial cameras, etc



- Conventional optics
- Digital signals

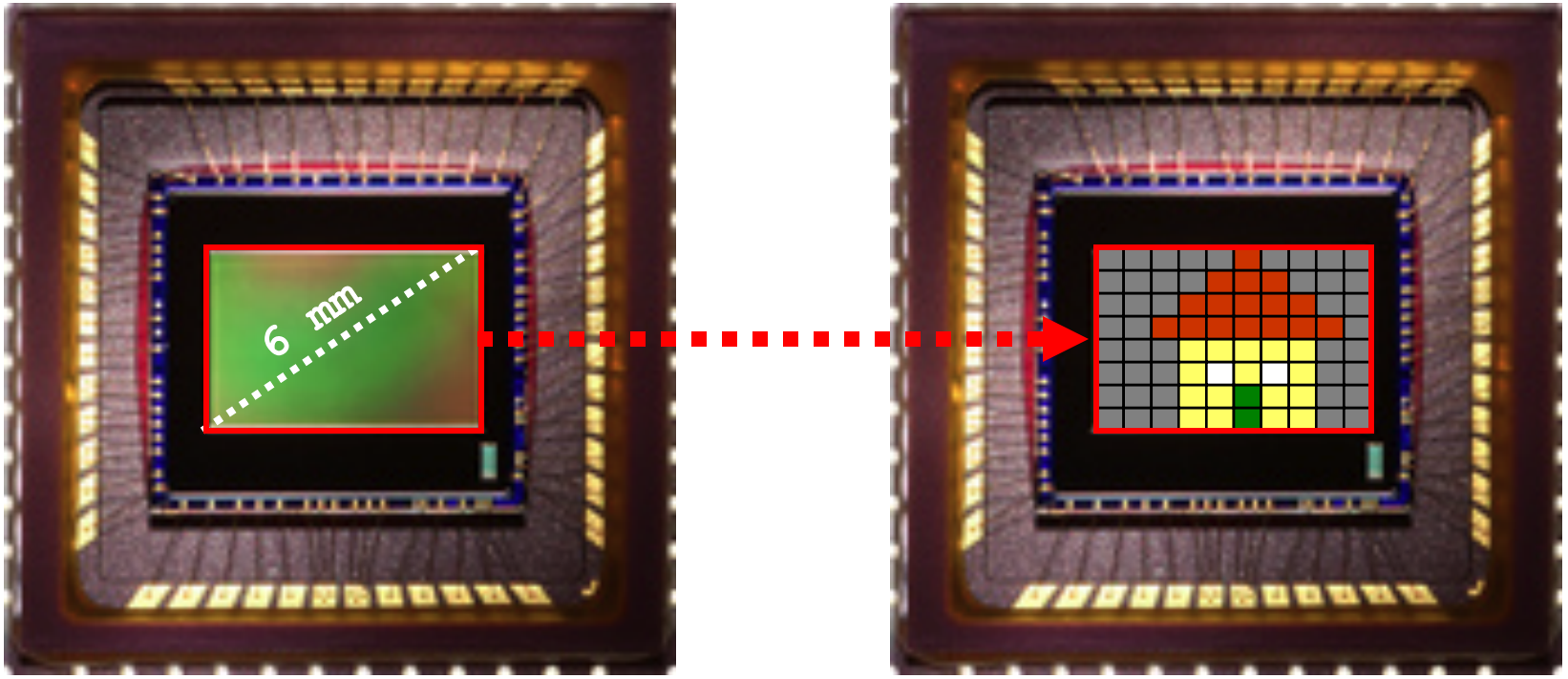
Acquisition process

Array of photosensitive elements (pixels)



Imaging sensor

Example: Aptina/ON global-shutter 1/3″ (i.e. diagonal is about ≈ 6 mm).



https://pixhawk.ethz.ch/electronics/camera

Technologies 1/2

There are two main technologies: CMOS (mainstream) and CCD (for specific application domains)

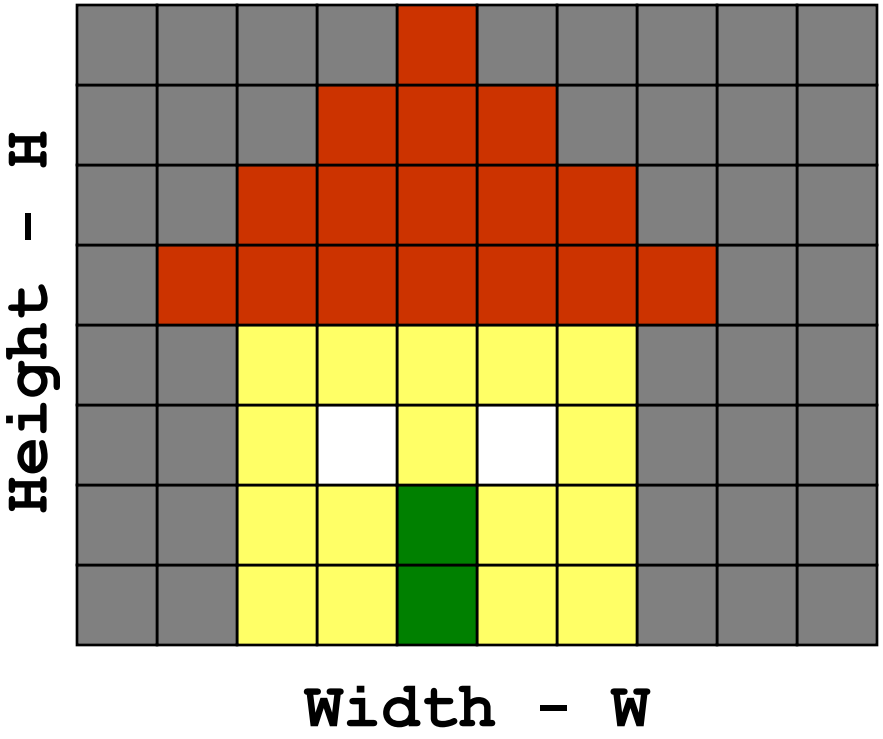


Image resolution: from 0.x MP to several MP

Widely adopted resolutions:

VGA : 640x480 (0.3 MP)

Wide VGA : 752x480 (0.35 MP)

Technologies 2/2

- *rolling shutter*
- *global shutter*

rolling shutter

www.ptgrey.com/

global shutter

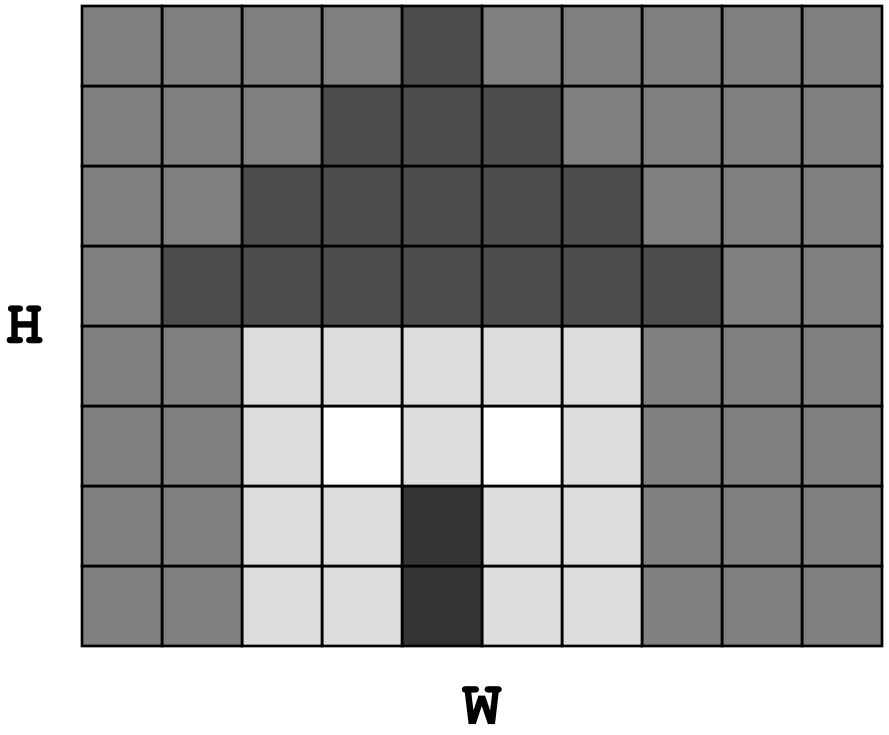
www.ptgrey.com/

Global shutter is better...

http://scorpionvision.co.uk/

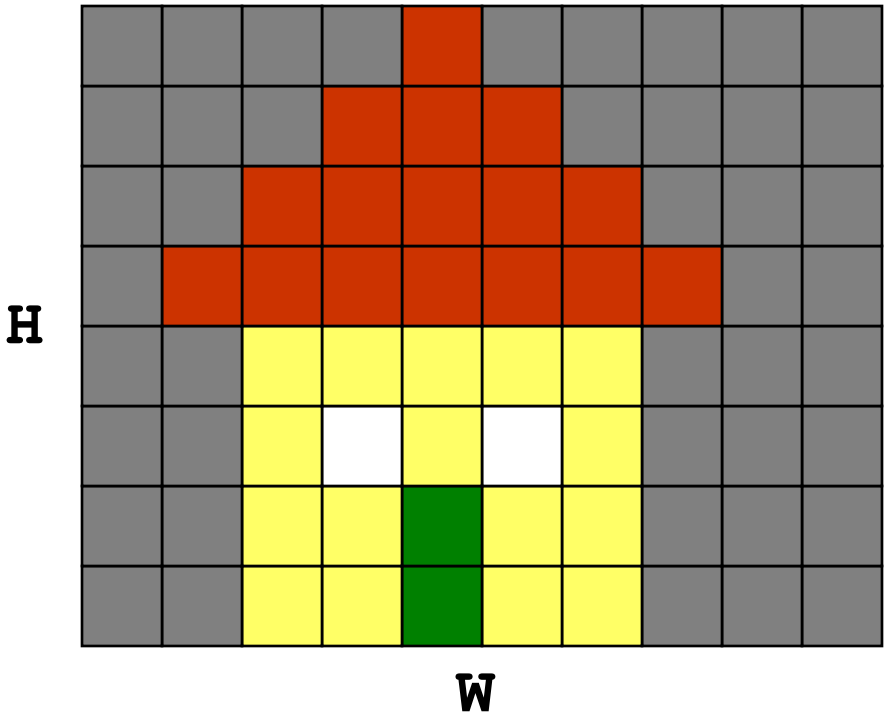
Grayscale (monochrome)

Every pixel is encoded with 8 or 10 bit (in most cases 8 bit). With 8 bit, the range is [0,255].



Color 1/4

Similarly to grayscale, most color sensors encode each pixel with 8 (or 10 bit) or 16 bit. There are several techologies/strategies but in most cases each pixel is encoded, as for greyscale, with 1 byte or 2 bytes.



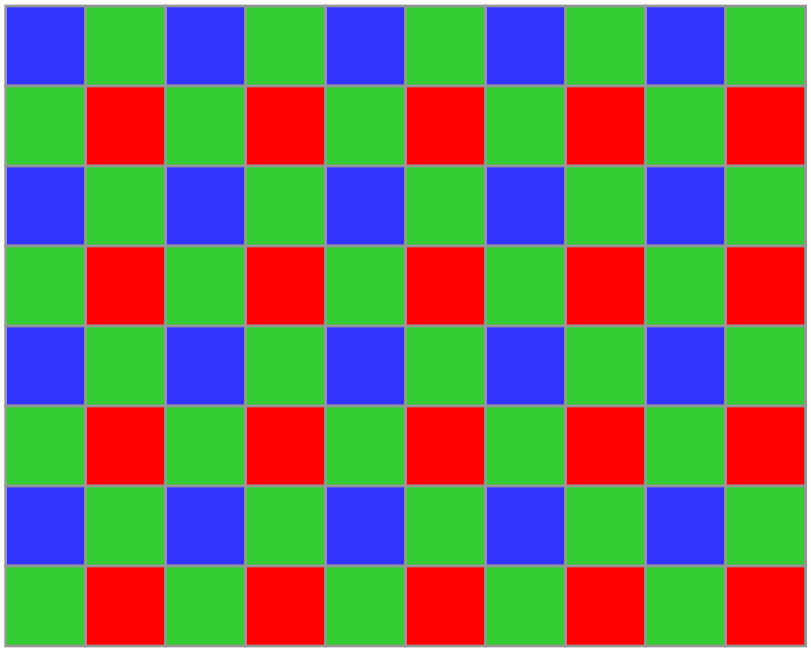
How to encode/obtain colors from 8 bit?

- Bayer pattern
- YUV/YCbCr (later)

Color 2/4 (Bayer 8 bit)

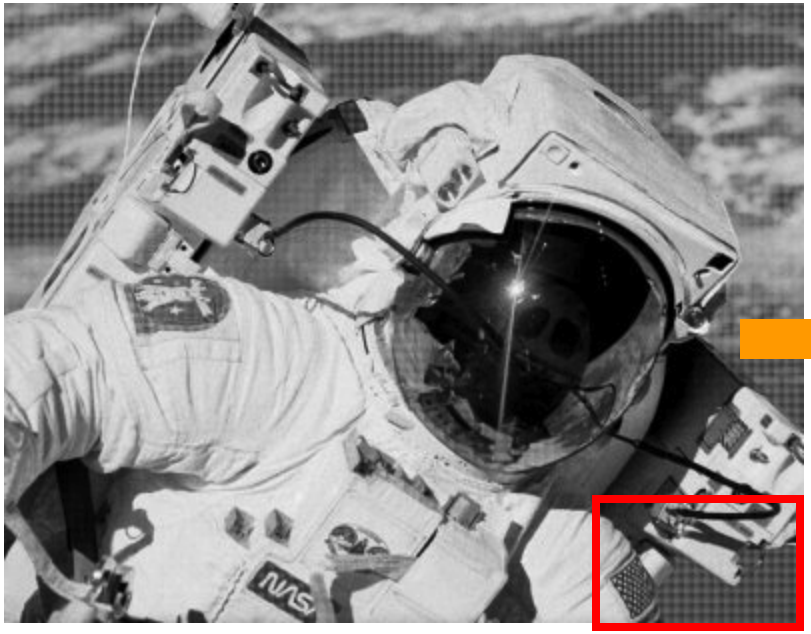
There is a filter for each pixel: Blue, Red and Green according to the followinng scheme (Bayer pattern)

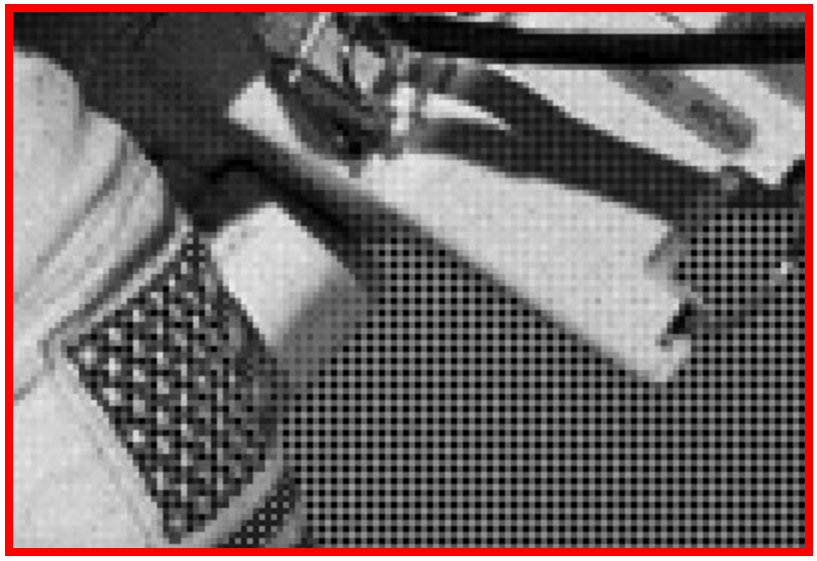
Bayer pattern



For each pixel, 2 out of 3 color channels are obtained by measn of interpolation. This fact implies a reduced spatial resolution with respec to to grayscale.
It is worth to note that, with this encoding scheme, the bandwidth is 1/3 (wrt to RGB encoding).

Color 3/4 (Bayer 8 bit)





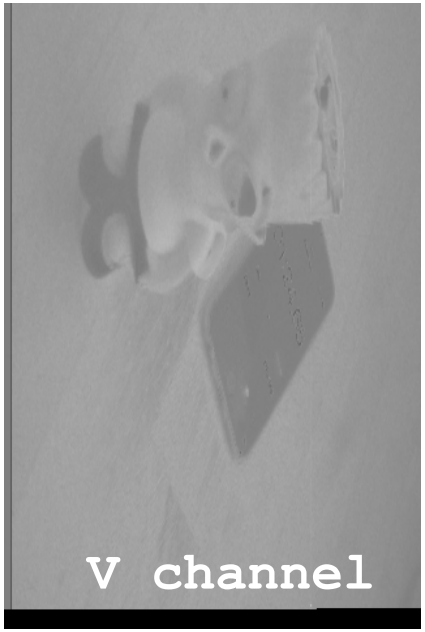
Example: *raw 8 bit image* with Bayer pattern (top left)

Color RGB image (top right)

Detail of the raw bayer pattern (on the left)

http://digital.ni.com/public.nsf/allkb/8AAAE9435D30CEB886256F80007A39AD

Color 4/4 (YUV or YCbCr 16 bit)



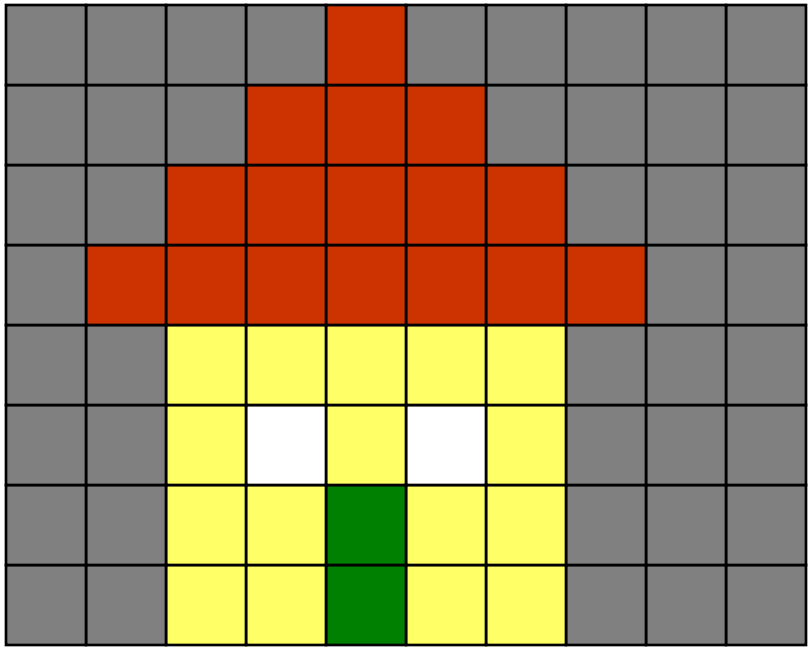
RGB image: more details in the next slides

Frame rate 1/2

The pixel are transmitted in *raster scan order* (from top-left to bottom right).

Streaming, at PIXEL_CLOCK, can't be paused!

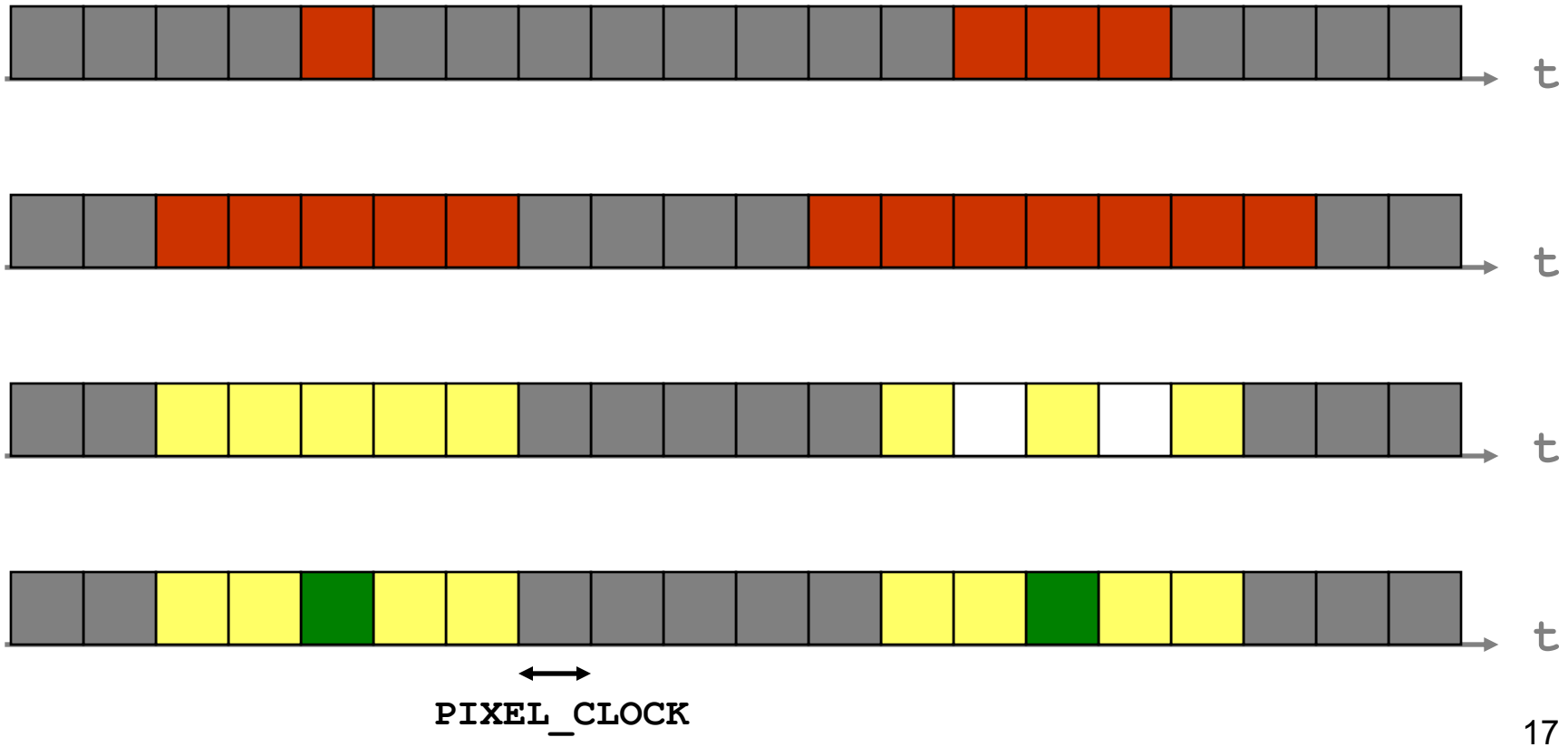
***frame per second* (fps): images/second**



Frame rate 2/2

In most cases pixel are processed at PIXEL_CLOCK

With the same frame rate, increasing the resolution yields to higher frequency (PIXEL_CLOCK gets smaller)



A common image streaming protocol with embedded control codes

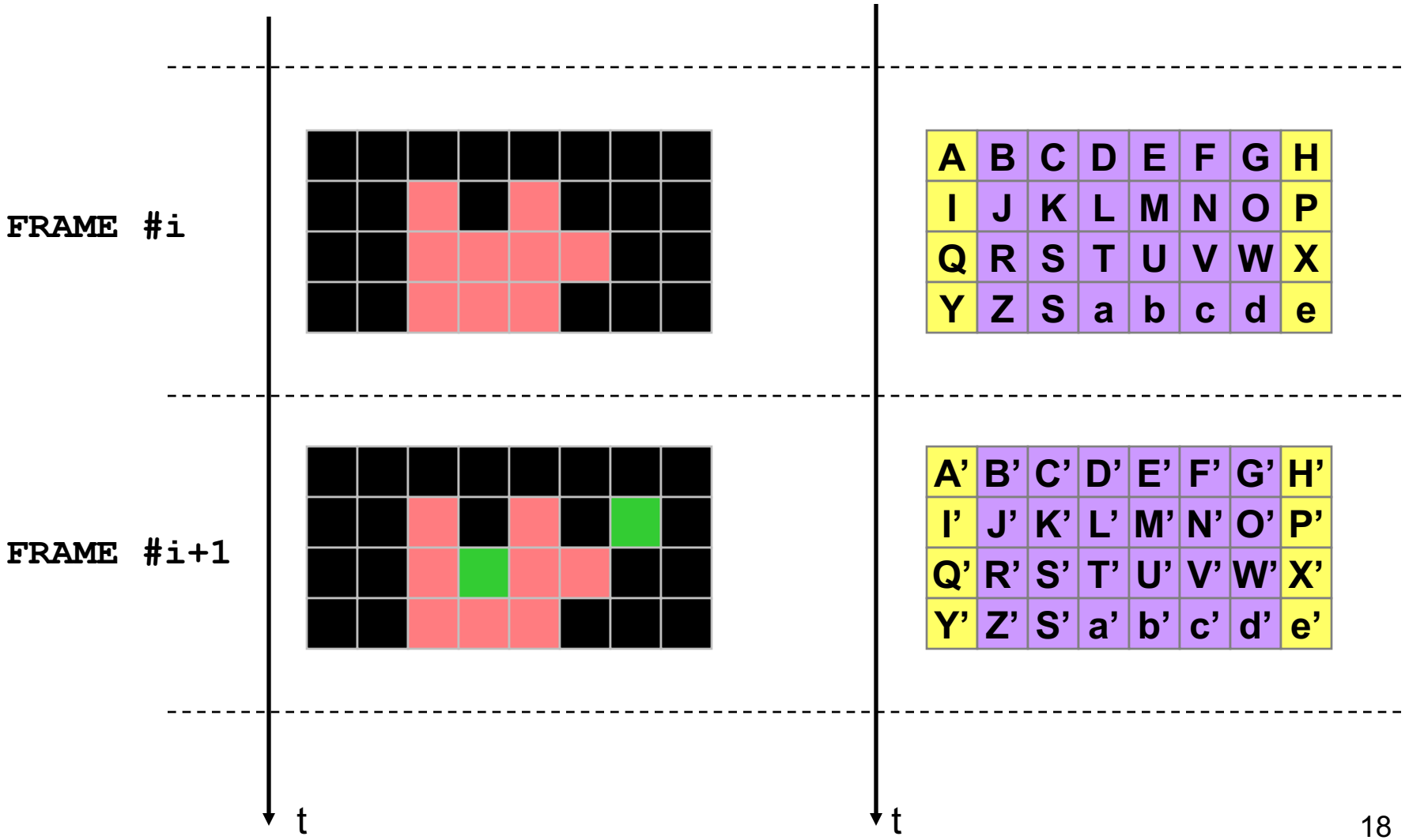
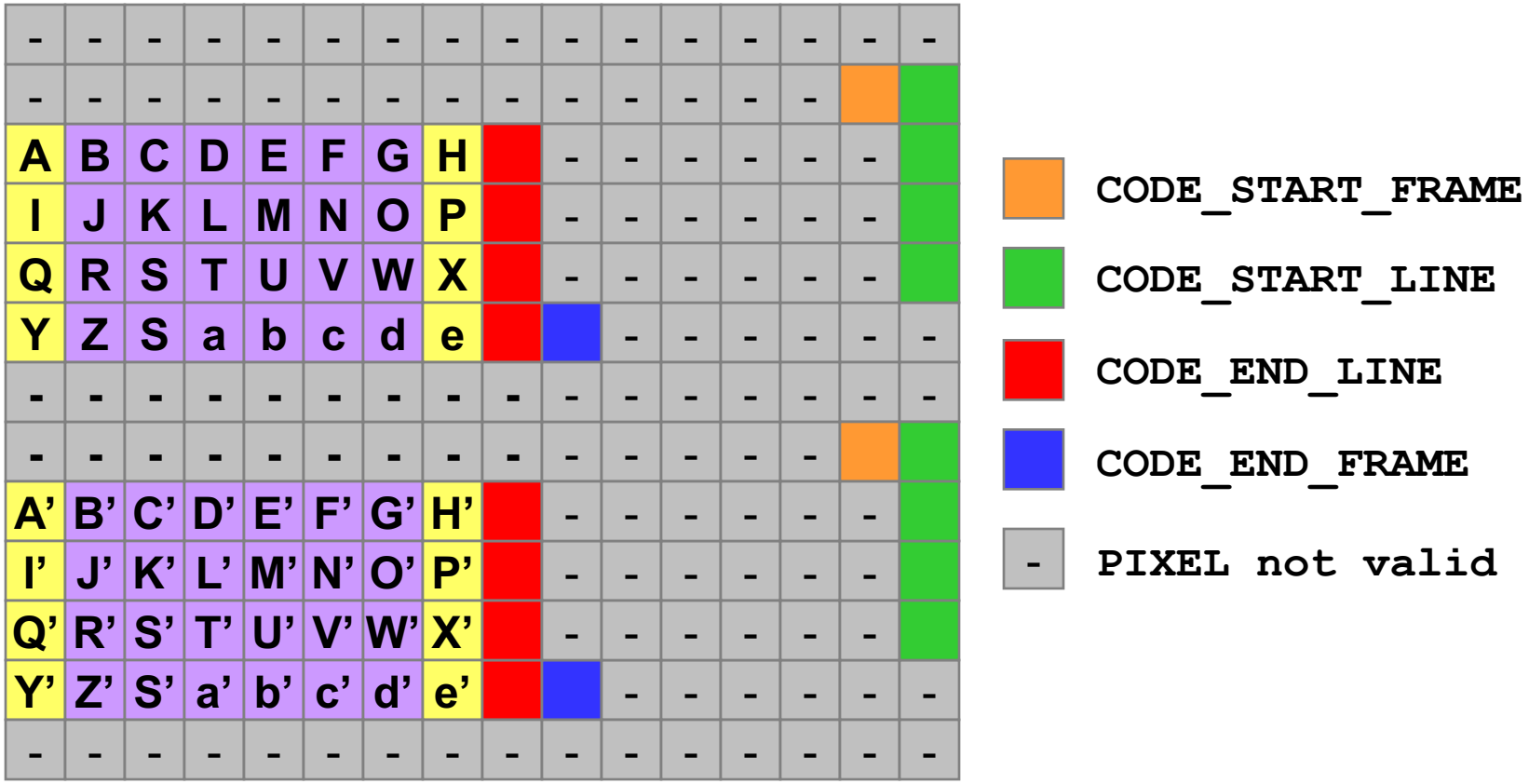
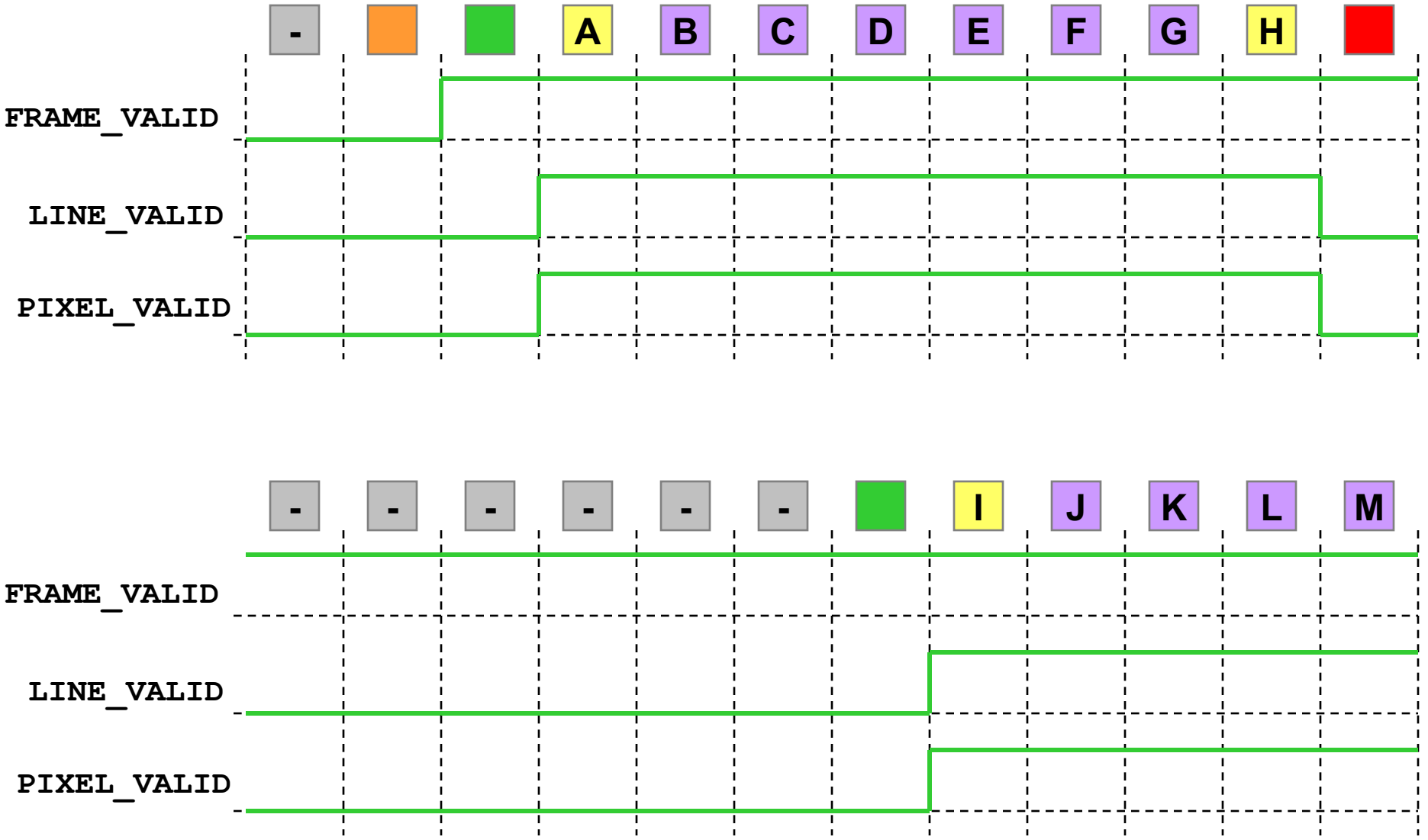


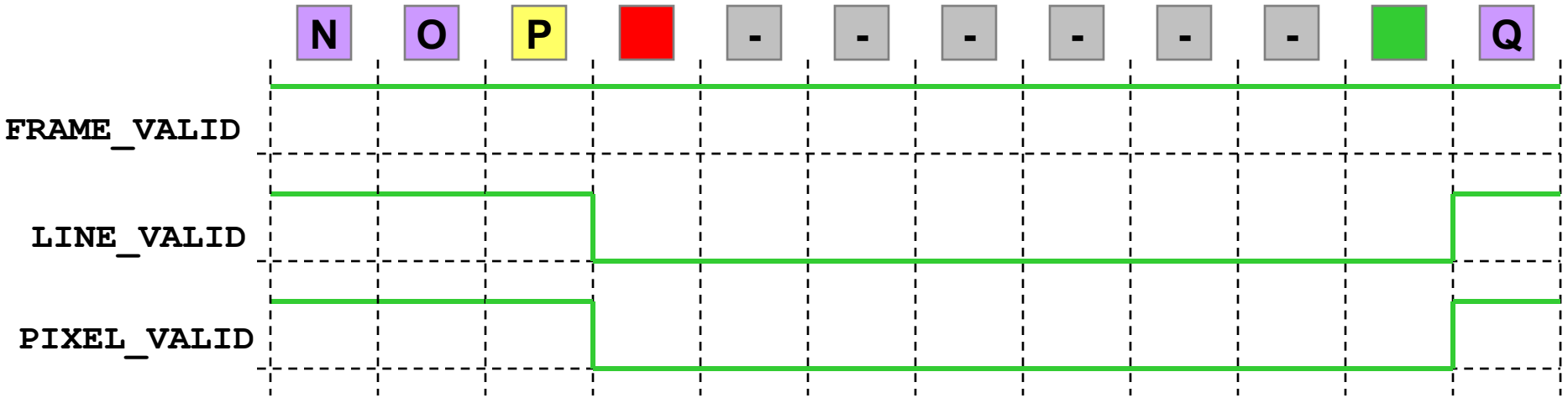
Image streaming contains embedded and reserved codes (*bytecode 0,1,2 e 3*) and aummy pixel dummy.
From this bytecodes, appropriate modules can infer the following signals: FRAME_VALID, LINE_VALID and also PIXEL_VALID

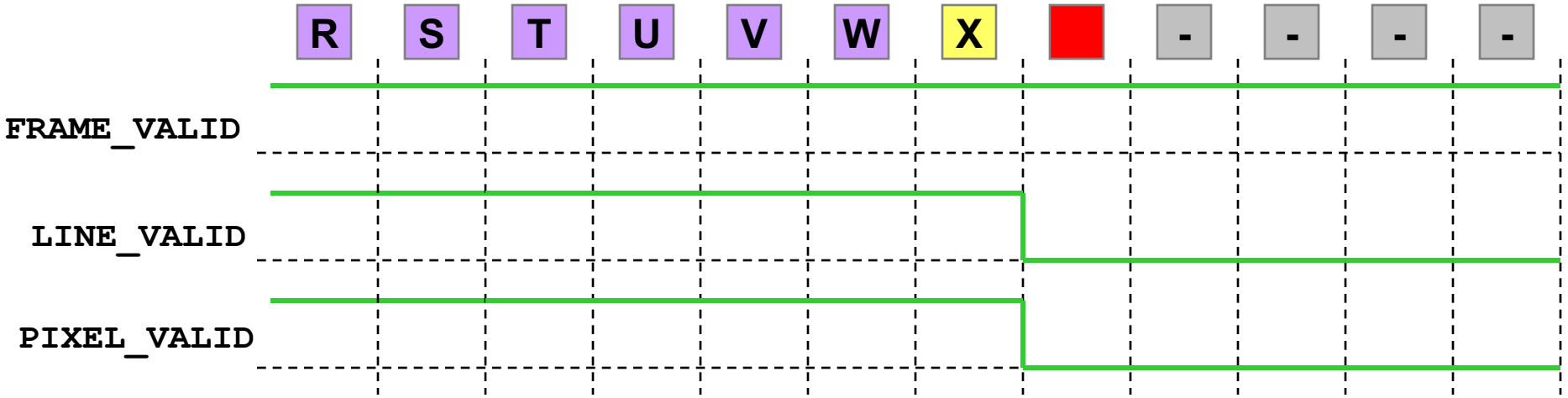


(Almost) standard control signals 1/3

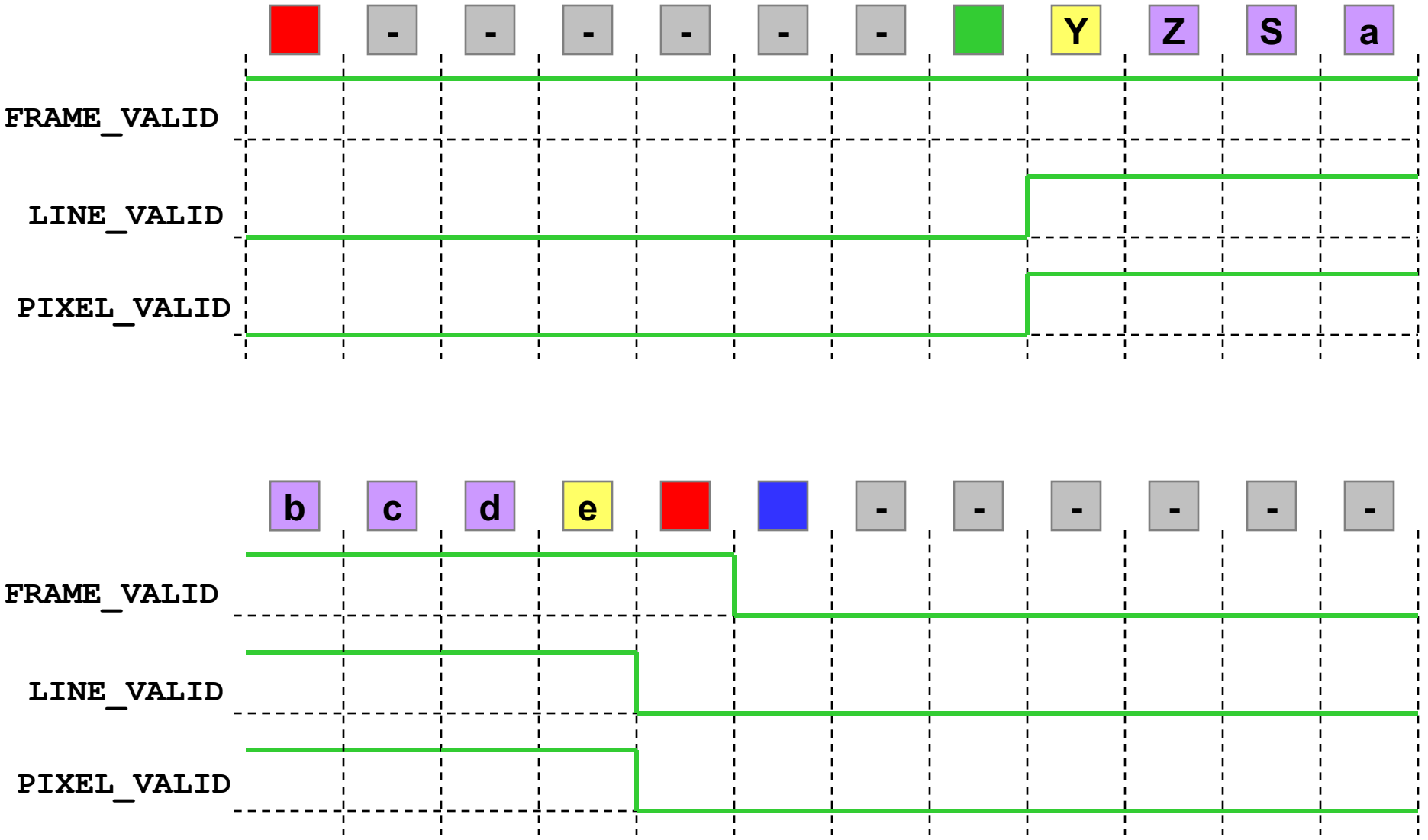


(Almost) standard control signals 2/3

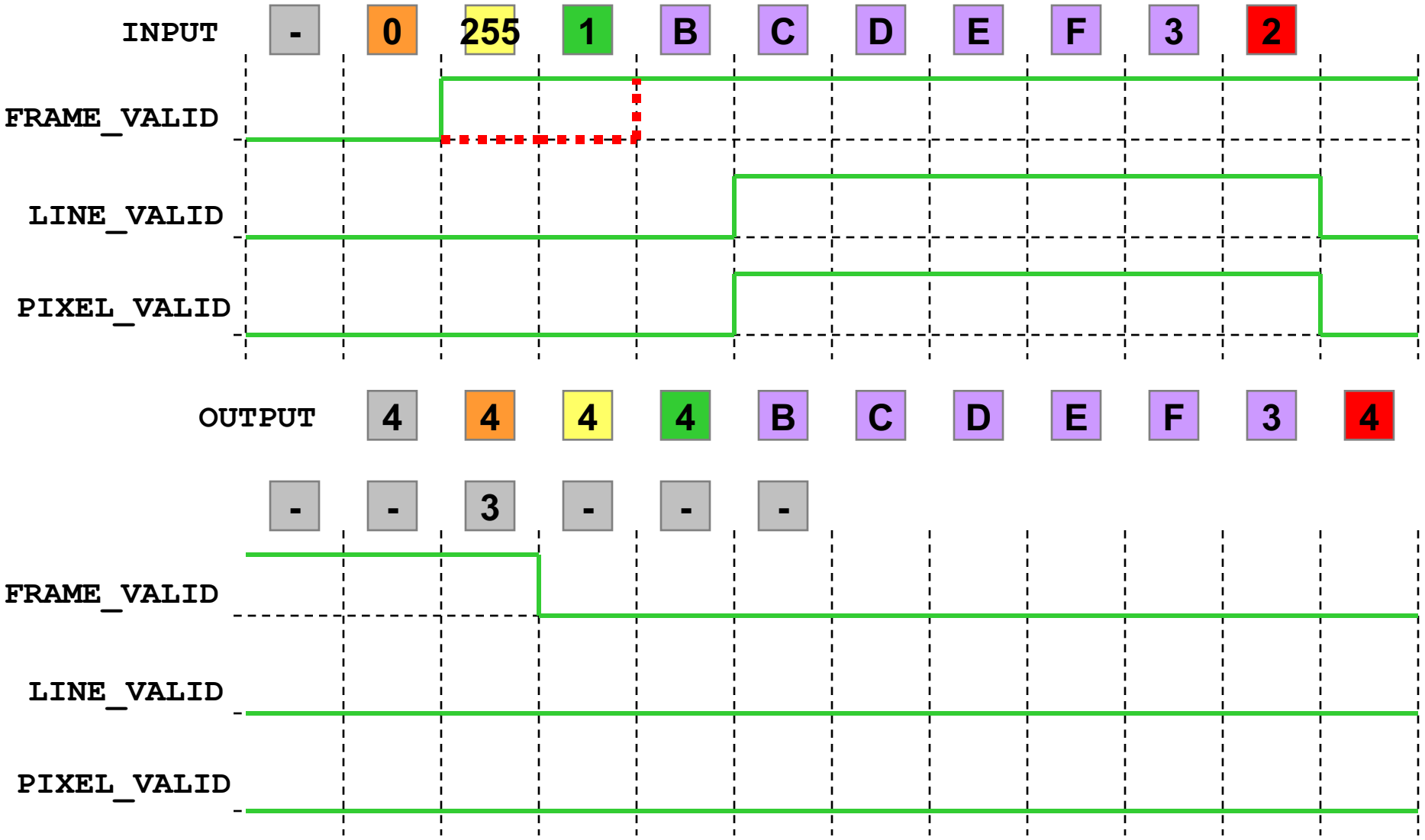




(Almost) standard control signals 3/3



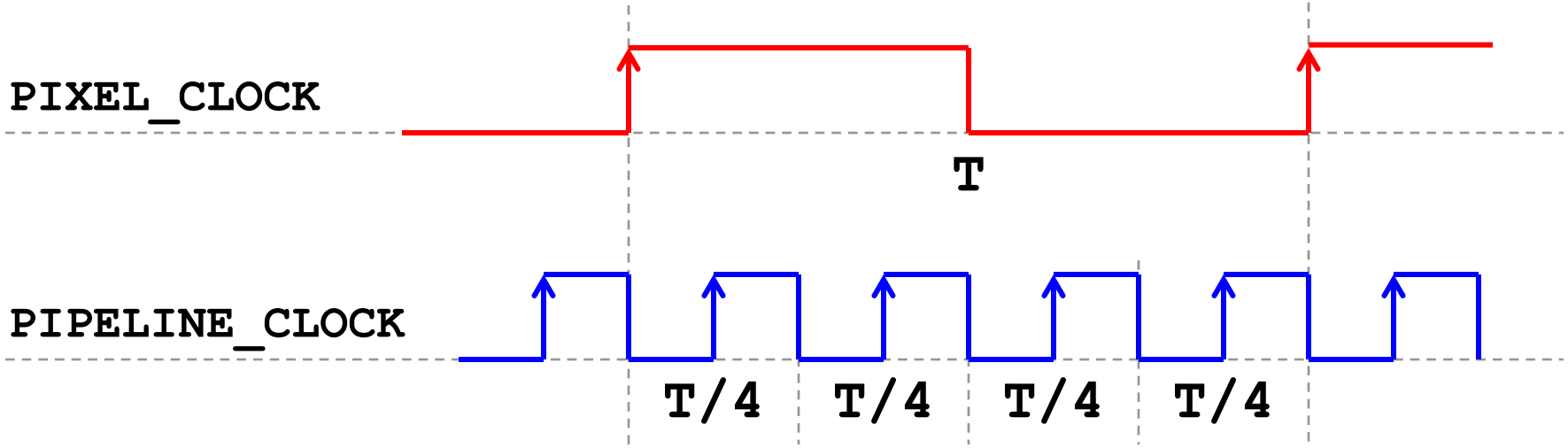
(Almost) standard control signals (1*/3)



Pixel clock e pipeline clock

In many cases, the image processing pipeline works at higher frequency, often referred to as PIPELINE_CLOCK, wrt PIXEL_CLOCK.

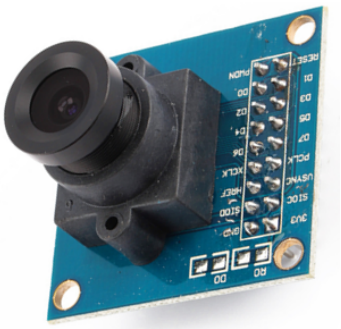
The two clocks are not always aligned (in general a simple FIFO with two clock domains would solve this issue).



OV 7670: specifications

Digital imaging sensor manufactured by Omnivision:

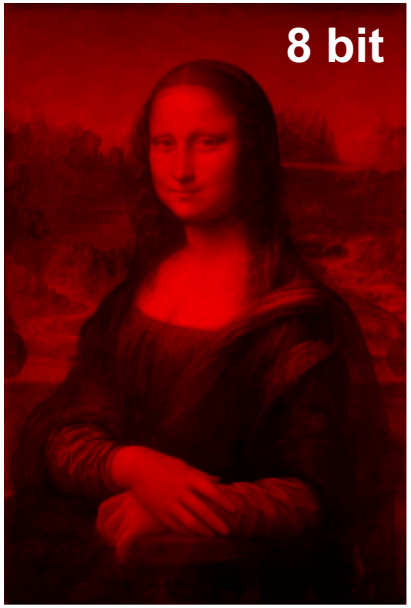
- Pixel size : 3.6 um x 3.6 um
- Resolution : 640x480 (VGA), 320x240 (QVGA), etc
- Frame rate : 30 fps
- Color format : RGB, YUV (4:2:2) and YCbCr (4:2:2)
- Scan mode : rolling
- Output : parallel (16 bit for YCbCr)
- Power supply : max 3.0 V
- Programming : I2C (7 bit address 0x21)

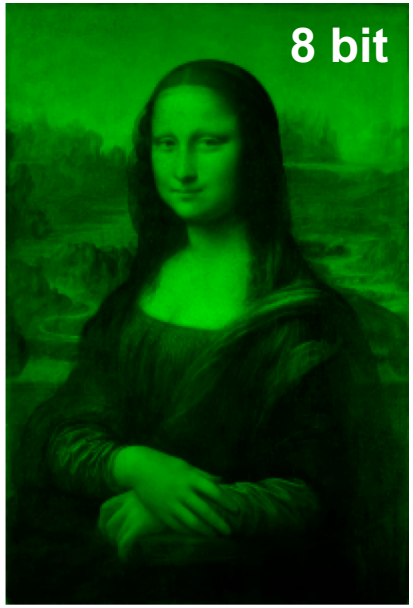


YUV or YCbCr (4:2:2) color encoding

- YcbCr and YUV enables a more compact encoding wrt RGB
- YUV/YCbCr vs RGB: 8+8 bit vs 8+8+8 per pixel
- Greyscale component is Luma (Y)
- Color is subsampled (horizontally in 4:2:2)
- CbCr and UV components: resolution is halved (W/2xH)
- In YCbCr or YUV the grayscale component is Luma (Y)
- Grayscale is extracted ignoring chroma components
- With YCbCr or YUV, with 8 bit datapath, each pixel requires more than 1 clock (e.g., 2 clocks for 4:2:2)
- In our setup, the OV7670 is configured for YUV 4:2:2
- Other sensors (e.g. Aptina/ONSEMI MT9v034C), use a different color encoding scheme based on Bayer pattern

https://msdn.microsoft.com/en-us/library/windows/desktop/dn424131(v=vs.85).aspx





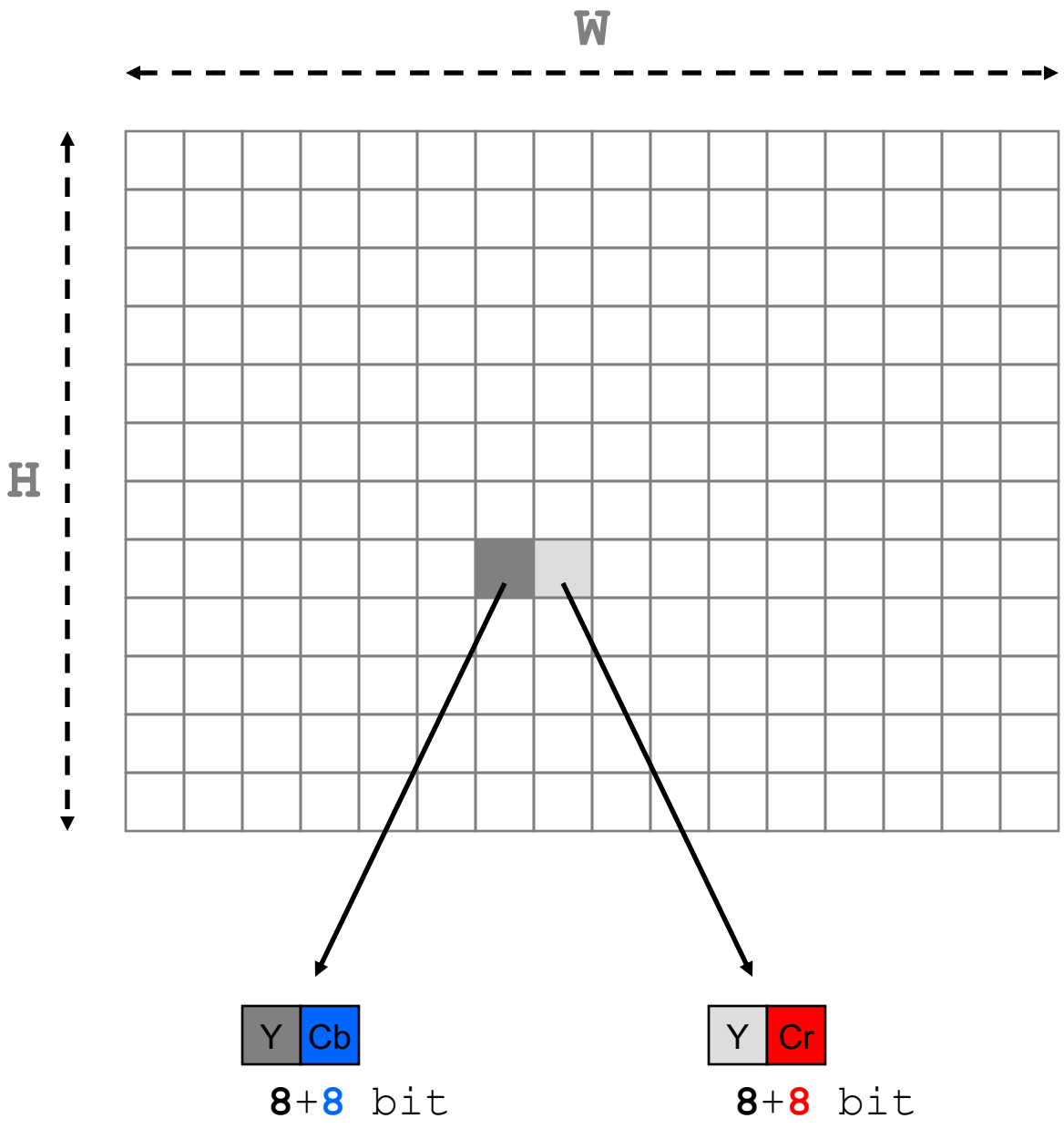
Original Red Green Blue

Original Luma Y Chroma Cb Chroma Cr

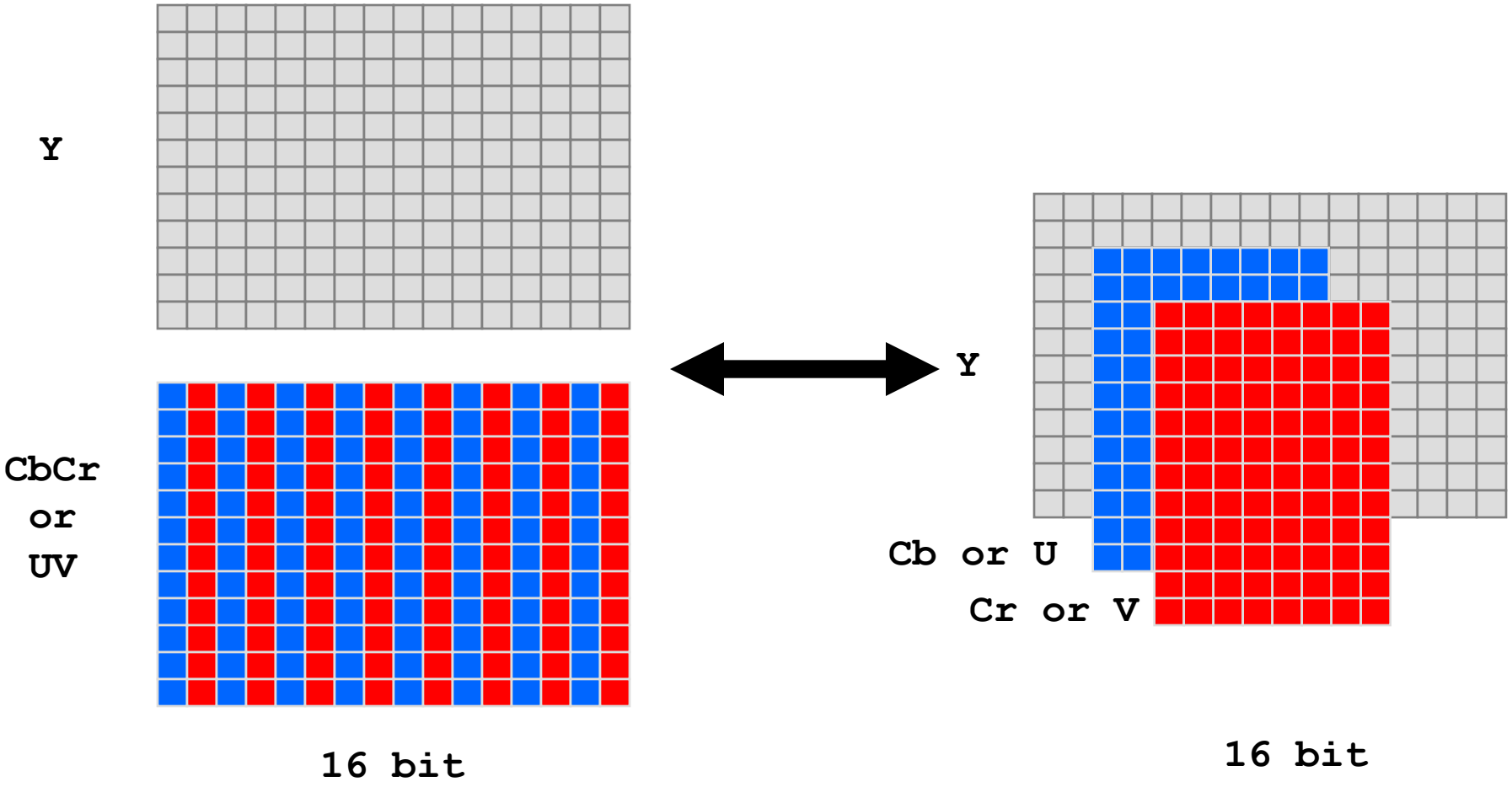
Alternative chroma components encoding for YCbCr or YUV:

- **4:4:4 (no subsampling) – 24 bits**
- **4:4:0 (vertical subsampling) – 16 bits**
- **4:2:0 (vertical and horizontal subsampling) 12 bits**

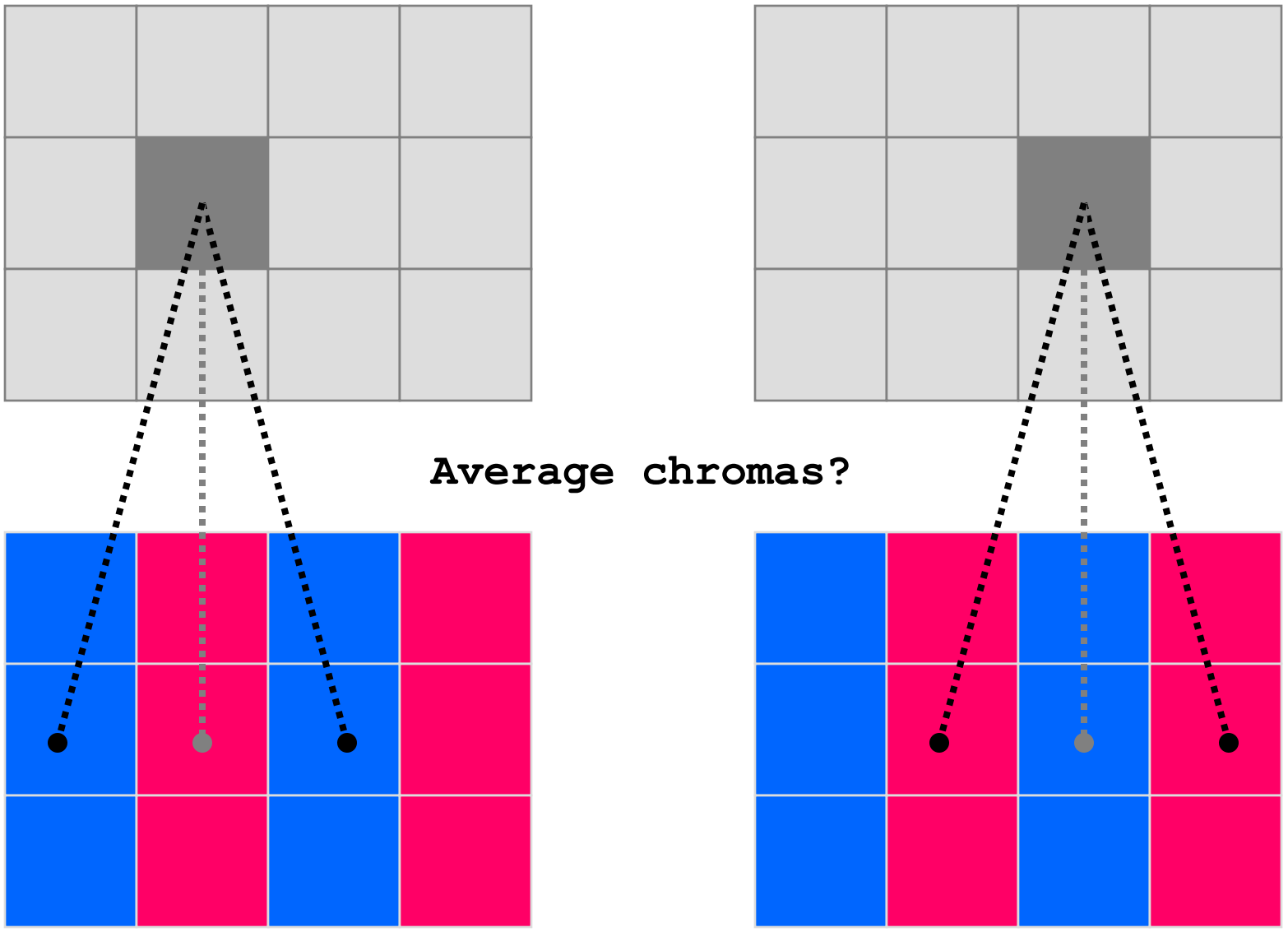
Subsampling mode	Horizontal downscale	Vertical downscale	Bits per pixel*
4:4:4	1x	1x	24
4:2:2	**2x**	**1x**	**16**
4:4:0	1x	2x	16
4:2:0	2x	2x	12


W
H
Y
Cb
8+8 bit
Y
Cr
8+8 bit

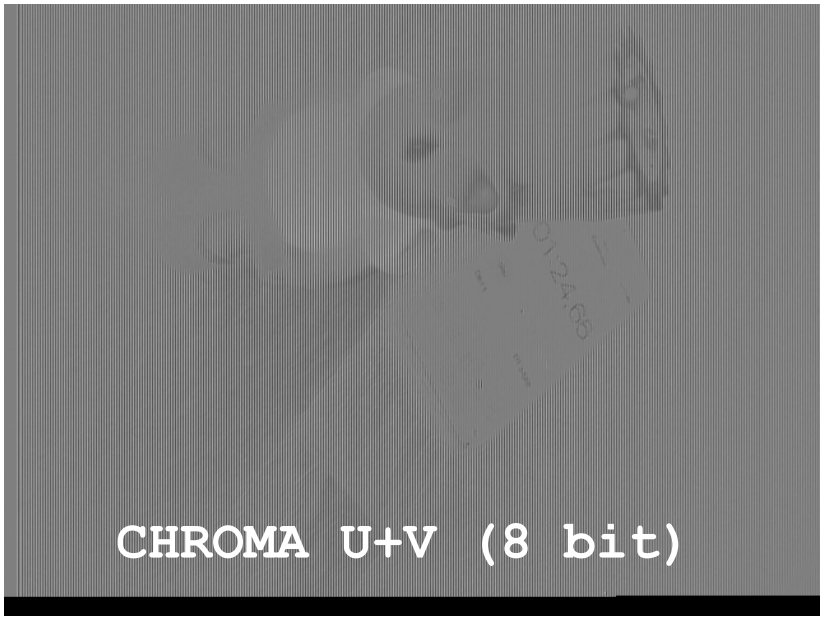
YCbCr or YUV 4:2:2



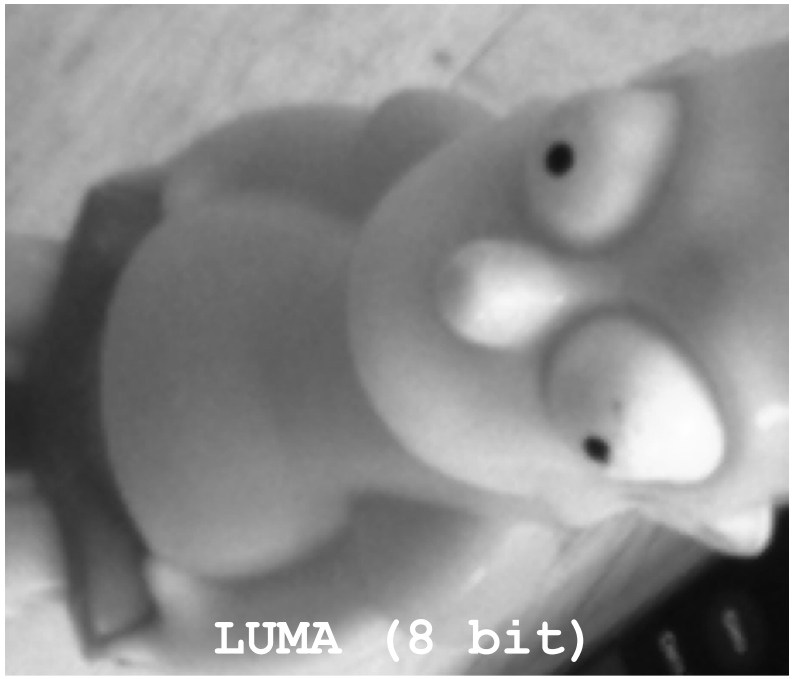
YCbCr or YUV 4:2:2

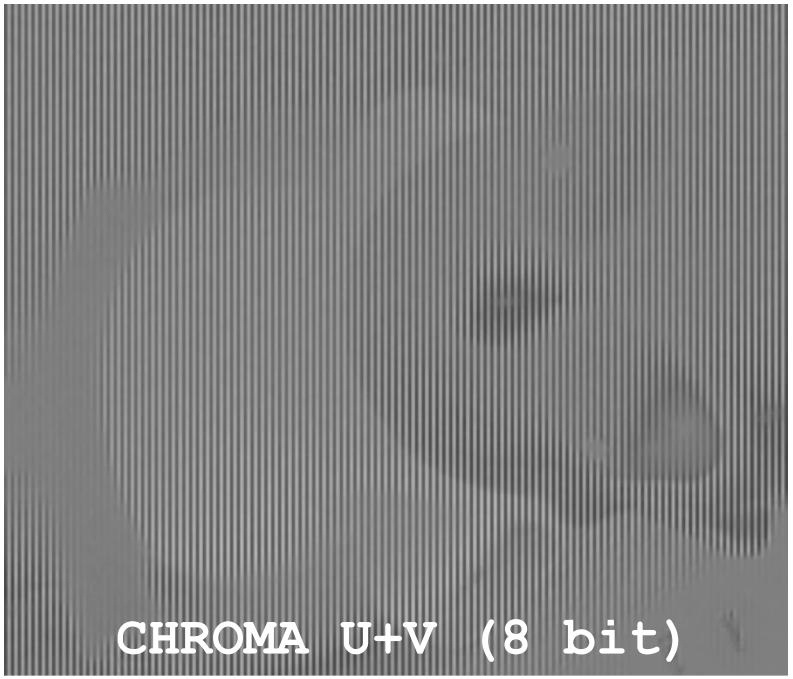



LUMA (8 bit)


CHROMA U+V (8 bit)

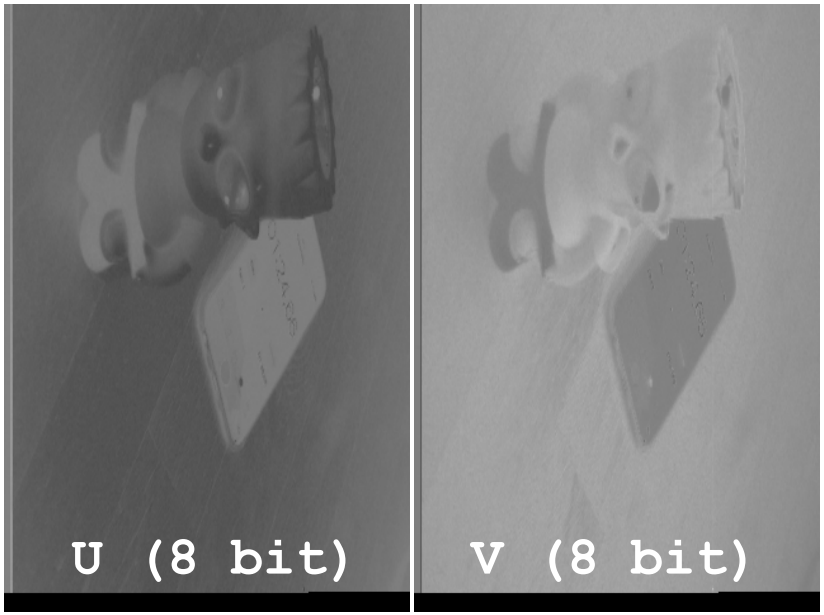

RGB


LUMA (8 bit)


CHROMA U+V (8 bit)


RGB

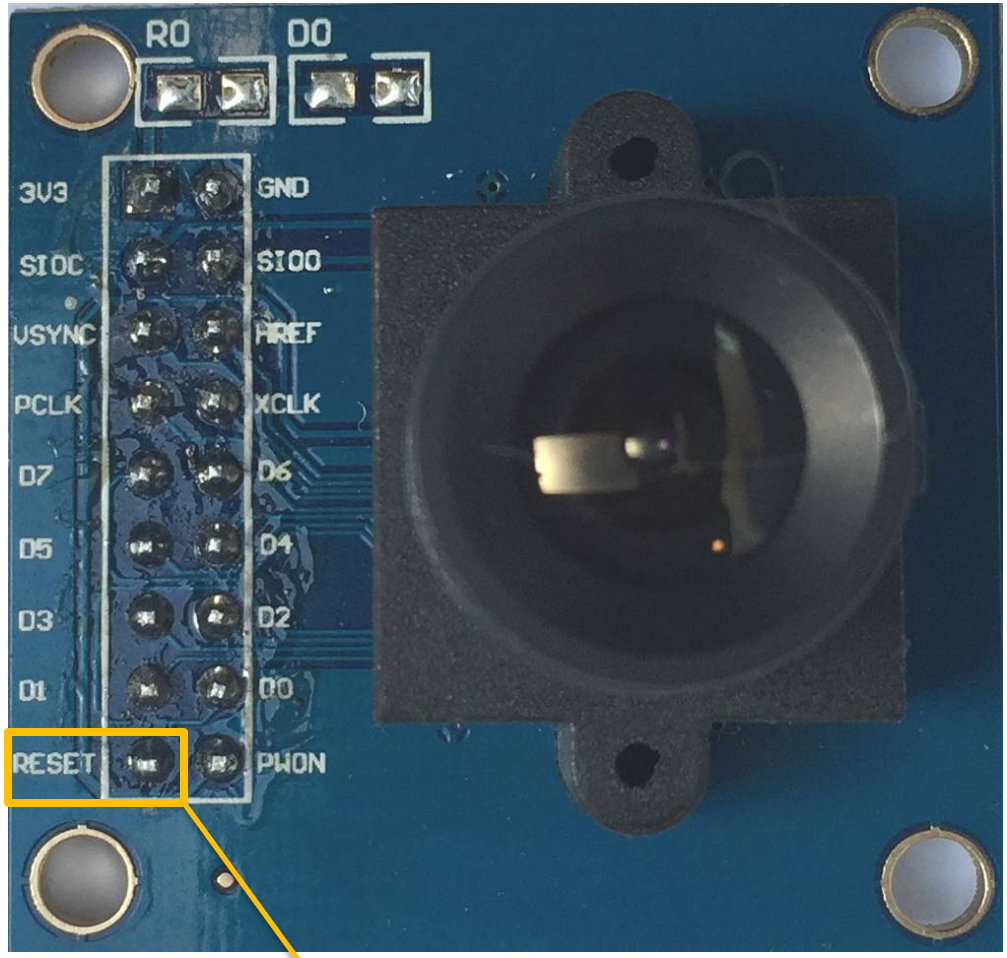

LUMA (8 bit)


U (8 bit)
V (8 bit)


RGB

OV 7670: pinout

3V3	GND
SIOC	SIOD
VSYNC	HREF
PCLK	XCLK
D7	D6
D5	D4
D3	D2
D1	D0
RESET*	PWDN



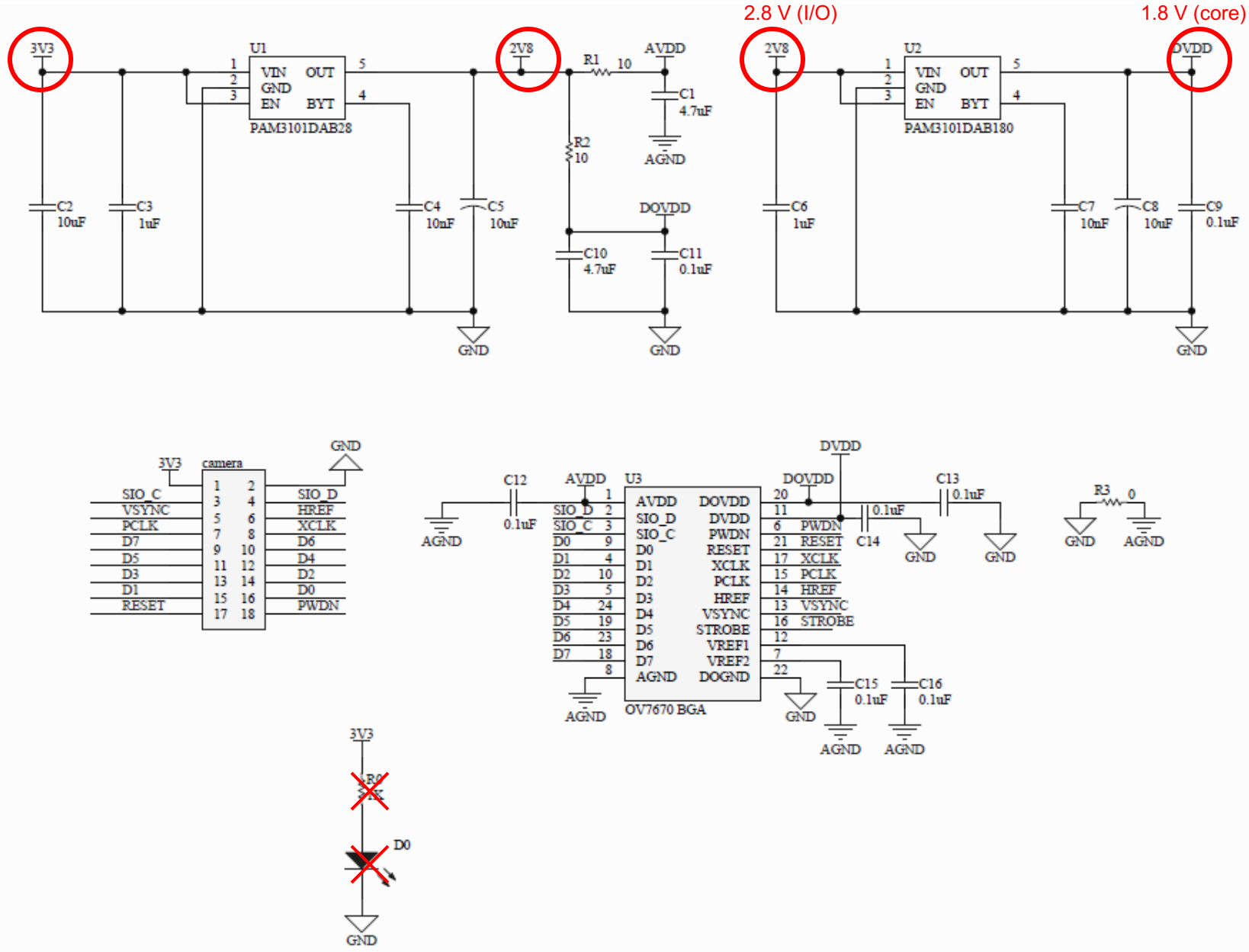
RESET is active low

Internal pull-down resistor
PWDN is active high

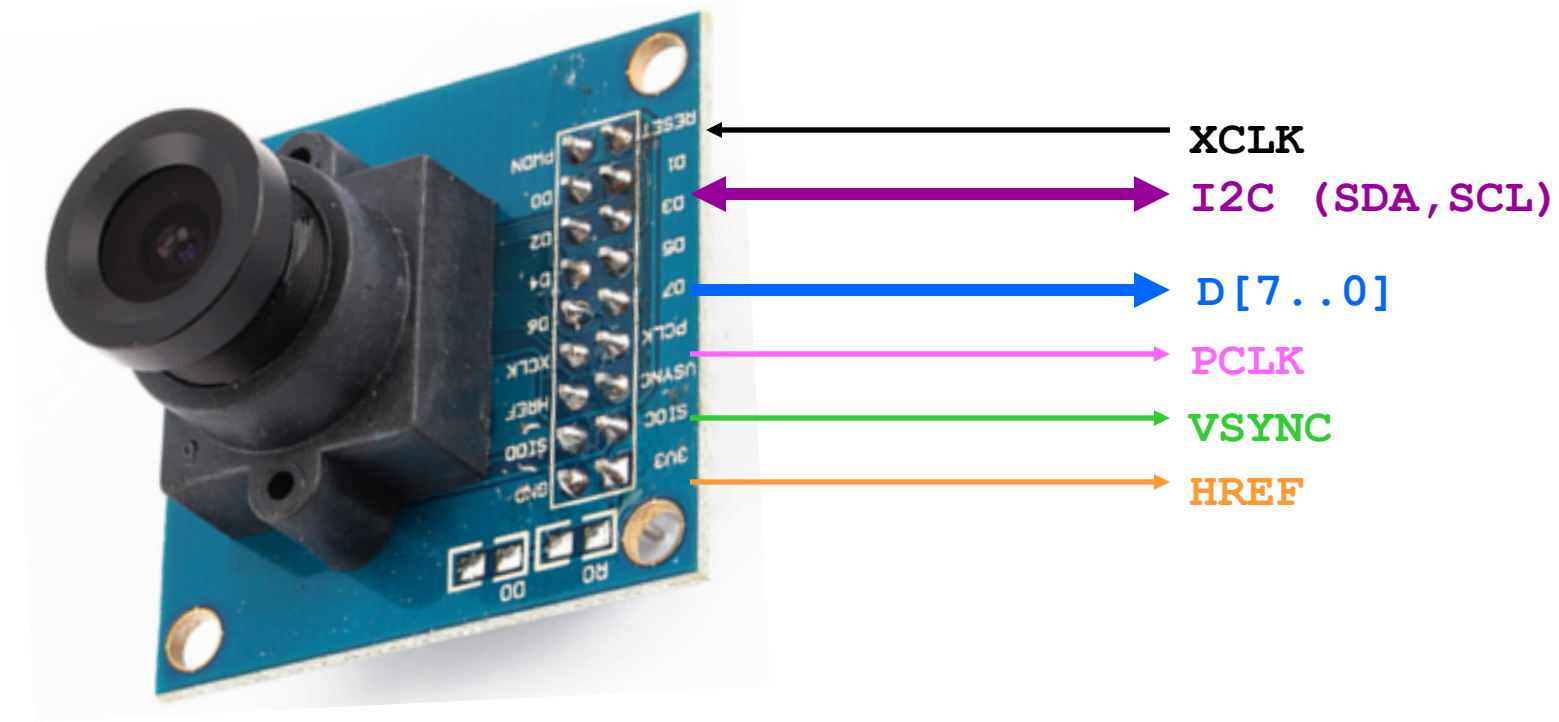
Since it is active low,
*it is renamed **RESET****

OV 7670: pin description

Pin	Type	Description
VDD/3V3	Supply	Power supply 3.3V
GND	Supply	Ground level
SIOC/SCL	**Input**	**I2C clock**
SIOD/SDA	**Input/Output**	**I2C data**
VSYNC	**Output**	**Vertical synchronization**
HREF	**Output**	**Horizontal synchronization**
PCLK	**Output**	**Pixel clock**
XCLK	**Input**	**System clock**
D0-D7	**Output**	**Video parallel output**
RESET*	Input	Reset (Active low)
PWDN	Input	Power down (Active high)

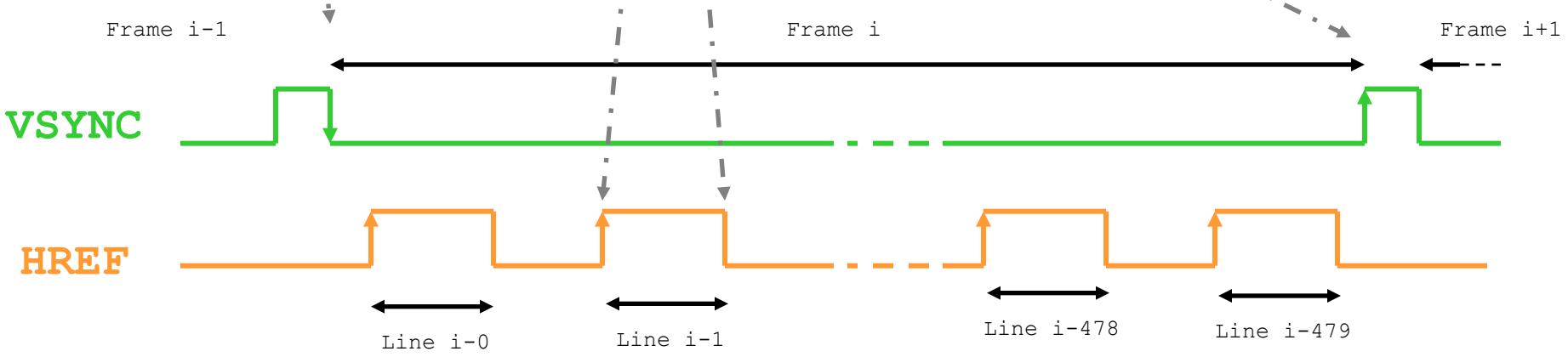

2.8 V (I/O)
1.8 V (core)
3V3
U1
VIN
GND
EN
OUT
BYT
PAM3101DAB28
2V8
R1
10
AVDD
C1
4.7uF
AGND
R2
10
DOVDD
C2
10uF
C3
1uF
C4
10nF
C5
10uF
C10
4.7uF
C11
0.1uF
GND
U2
PAM3101DAB180
DVDD
C6
1uF
C7
10nF
C8
10uF
C9
0.1uF
camera
SIO_C
VSYNC
PCLK
D7
D5
D3
D1
RESET
SIO_D
HREF
XCLK
D6
D4
D2
D0
PWDN
C12
0.1uF
U3
AVDD
SIO_D
SIO_C
D0
D1
D2
D3
D4
D5
D6
D7
AGND
DOVDD
DVDD
PWDN
RESET
XCLK
PCLK
HREF
VSYNC
STROBE
VREF1
VREF2
DOGND
OV7670 BGA
C13
0.1uF
C14
0.1uF
R3
0
C15
0.1uF
C16
0.1uF
R0
D0

OV 7670: functional description



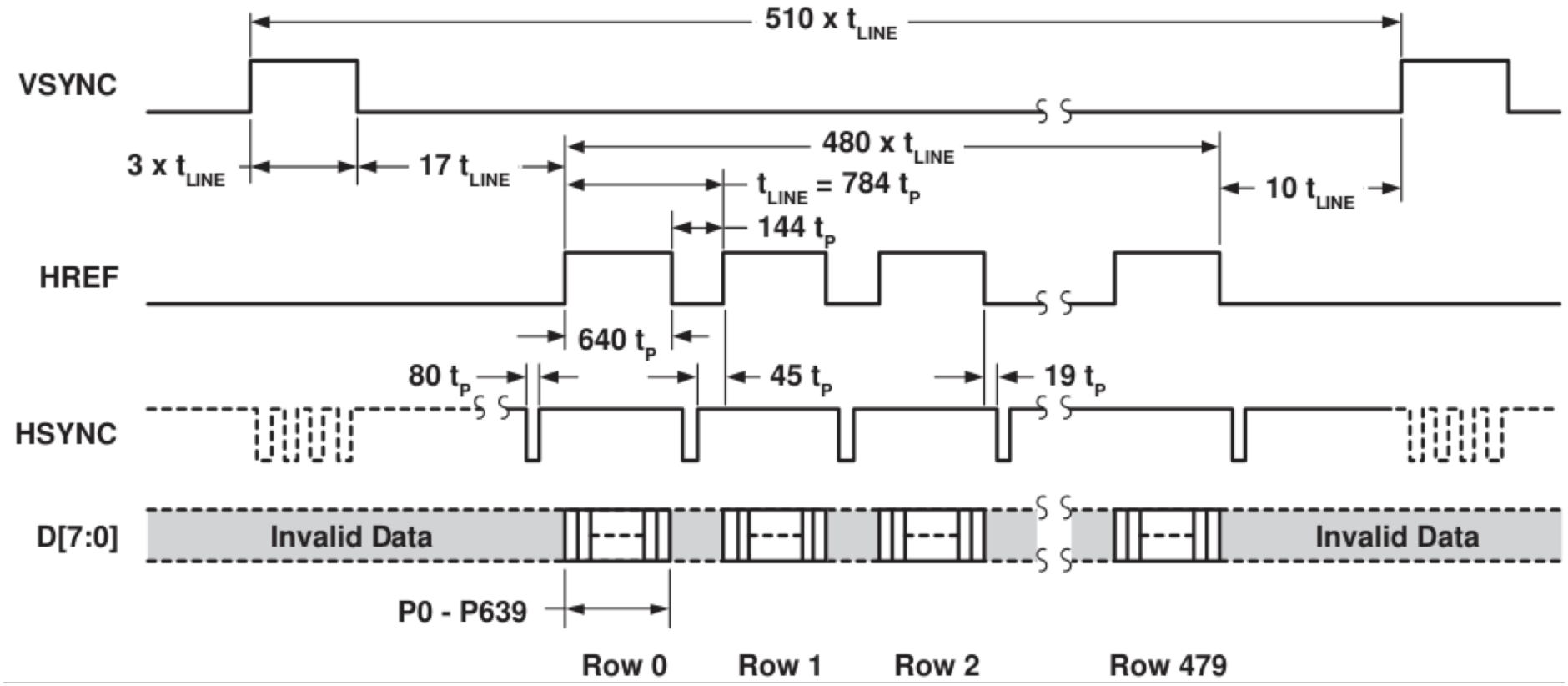
- The OV 7670 takes as input the XCLK clock signal (min frequency 10 MHz, max 48 MHz)
- The output pixel clock (PCLK) is programmable (I2C)
- The synchronous (PCLK) 8 bit data output D[7..0] and VSYNC and HREF signal encode the image content

- The falling edge of VSYNC encodes a new frame
- The rising edge of VSYNC encodes the end of a frame
- The rising edge of HREF encodes the start of new line
- The falling edge of HREF encodes the end of line



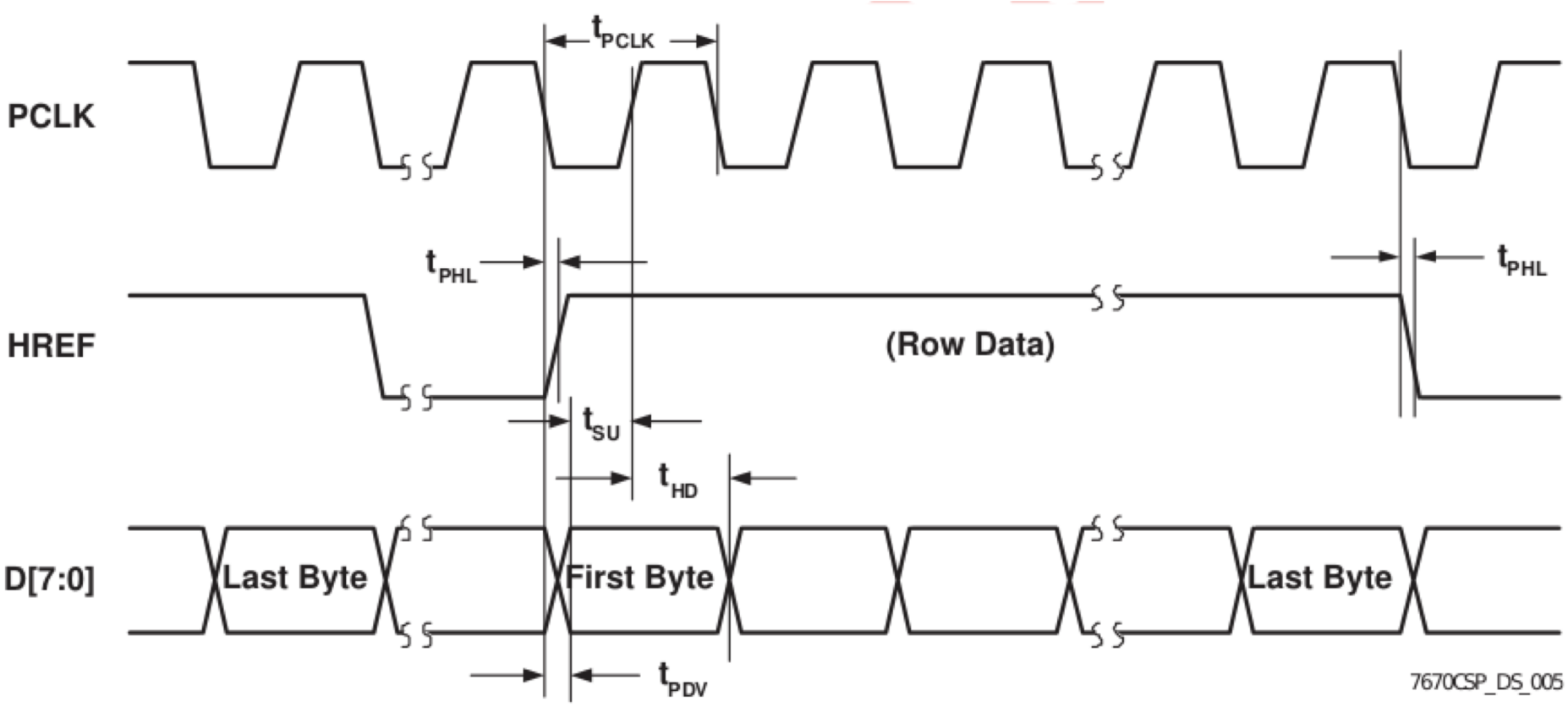
One byte is not a pixel! See YCbCr/YUV section

OV 7670: detailed waveforms 1/2



http://embeddedprogrammer.blogspot.it/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

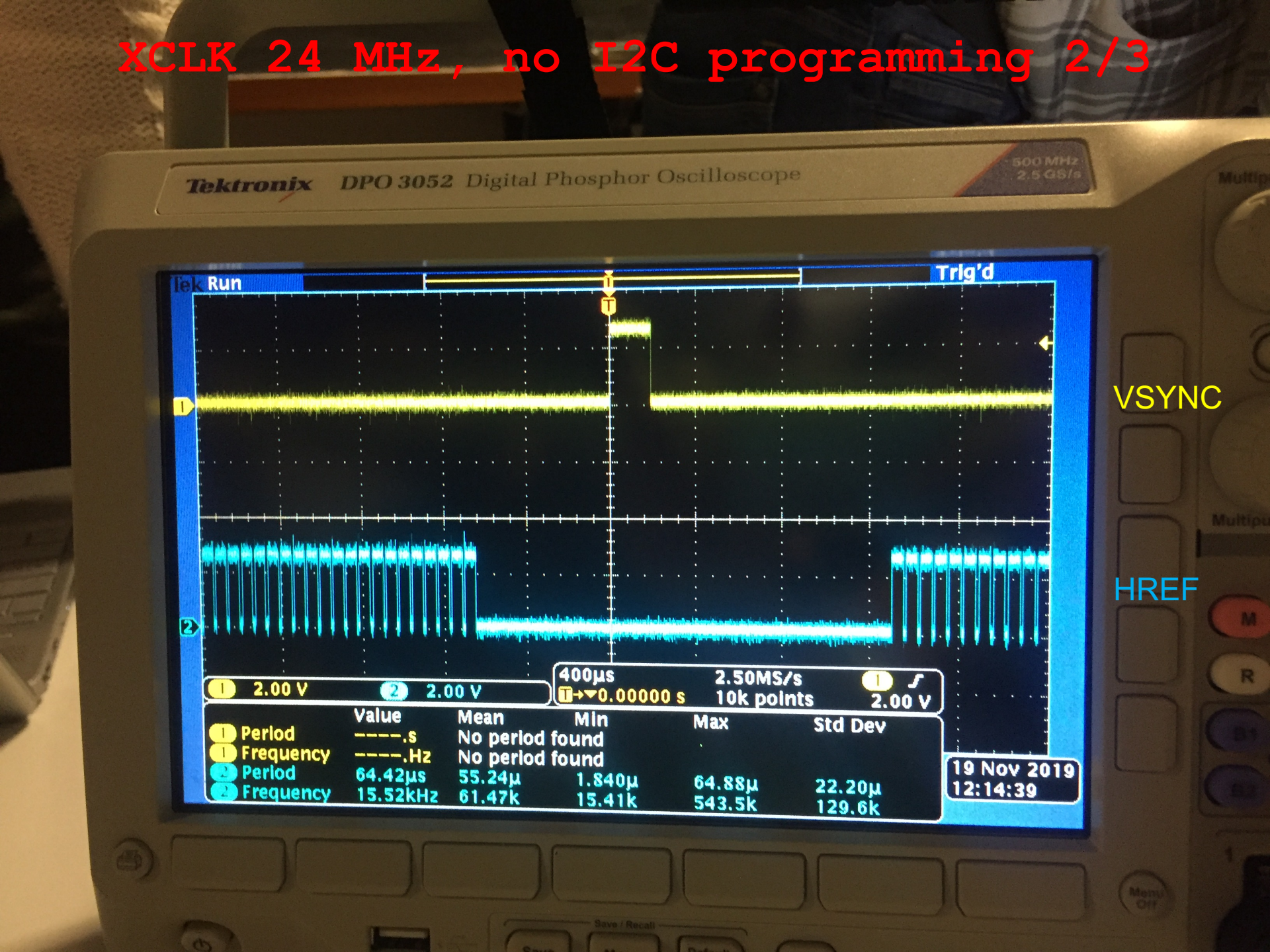
OV 7670: detailed waveforms 2/2



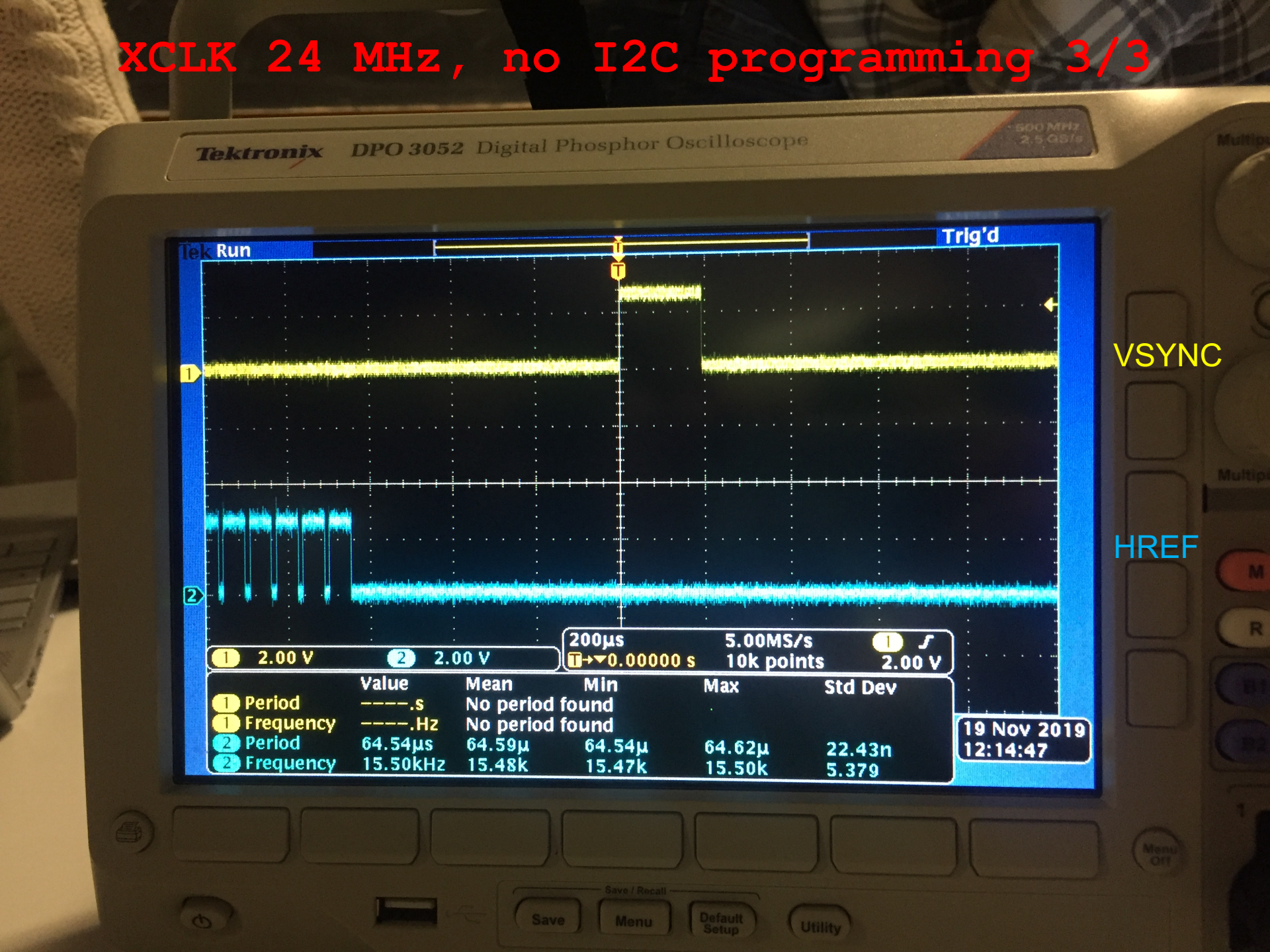
http://embeddedprogrammer.blogspot.it/2012/07/hacking-ov7670-camera-module-sccb-cheat.html

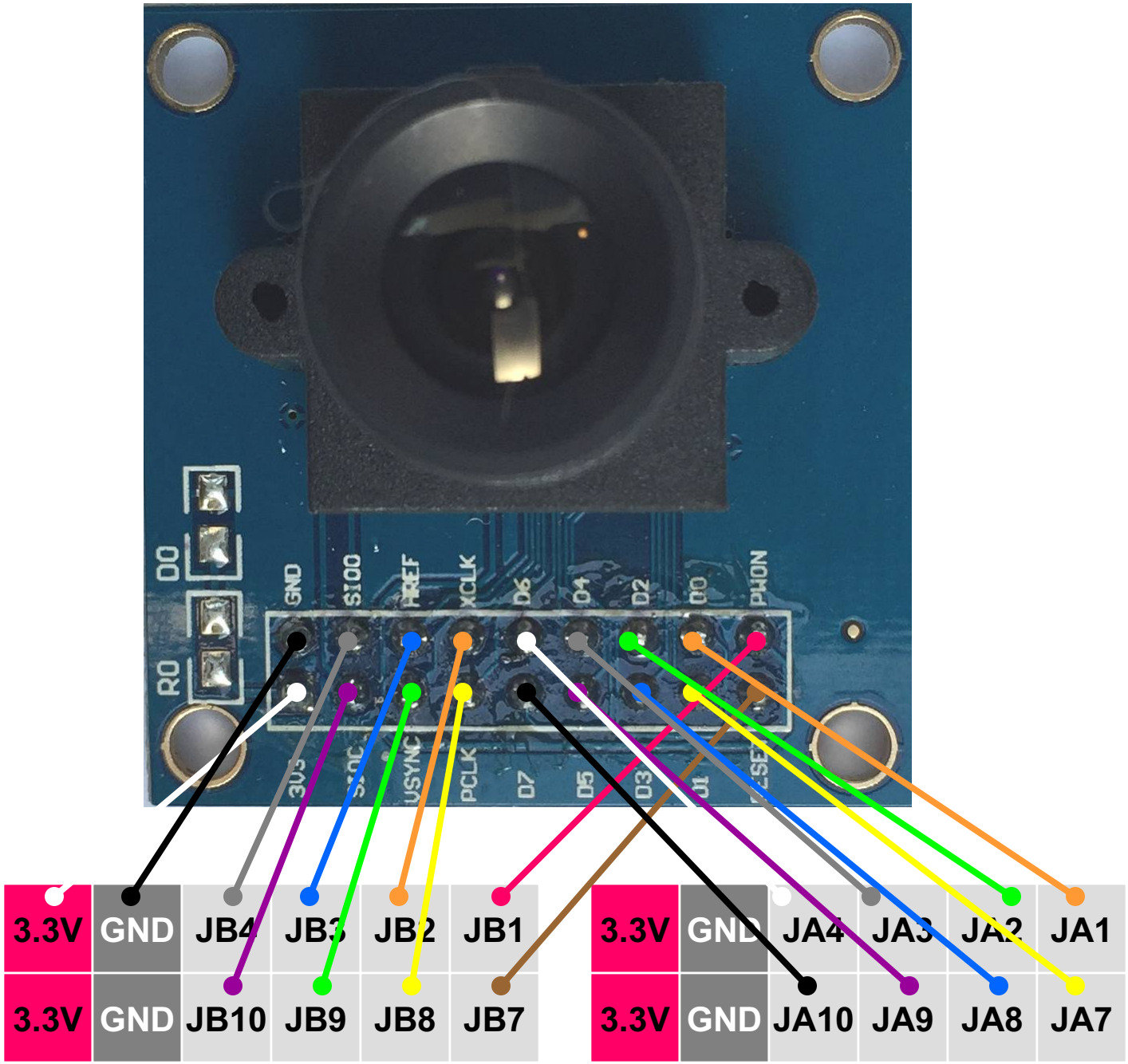
XCLK 24 MHz, no I2C programming 1/3

XCLK 24 MHz, no I2C programming 2/3



XCLK 24 MHz, no I2C programming 3/3



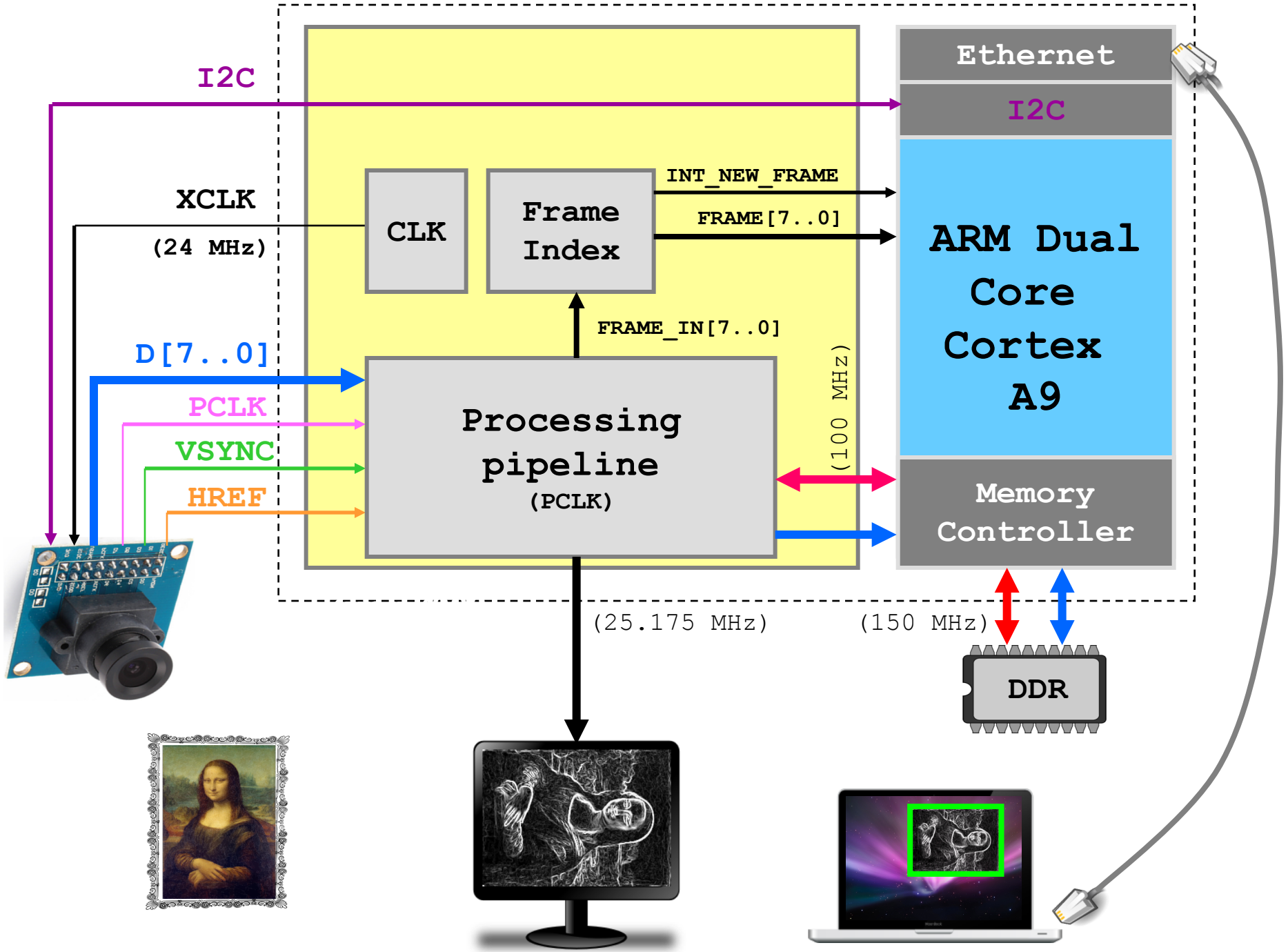

3.3V
GND
JB4
JB3
JB2
JB1
3.3V
GND
JB10
JB9
JB8
JB7
3.3V
GND
JA4
JA3
JA2
JA1
3.3V
GND
JA10
JA9
JA8
JA7

PMOD E PIN
JE3
JE2

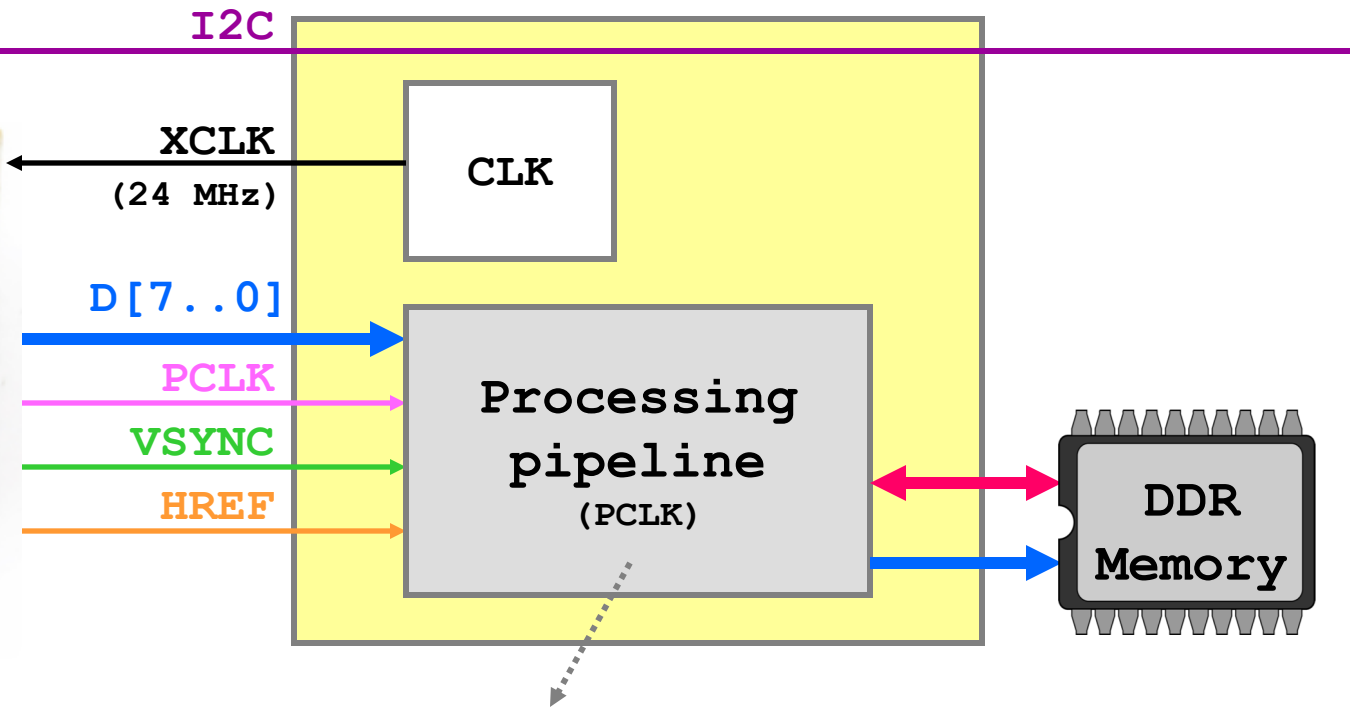
OV 7670	PMOD A,B PIN	ZYNQ PIN
3.3 V	VCC/JB6	-
GND	GND/JB5	-
SIOD/SDA	JB4	W8
SIOC/SCL	JB10	V8
HREF	JB3	V10
VSYNC	JB9	V9
XCLK	JB2	W11
PCLK	JB8	W10
PWDN	JB1	W12
RESET*	JB7	V12
D0	JA1	Y11
D1	JA7	AB11
D2	JA2	AA11
D3	JA8	AB10
D4	JA3	Y10
D5	JA9	AB9
D6	JA4	AA9
D7	JA10	AA8

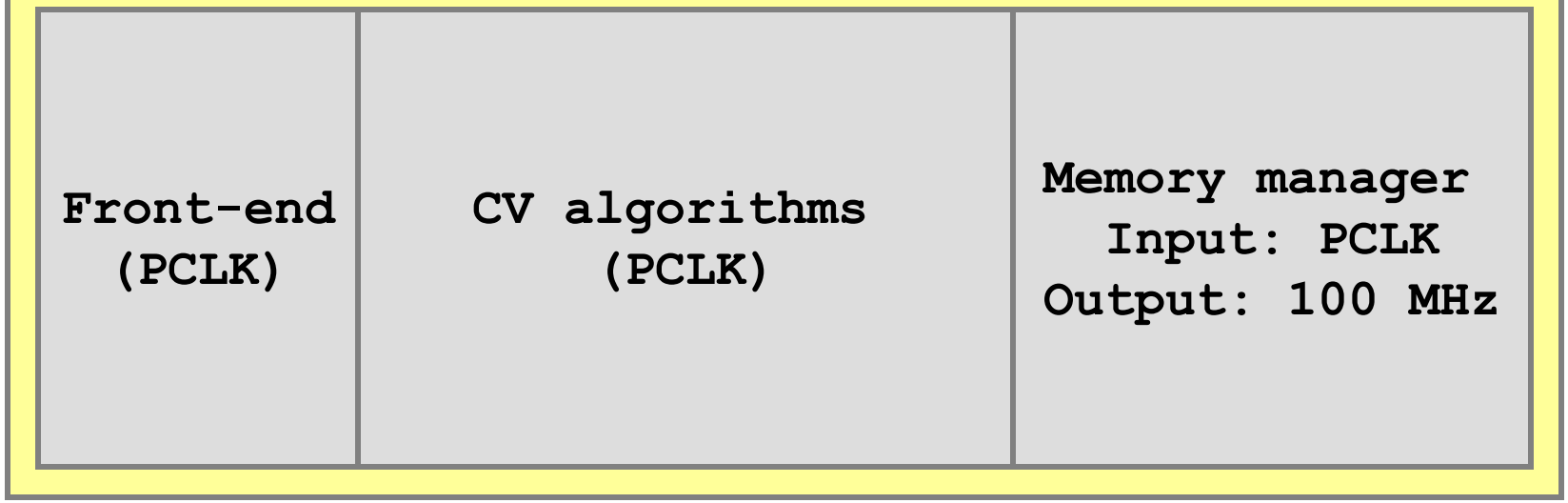
Constraints (Vivado):

```
set_property IOSTANDARD LVCMOS33 [get_ports XCLK]
set_property PACKAGE_PIN W11 [get_ports XCLK]
. . . .
. . . .
. . . .
. . . .

set_property IOSTANDARD LVCMOS33 [get_ports PCLK]
set_property PACKAGE_PIN W10 [get_ports PCLK]

// only for PCLK - the PMOD pin is not CC (clock capable)
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets PCLK_IBUF]
```

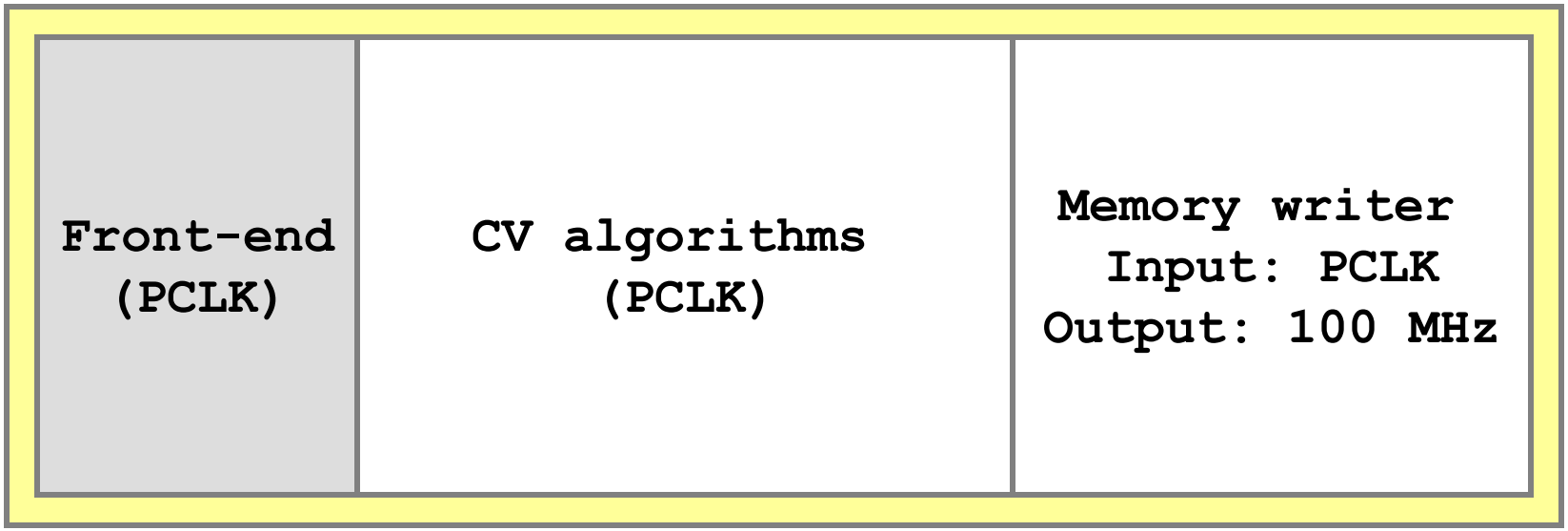

I2C
XCLK
(24 MHz)
D[7..0]
PCLK
VSYNC
HREF
CLK
Frame Index
INT_NEW_FRAME
FRAME[7..0]
FRAME_IN[7..0]
Processing pipeline
(PCLK)
(100 MHz)
Ethernet
I2C
ARM Dual Core Cortex A9
Memory Controller
(25.175 MHz)
(150 MHz)
DDR

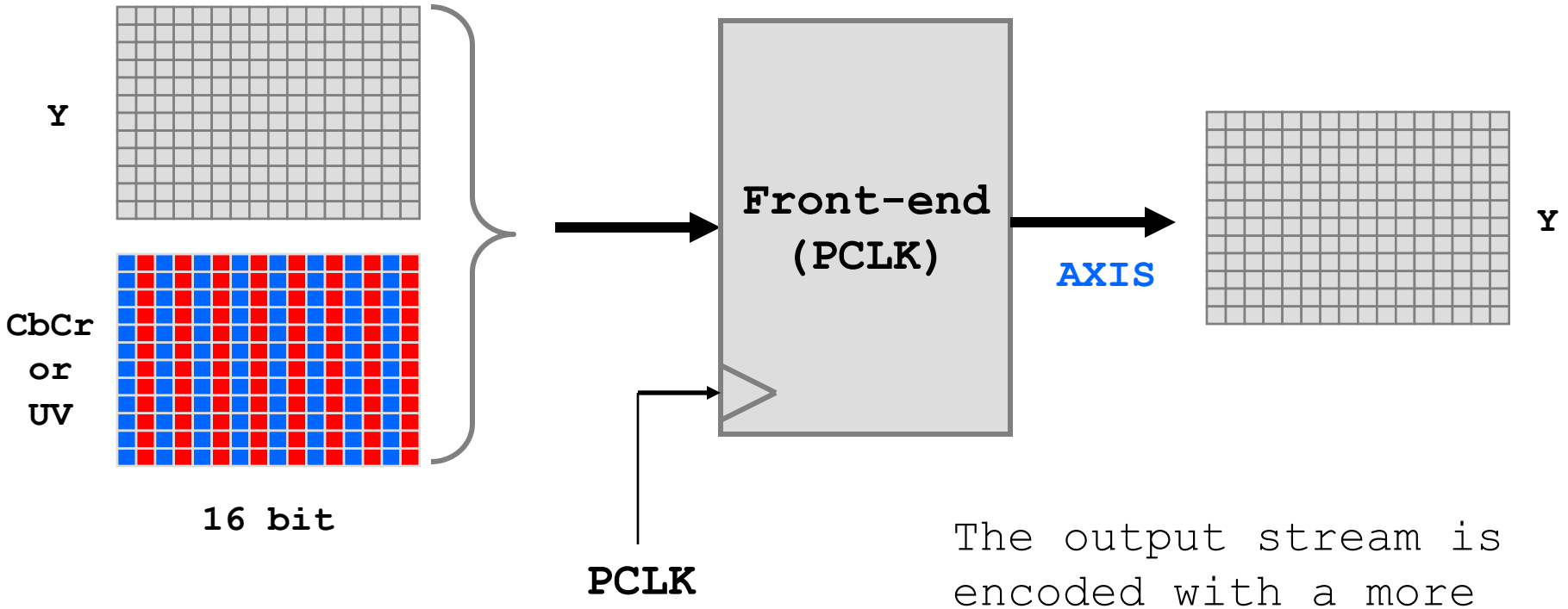
Processing pipeline





Processing pipeline: front-end (OV7670)





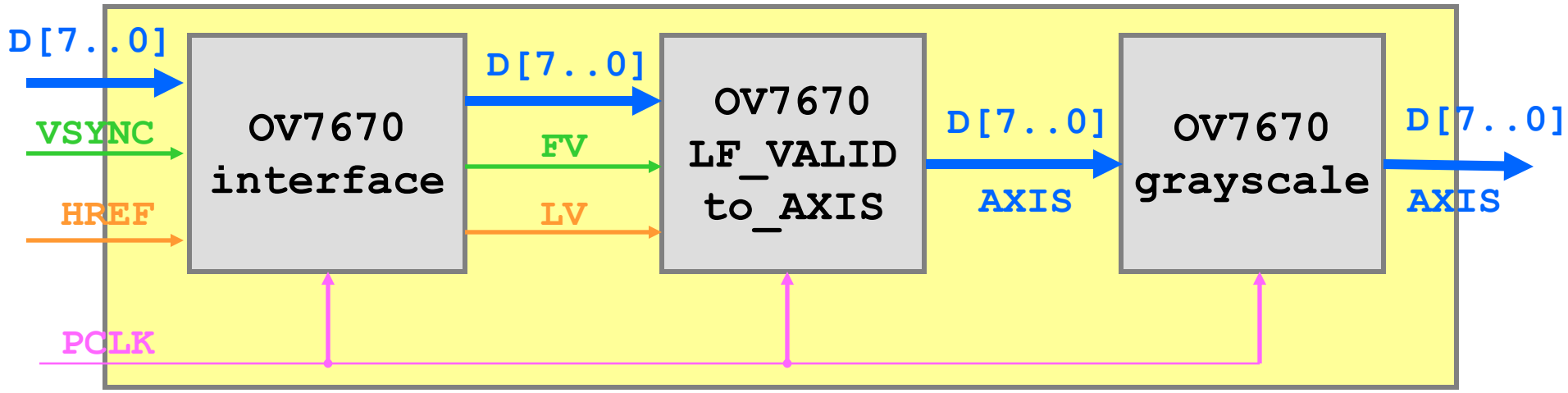
The input stream is encoded with the OV custom/proprietary protocol

The output stream is encoded with a more standard interface (AXIS).
From this point the processing pipeline is agnostic to the specific image sensor

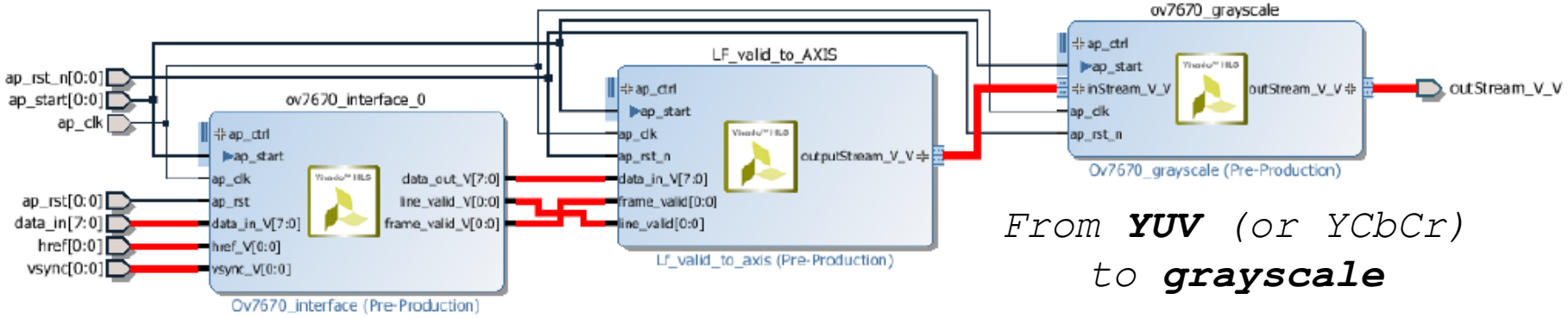
Transcodes from HREF-VREF to LV-FV

Converts the LF-FV format to AXIS

Picks only one out of 2 bytes, the Luma (Y)



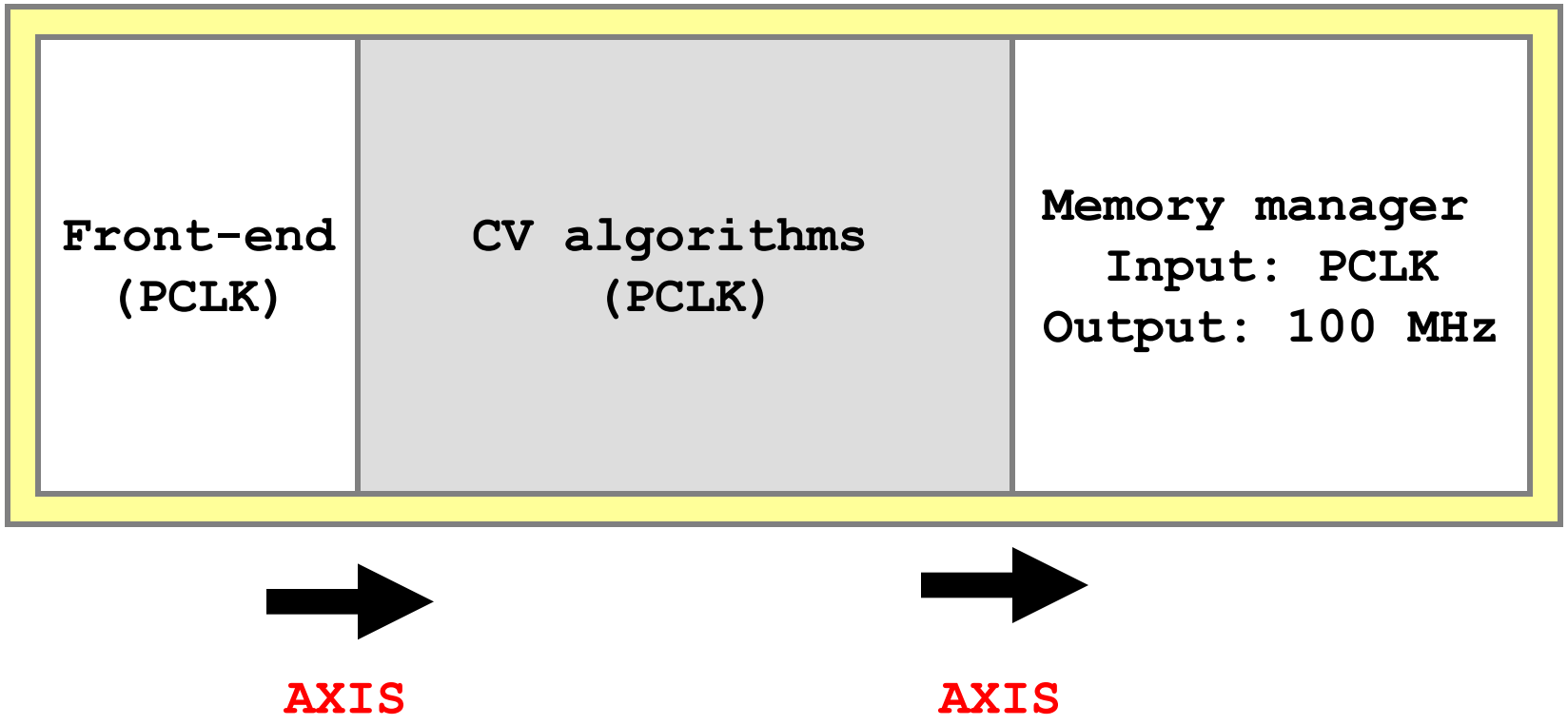
From here, the encoding is hopefully agnostic to the imaging sensor interface



*From **YUV** (or YCbCr) to **grayscale***

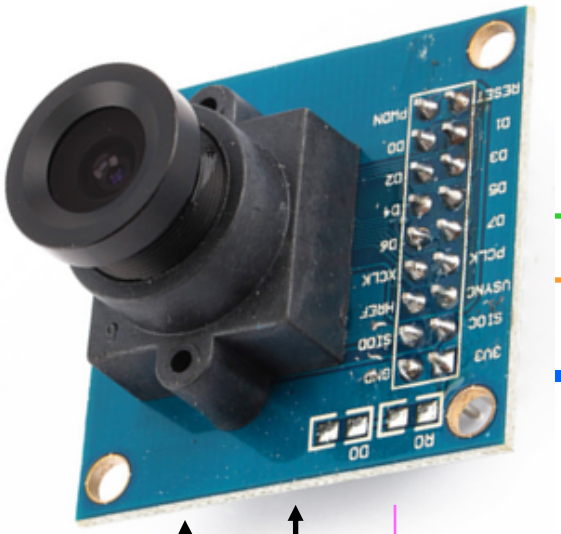
Processing pipeline: CV algorithms

This module is application specific and methodology to implement computer vision algorithms on FPGA will be discussed later.



Exercise

Generate **FRAME_VALID**, **LINE_VALID** and **D[7..0]** signals from an OV7670 video stream. Moreover, **LED_BLINK** inverts its value every 30 images



VSYNC

HREF

D[7..0]

VSYNC

HREF

D[7..0]

FRAME_VALID

LINE_VALID

D_OUT[7..0]

LED_BLINK

OV_7670
INTERFACE

XCLK
(24 MHz)

PCLK

PWDN
(1)

RESET*
(0)

// only for PCLK - the PMOD pin is not CC (clock capable)
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets PCLK_IBUF]

I2C and OV7670 (ARM side) 1/2

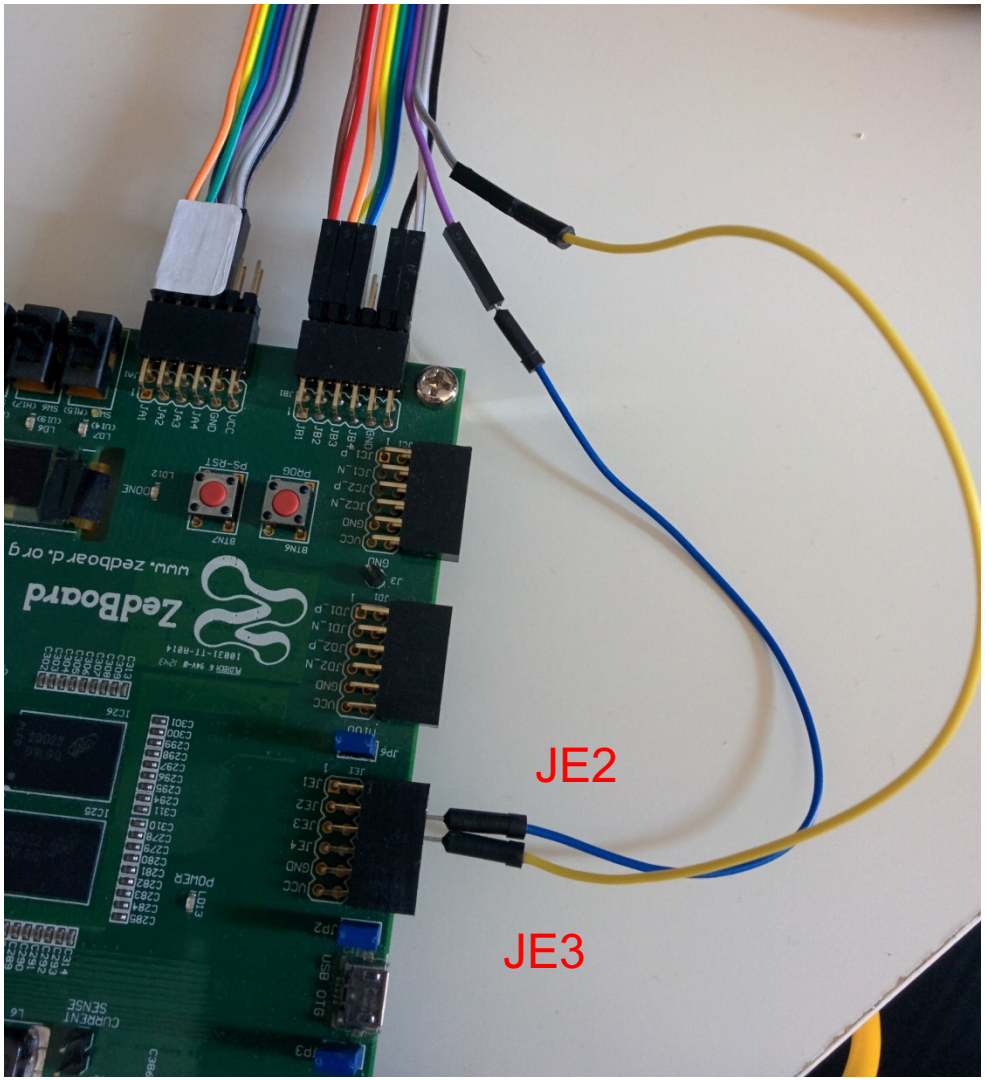
Connections between the PMOD connector JE and the OV7670 camera using the flat cable/wires:

SIOC (clock), purple, connected to **SCL** pin(**JE2**)

SIOD (data), gray, connected to **SDIOD** pin (**JE3**)

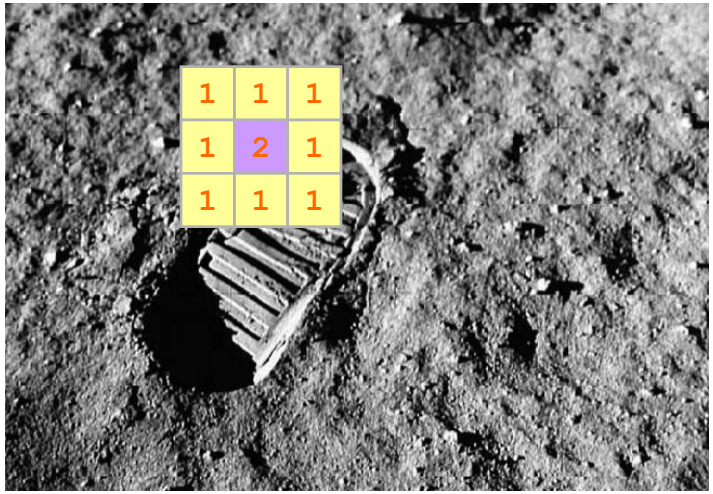
I2C and OV7670 (ARM side) 2/2

OV 7670
SIOD/SDA
SIOC/SCL



PMOD E PIN
JE3
JE2

Example: convolution filter



200	100	100
100	100	200
100	200	100

*

1	1	1
1	2	1
1	1	1

=

130

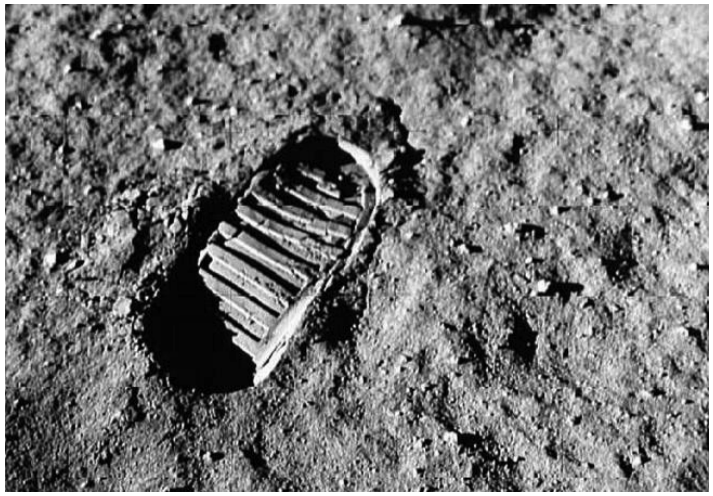
$$(200 \cdot 1 + 100 \cdot 1 + 100 \cdot 1 + 100 \cdot 1 + 100 \cdot 2 + 200 \cdot 1 + 100 \cdot 1 + 200 \cdot 1 + 100 \cdot 1)/10 = 130$$

What is a convolution kernel

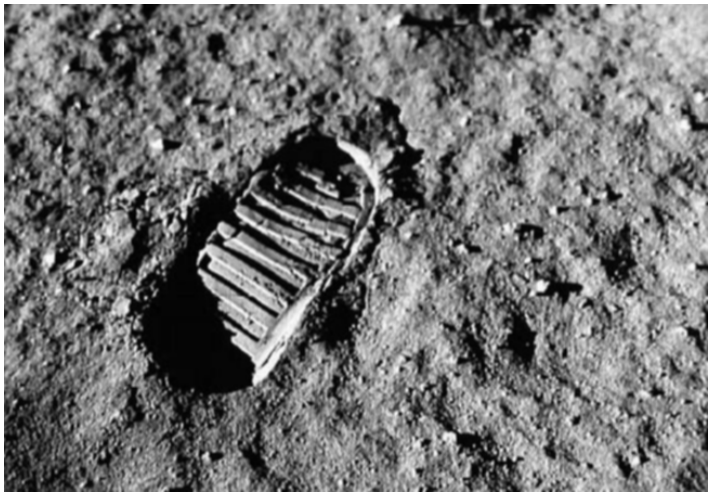
A convolution kernel is a set of weights that allows obtaining, through the weighted sum of neighboring pixels, the value of the central output one

Precisely as in the previous examples, whose effect on the input image on the left is reported on the right image

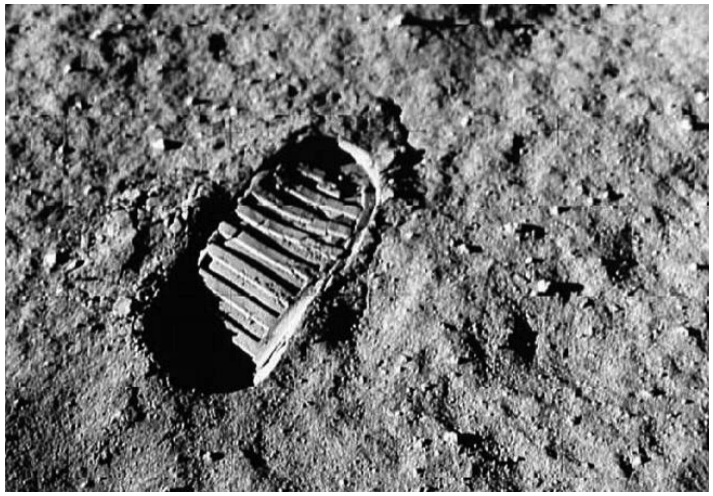
Kernel (weighted average)



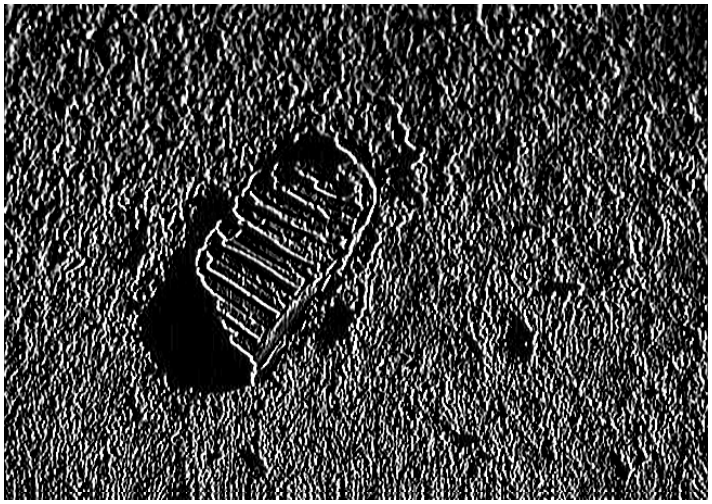
1	1	1
1	2	1
1	1	1



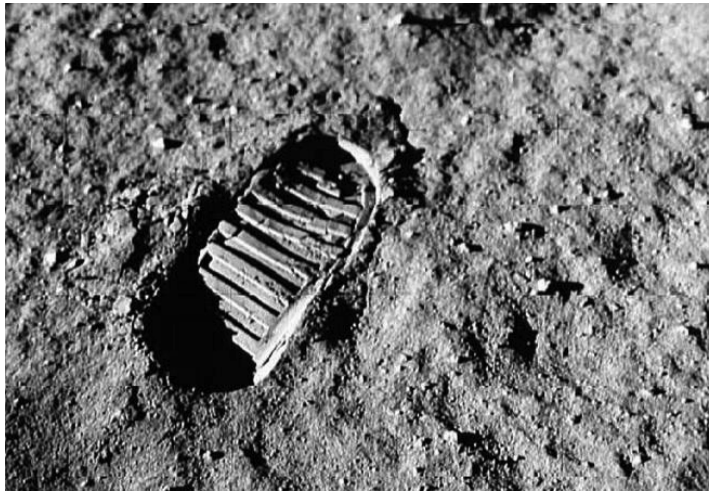
Kernel (Gradient x): |Gx|



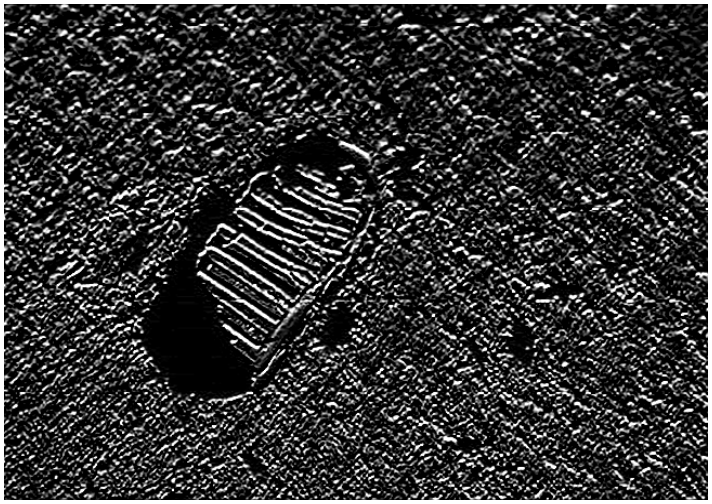
-1	0	1
-2	0	2
-1	0	1



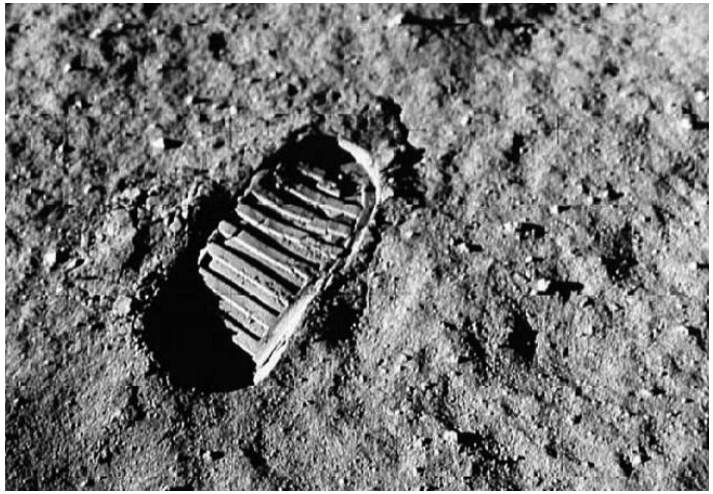
Kernel (Gradient y): |Gy|



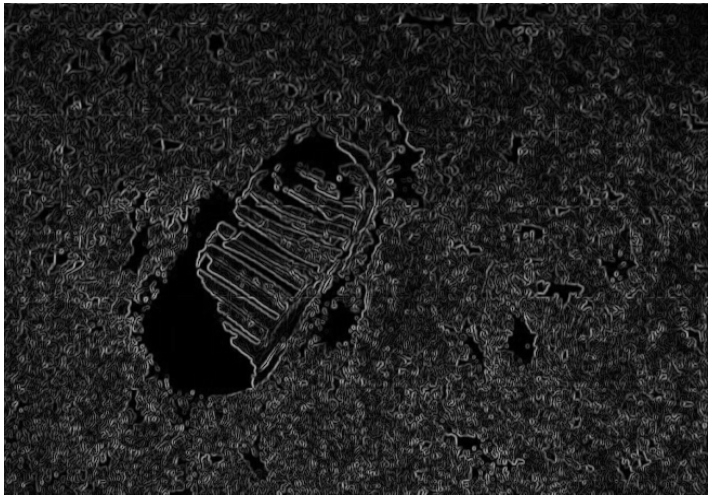
-1	-2	-1
0	0	0
1	2	1



Sobel (Magnitude)



$$G = \sqrt{(G_X)^2 + (G_y)^2}$$



- **Changing the weights, we obtain different results**
- **In contrast to convolution, we can also perform on the patches non linear operations such as *bilateral filtering*, *mediant filtering* and so on.**

These operations, especially linear ones, are at the core of Convolutional Neural Networks (CNNs). In this case, the weights are not set beforehand but adjusted during the training of the networks.

Example: weighted average 1/10

- Each output pixel is the weighted sum of its neearby points within the patch (3x3 in this example)

Intorno di Pin0

Pin1	Pin2	Pin3
Pin4	Pin0	Pin5
Pin6	Pin7	Pin8

Kernel dei pesi

1	1	1
1	2	1
1	1	1

Concolution kernel (weighted average)

DIM_KERNEL(K_H,K_W)

Pout0 = (Pin1 + Pin2 + Pin3 + Pin4 + Pin5 + Pin6 + Pin5 + Pin8 + (2 * Pin0)) / 10

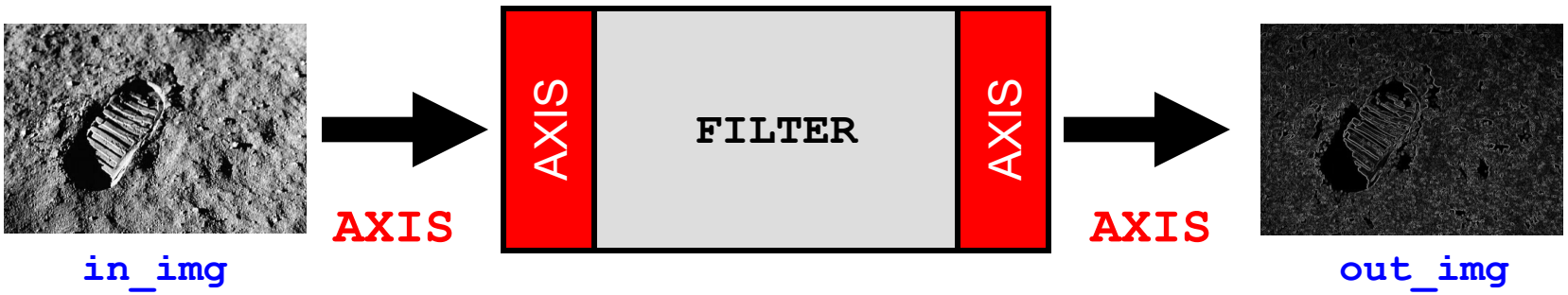
- The HLS design is implemented in a PIPELINE fashion, at each clock the modulo reads an input pixel and compute the output (ie, *stream* processing)

Example: weighted average 2/10

- We need to specify which is the interface to handle input and output values (ie, pixels)
- For instance, we can adopt the AXI stream protocol, deploying the **AXIS** directive in Vivado HLS

- It is an interface enabling sequential communication, and it is not *memory mapped (ie, no address specified)*
- At the lowest level, data is buffered and transmitted using the *handshake* protocol

```
#pragma HLS INTERFACE axis port=in_img
#pragma HLS INTERFACE axis port=out_img
```



Example: weighted average 3/10

- The "top function" processes the input image and produces the filtered image according to the kernel weights
- Images can be modeled as 2D arrays

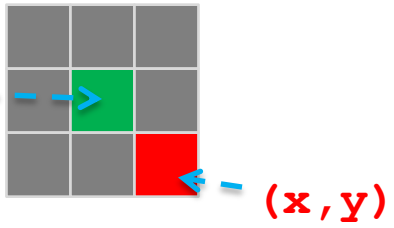
```
void filtroMedia (ptype   in_img[HEIGHT_IMG][WIDTH_IMG],
                  ptype   out_img[HEIGHT_IMG][WIDTH_IMG]
                 )
```

- The C code structure consists of two for cycles: one (outer) for rows and one (inner) for columns

```
34 Loop_row: for(int row = 0; row < HEIGHT_IMG + Km_H; row++){
35     Loop_col: for(int col = 0; col < WIDTH_IMG + Km_W; col++)
```

- At each iteration of the inner loop: the module receives the pixel at coordinates (x,y) and computes the output pixel at coordinate:

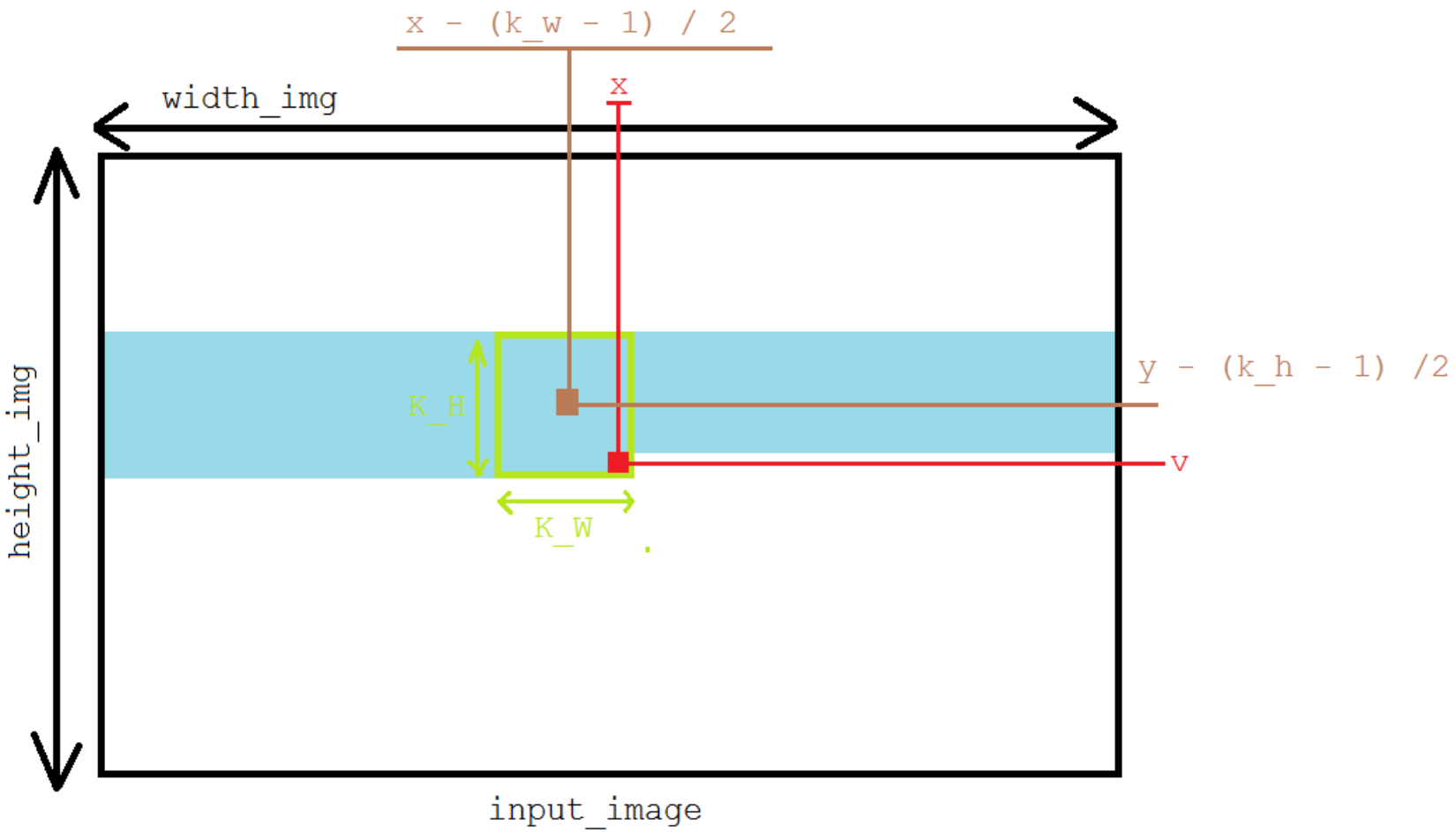
(x - (k_w - 1)/2, y -(k_h - 1) / 2)



Example: weighted average 4/10

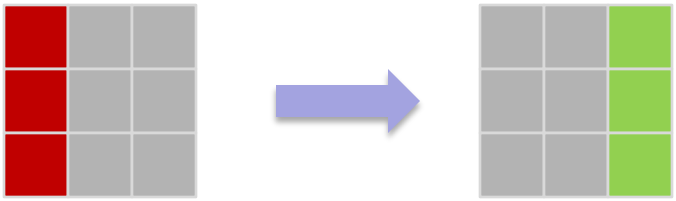
If a subset of previous pixels has been buffered, when the pixel (x,y) is fed to the module, we can process the pixel at

(x - (k_w - 1)/2, y -(k_h - 1) / 2)



Example: weighted average 5/10

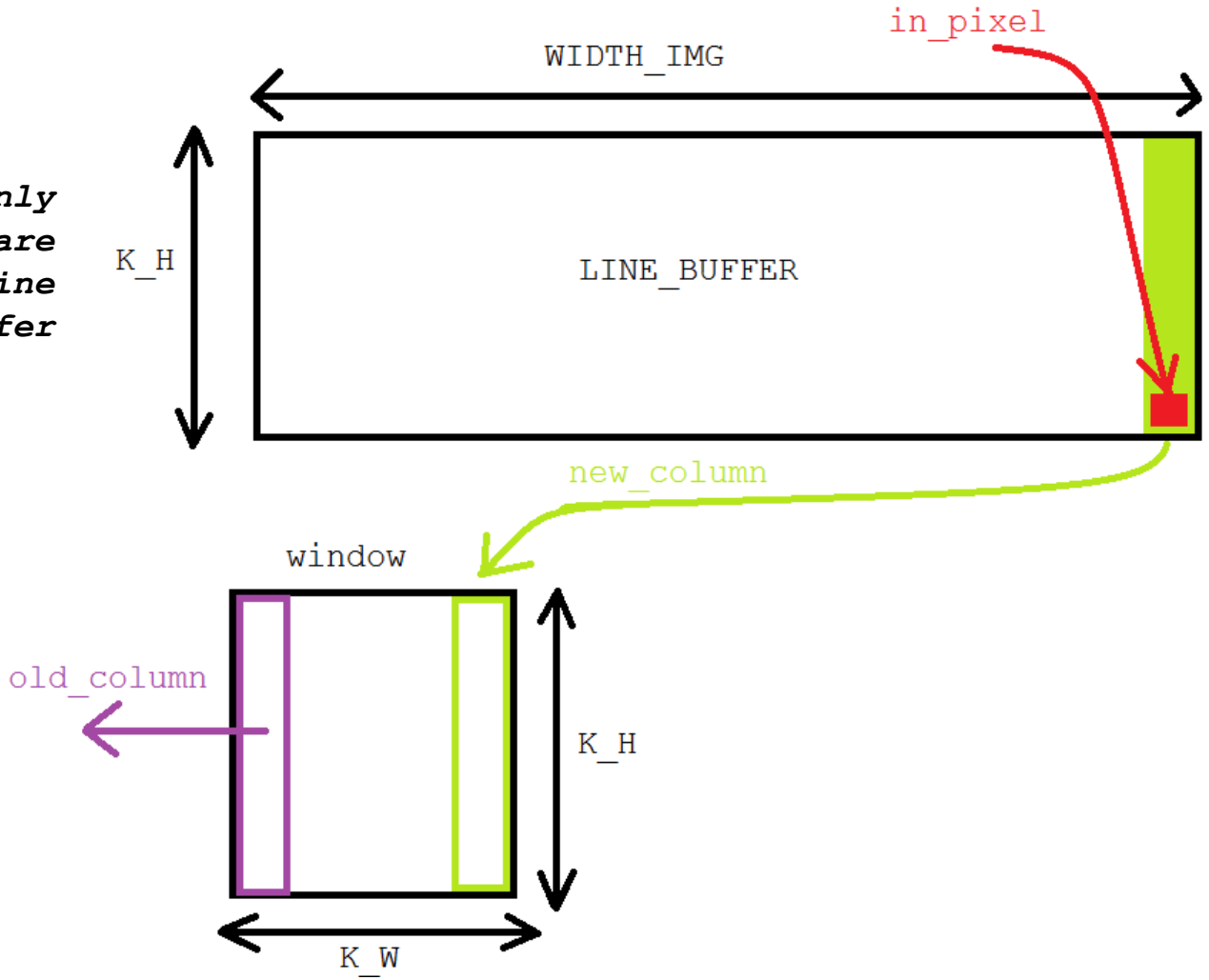
- Each pixel of the input image is received starting from the top left one, line after line
- The lines are buffered in line buffer (in C it is a 2D array). In each entry of the array is buffered a line. The size of the line buffer is: WIDTH_IMG x (K_H - 1)
- In a data strucure of smaller size (window) are stored the pixels needed to compute the weighted average. The size of such a window is: K_H x K_W
- After each iteration, the least recent column is replaced with a one (from line buffer) in the window data structure



Example: weighted average 6/10

Datas structures line buffer and window:

Actually, only K_H - 1 lines are needed in the line buffer



Example: weighted average 7/10

- The window data structure, once initialized, is fed to the media_pixel function aimed at computing the weighted average. Purposely, all elements of the window data structure need to be accessed concurrently.

- The outcome of such a computation (ie, the weighted average) is the output of the *top function*.

```
ap_uint<8> out_temp = 0;
//function invocation
mediaPixel (window, &out_temp);
//write output
out_img[row - Km_H][col - Km_W] = out_temp(7,0);
```

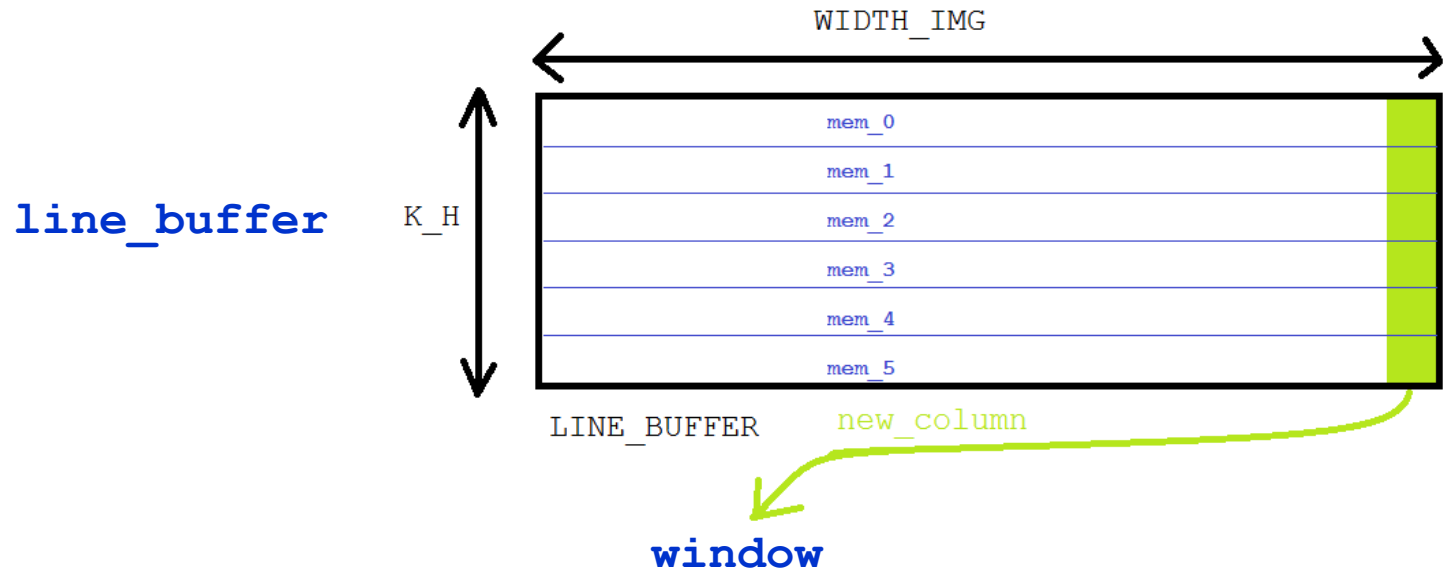
```
//// kernel function
void mediaPixel (ptype y_window[K_H][K_W],ptype *out)
{

    ap_int<20> out_temp = 0;

    Edge_i:  for(int i=0; i < K_H; i++)
        Edge_j: for(int j = 0; j < K_W; j++)
            out_temp = out_temp + y_window[i][j]*M[i][j];


    out_temp = out_temp / SumF;

    *out = out_temp(7,0);

}
```

Example: weighted average 8/10

- For an efficient implementation, at each clock clock all the elements of the line_buffer need to be accessed concurrently. Therefore, the line_buffer, is partitioned in independent rows (ie, each row is mapped on a different memory device).

```
set_directive_array_partition -type complete -dim 1 "filtroMedia" line_buffer_temp
```

- Following this strategy, in a single clock cycle all the elements of the column in green can be read and then copied in the window data structure



Example: weighted average 9/10

- At each clock is computed a weighted average. Therefore, all the elements of the window data structure need to be accessed cconcurrently implying that each element of window is a register
- Directive partition, -dim 0 => each dimension of the array is partitionaed (hence, each element in a register)

```
set_directive_array_partition -type complete -dim 0 "filtroMedia" window
```

Window

FF_0	FF_1	FF_2
FF_3	FF_4	FF_5
FF_6	FF_7	FF_8

Kernel weights

1	1	1
1	2	1
1	1	1

Pout0 = (Pin1 + Pin2 + Pin3 + Pin4 + Pin5 + Pin6 + Pin5 + Pin8 + (2 * Pin0)) / 10

Example: weighted average 10/10

- Outcome of the implementation of the weighted average filter on a Spartan 6 45 FPGA:

Area Estimates

Summary

	BRAM	DSP48A	FF	LUT	SLICE
Component	-	-	-	-	-
Expression	-	2	0	216	-
FIFO	-	-	-	-	-
Memory	2	-	-	-	-
Multiplexer	-	-	-	73	-
Register	-	-	176	-	-
Total	2	2	176	289	0
Available	116	58	54576	27288	6822
Utilization (%)	1	3	~0	1	0

Details

Memory

	Words	Bits	Banks	W*Bits*Banks	BRAM
line_buffer_0_V_U	640	8	1	5120	1
line_buffer_1_V_U	640	8	1	5120	1
Total	1280	16	2	10240	2

Resource Usage

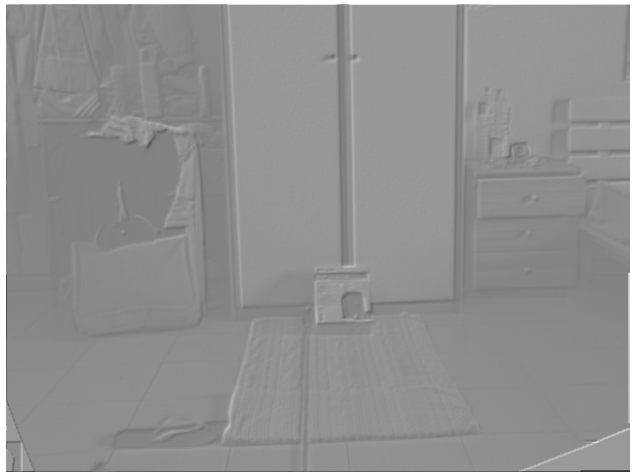
	VHDL	Verilog
SLICE	65	-
LUT	143	-
FF	152	-
DSP	2	-
BRAM	2	-

Only 2 BRAM for buffering the 3 lines

- Further optimization strategies described in the slides «08_Convolution_filters.pdf»

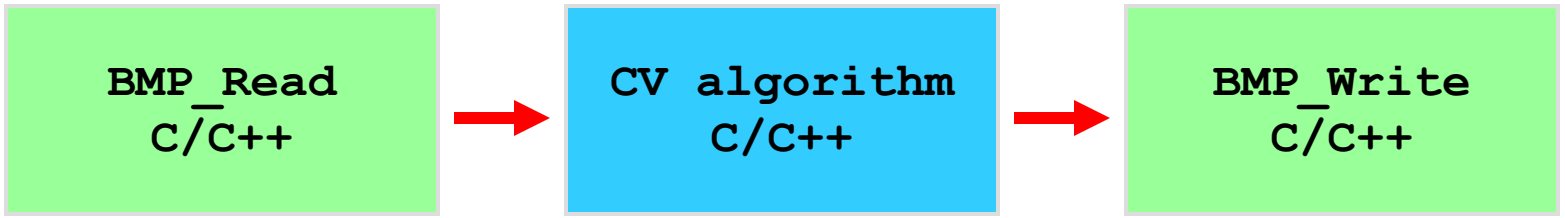
Testbench in C/C++ (HLS)

```
#include "ap_bmp.h"
```



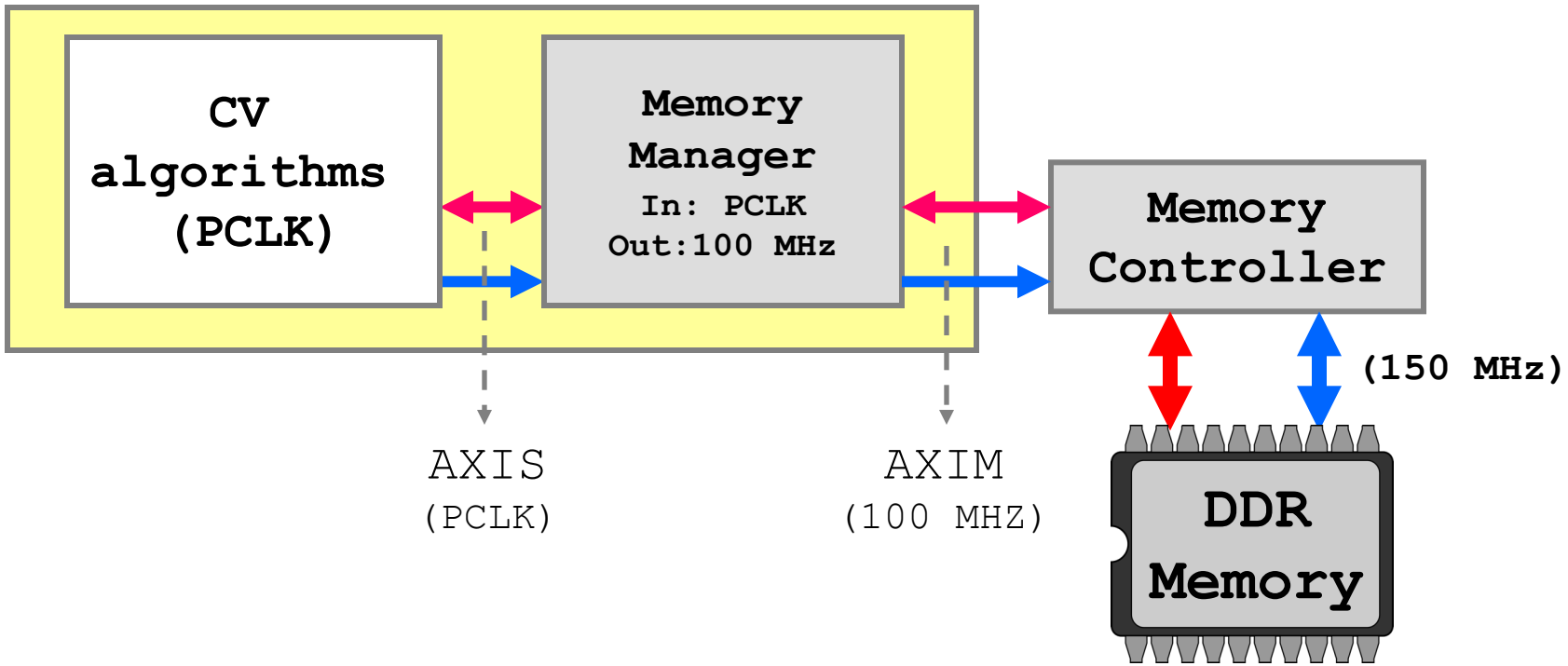
BMP RGB format

BMP RGB format



Memory transfer manager

- It is (upstream) connected (AXIS) to the CV pipeline and (downstream) to the memory controller (HP0) of the Zynq/PS with a 64 bit AXI master (AXIM) interface
- Writes data in the frame-buffer (in DDR memory)
- Memory addresses (frame buffer) can be configured by the ARM processor via AXI_LITE
- A valid address for the frame buffer is 0x10000000

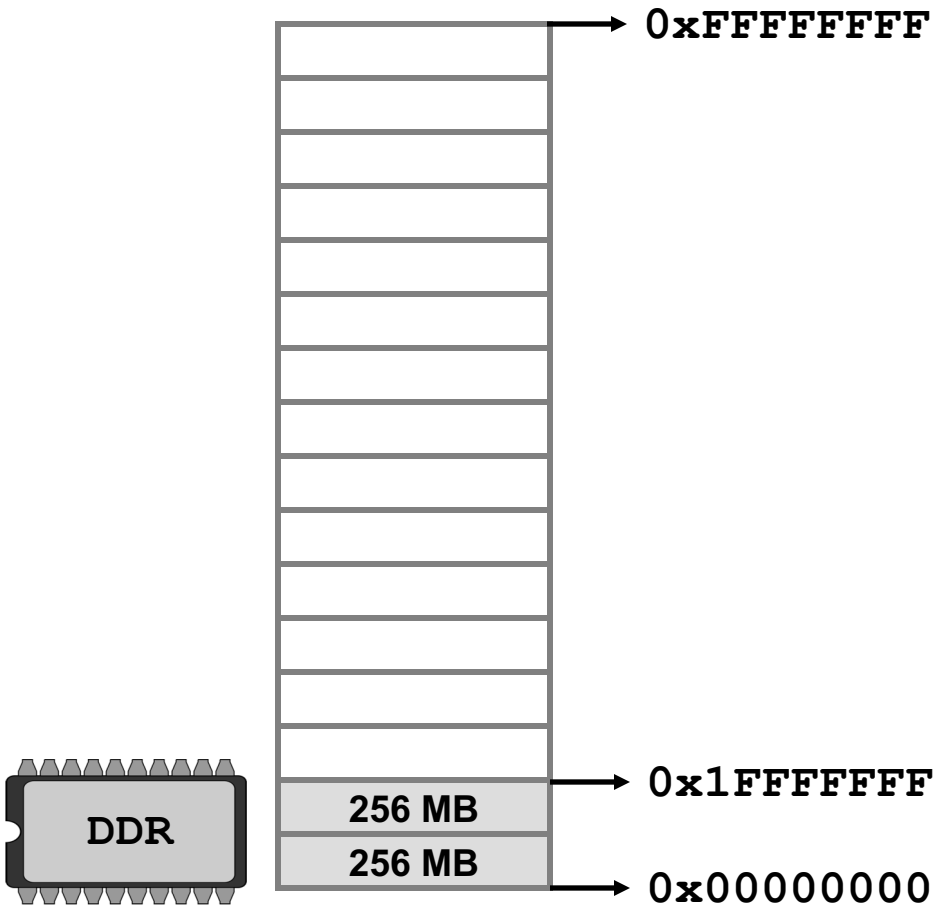


Zedboard: address space and available DDR memory

The Zynq processor has a 4 GB address space

The Zedboard contains 2x256 MB DDR3 memory devices

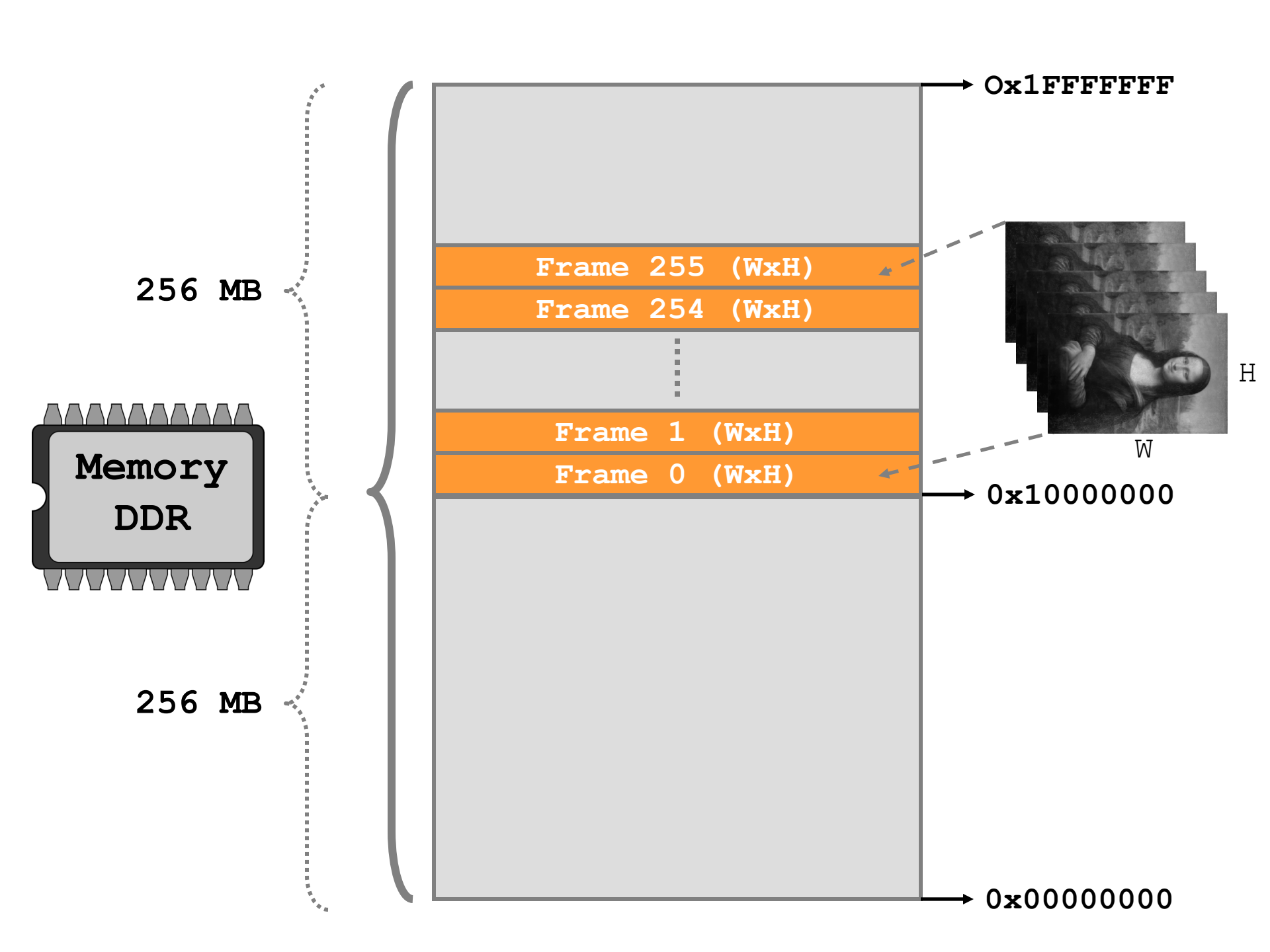
The overall DDR memory available (Zedboard) is 512 MB



The frame-buffer

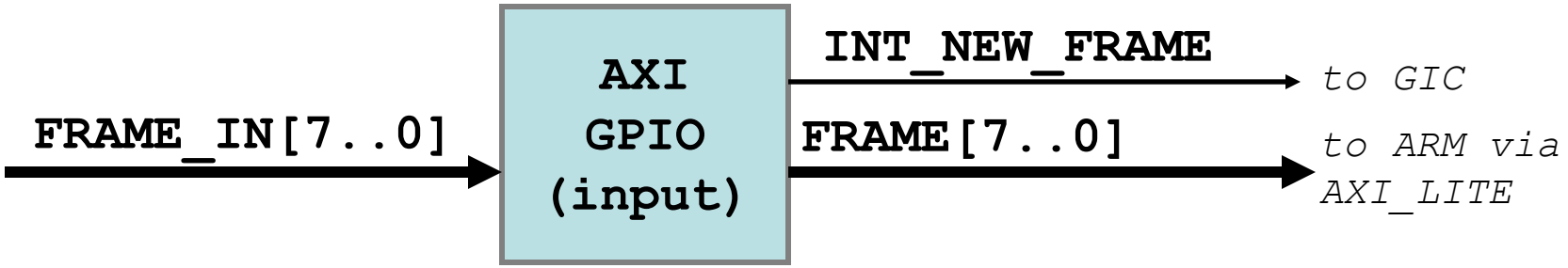
- The design includes a frame-buffer in DDR
- The number of frames is configurable (via AXI-Lite)
- Max 256 frames (typically 8)
- Images are stored in consecutive locations starting from the base of the frame-buffer
- A valid address for the frame buffer is 0x10000000
- Image size is configurable
- The CPU efficiently sends to the host (e.g., via ethernet) the last images stored in the frame buffer by means of an interrupt-driven approach
- The frame-buffer is also a crucial for displaying images on a stadard monitor (e.g., by means of the stadard VGA interface)

The next slide depicts the structure and addresses of the frame-buffer in the DDR memory


0x1FFFFFFF
Frame 255 (WxH)
Frame 254 (WxH)
Frame 1 (WxH)
Frame 0 (WxH)
0x10000000
0x00000000
256 MB
256 MB
Memory
DDR
H
W

The Frame-Index

- This module contains the index of the last frame completely written in memory (i.e., frame-buffer)
- Once an image is completey written in memory, the frame-idexer is filled, by the memory writer, with the index of the frame-buffer element
- This operation triggers an interrupt for the ARM processor
- The interrupt handler reads via AXI-Lite the index and performs appropriate operations accordingly
- The Frame-Index is mapped as peripheral in the address space and based on a standard AXI_GPIO module



Interrupt handling

In the basic embedded CV pipeline outlind the handler manages the pending interrupts according to the strategy configured in the GIC.

In the basic project decribed so far, for each frame the new images is simply transmitted using the ethernet port

It is worth to note that, with I2C, there is at least another interrupt source

The current project is compatible with *Standalone* OS, often referred to as *Bare-metal* OS

However, the same project can be used with Linux OS without any modification to the bitstream

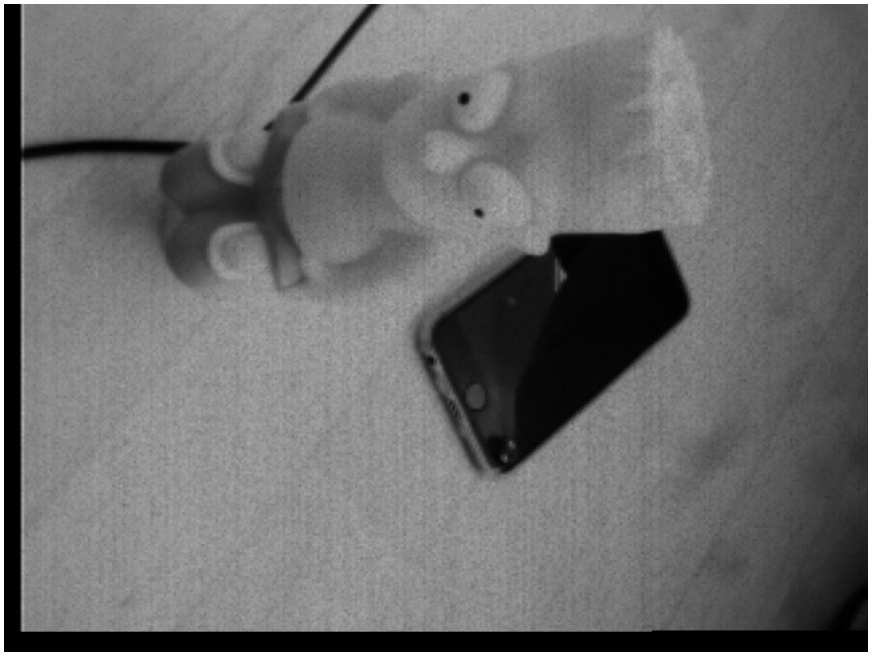
UDP streaming (ARM-Standalone)

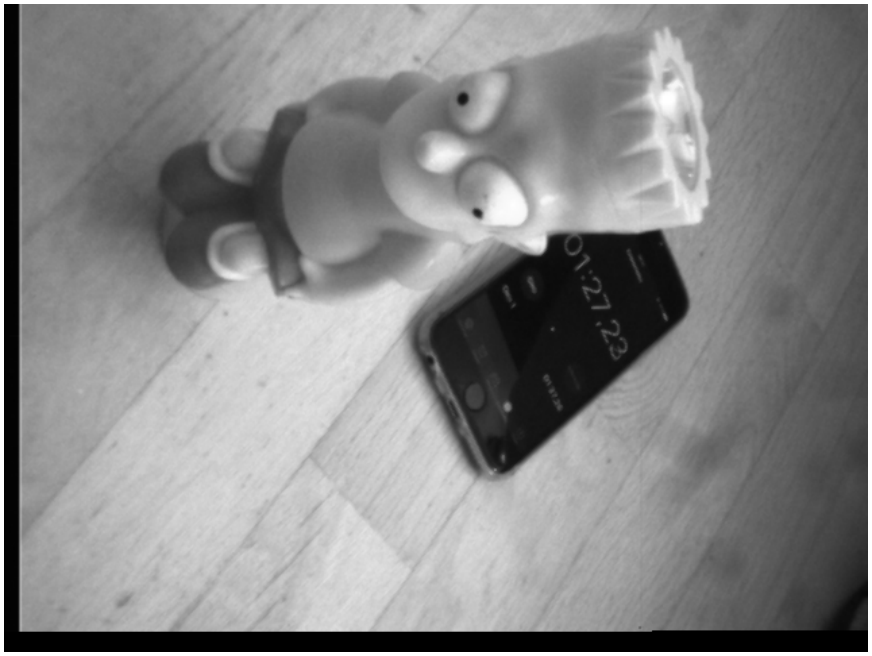
Images, are transmitted as UDP packets using the LwIP - Lightweight IP – framework. For this purpose, it is important to start with the "lwIP Echo Server" application template available in Vivado SDK.

In the next slides some images acquired with a software PC client based on OpenCV connected to the Zynq camera via ethernet

With three images (LUMA, CHROMA and output of the Zynq processing) the bandwidth at VGA resolution is 27 MB/s (640x480x3x30 MB/sec).

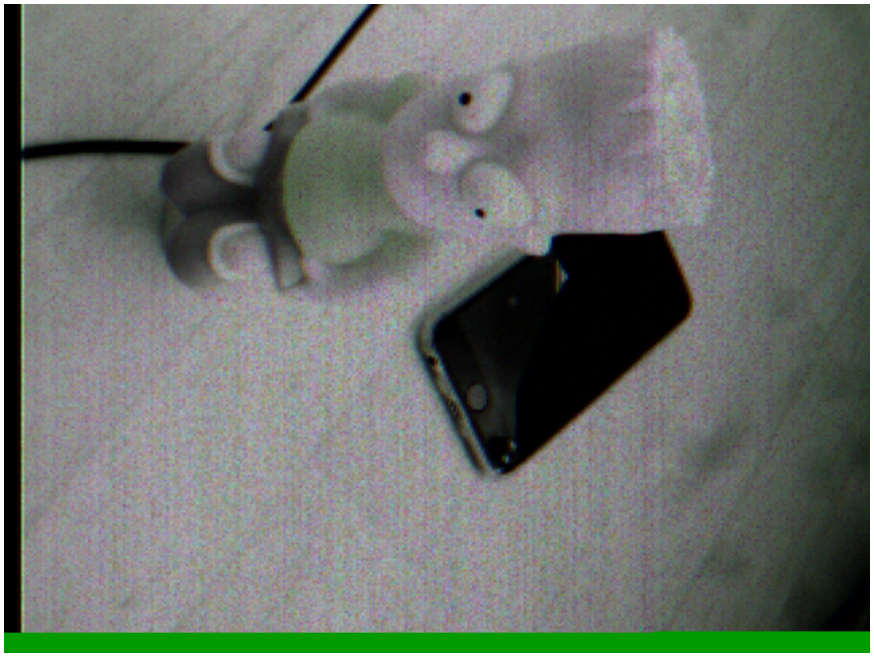
Thus, with three images a Gigabit ethernet adaptter (or a USB 3.0 to Gigabit ethernet adapter connected to a USB 3.0 port) is required. With a single image at 30 fps(about 9 MB/s) a 100 Mbit (or a USB 2.0 to ethernet adapter) has enough bandwidth.





Luma:

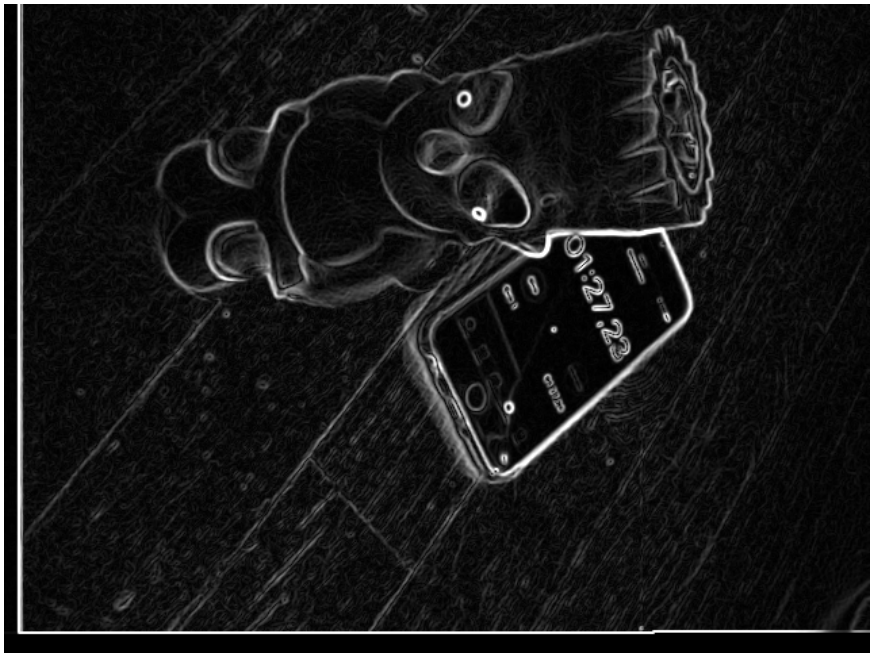
Left, OV7670 default configuration at startup
Right, after I2C configuration



Color:

Left, OV7670 default configuration at startup
Right, after I2C configuration

Luma (Y)



Zynq output

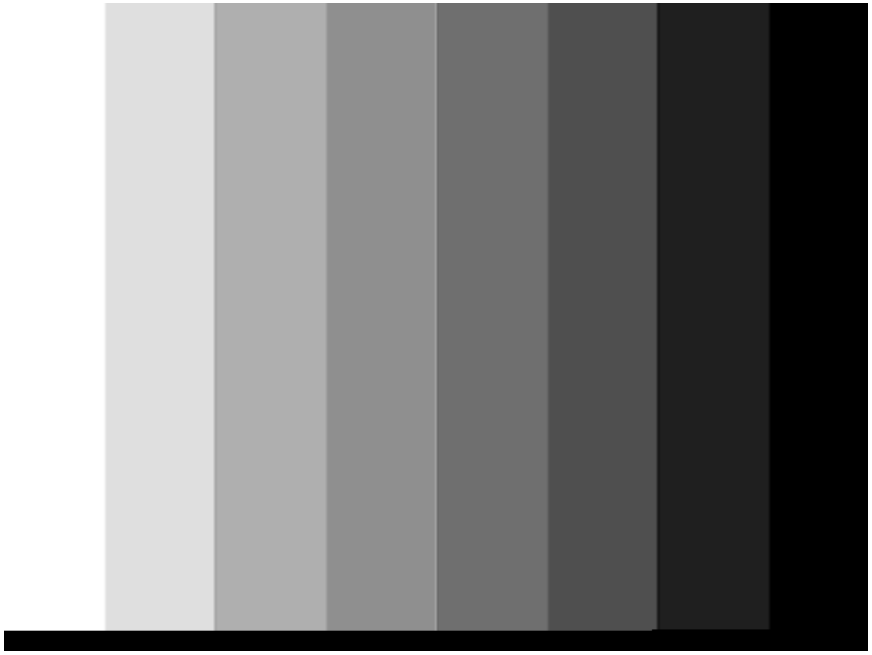
Zynq PL processing:

The OV7670 input stream is processed in real-time by a computer vision pipeline implemented in the PK (i.e., FPGA) with HLS tools (i.e., Vivado HLS)

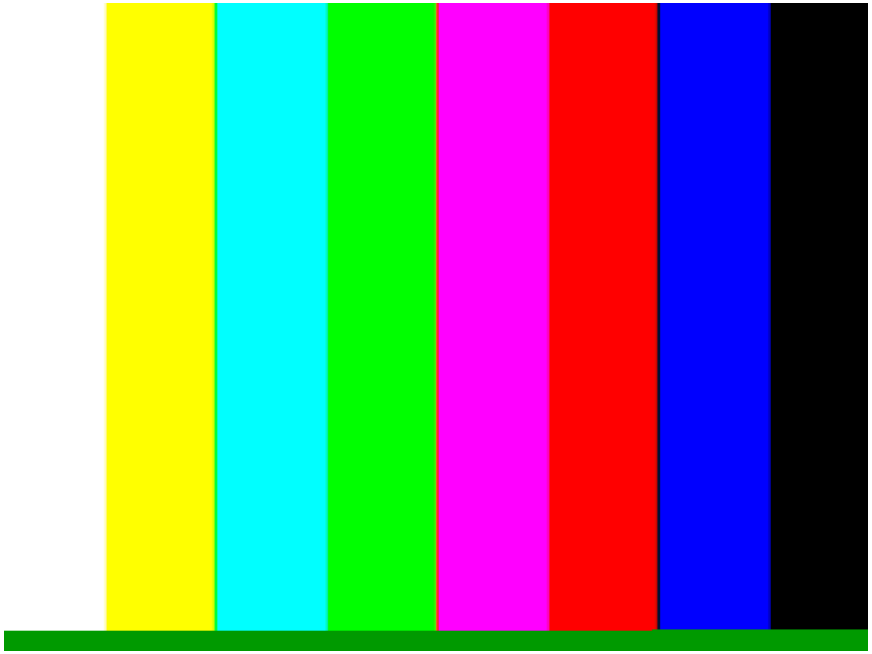
Luma (Y)

Color (RGB)

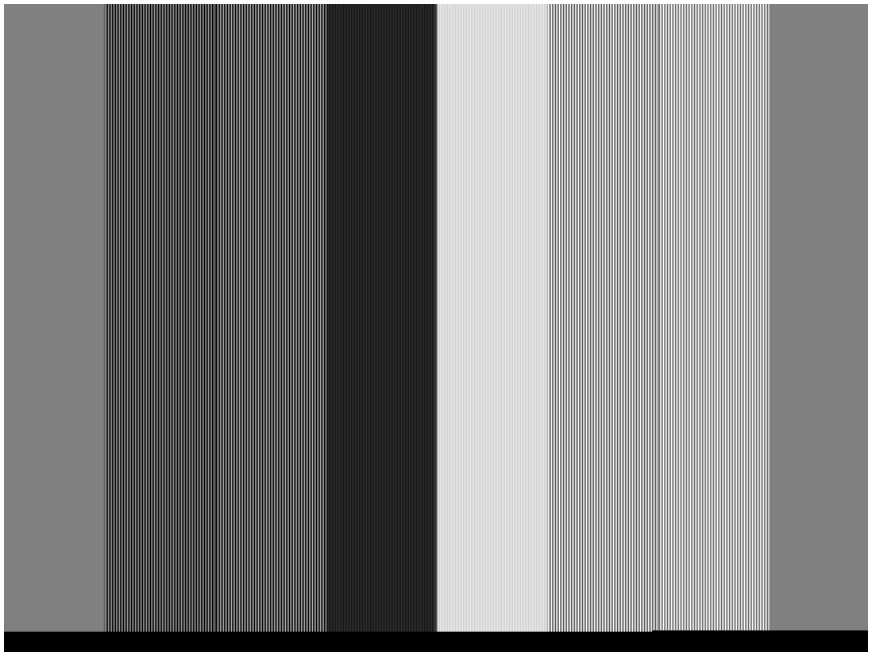
Chroma (UV)



Test pattern Luma (Y)



Test pattern Color (RGB)



Test pattern Chroma (UV)

VGA controller

This (optional) displays on a standard monitor the output video stream of the CV pipeline with (almost) zero latency.

The VGA controller (see the previous project) is fed with data read from the DDR memory using an appropriate READER and the same strategy based on the information provided by the Frame-Index.

In order to account for the critical timing requirements of the VGA video format, clocked at 25.175 MHz at VGA resolution, the READER includes an input buffer (mapped on BRAM).

The VGA module displays color or grayscale (default) images using the VGA output interfce available in the Zedboard. Unfortunately this interface has only 4 bits per channel

Nevertheless, external VGA controllers could be easily plugged in the PMOD connectors

Receiver application (OpenCV)

The image stream is sent to a device (typically a PC) connected (point to point) to the ethernet port of the Zedboard.

The image stream is decoded and displayed using a simple OpenCV application running on the receiver device with minimal overhead exploiting threads.

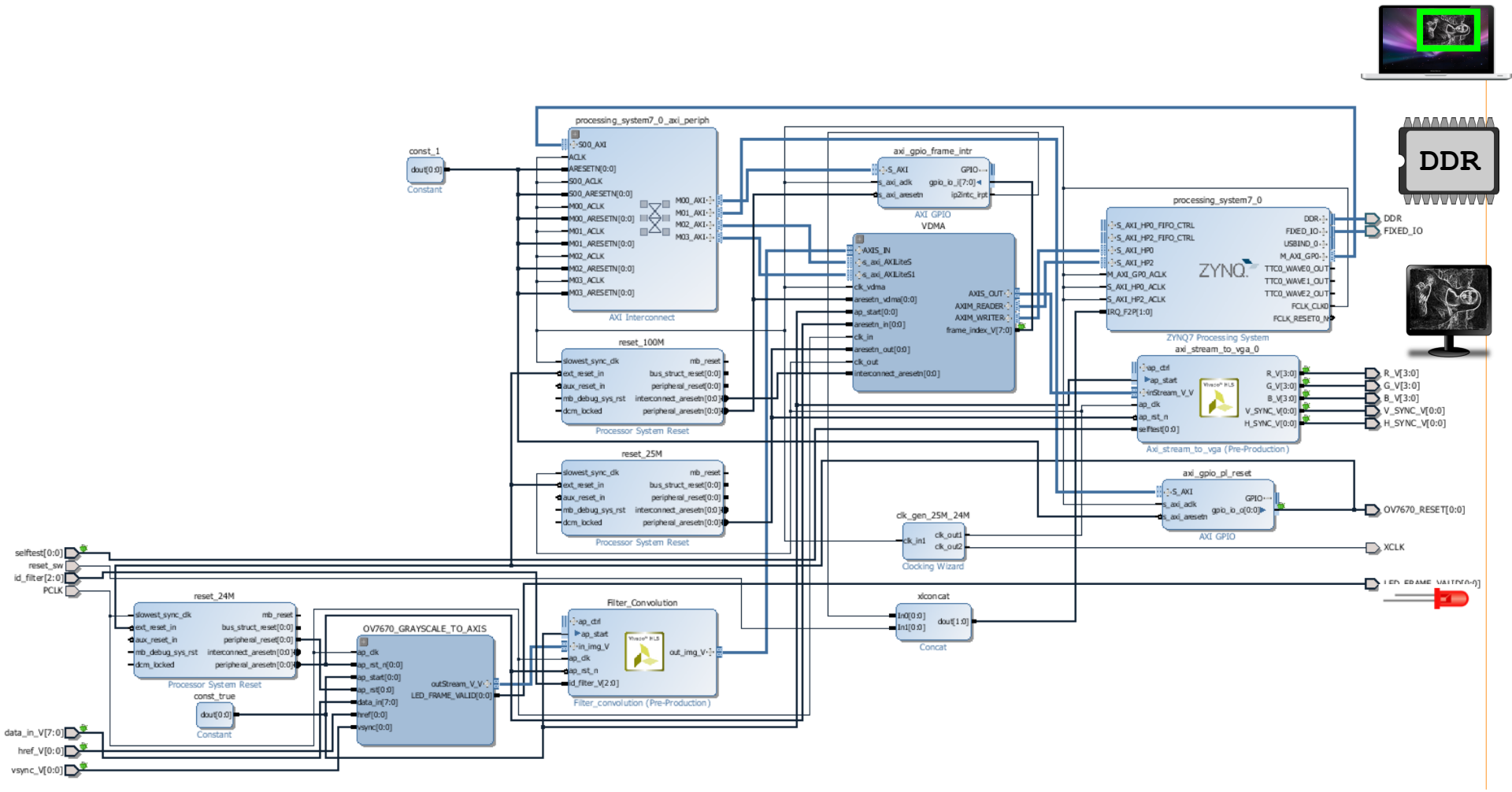
Camera settings:

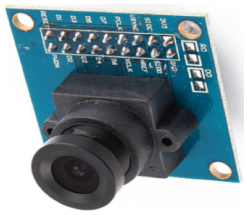
IP	192.168.1.50
Netmask	255.255.255.0
Gateway	192.168.1.1

Host/PC settings:

IP	192.168.1.100

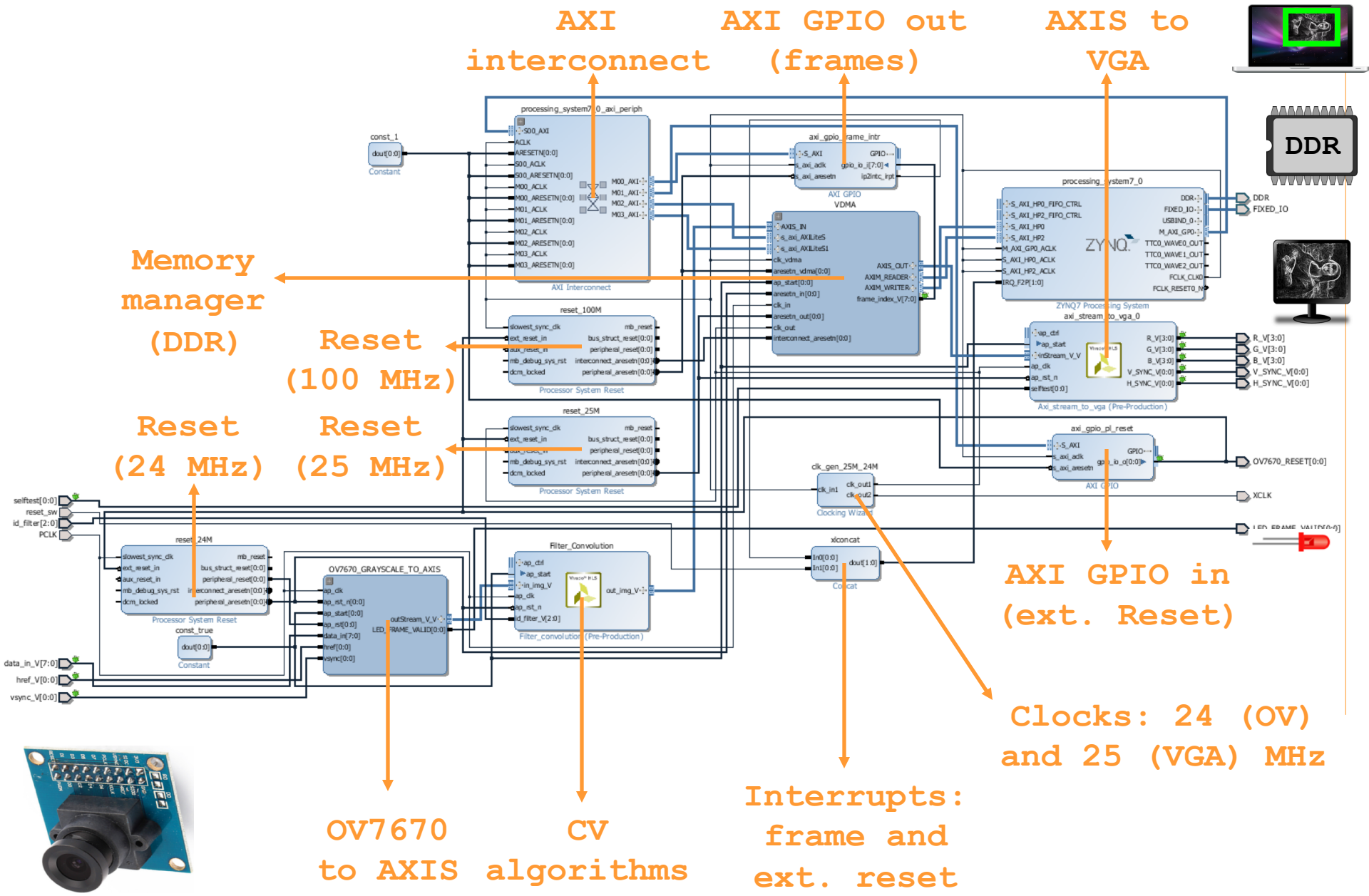
Overall design (Zynq PL) 1/2





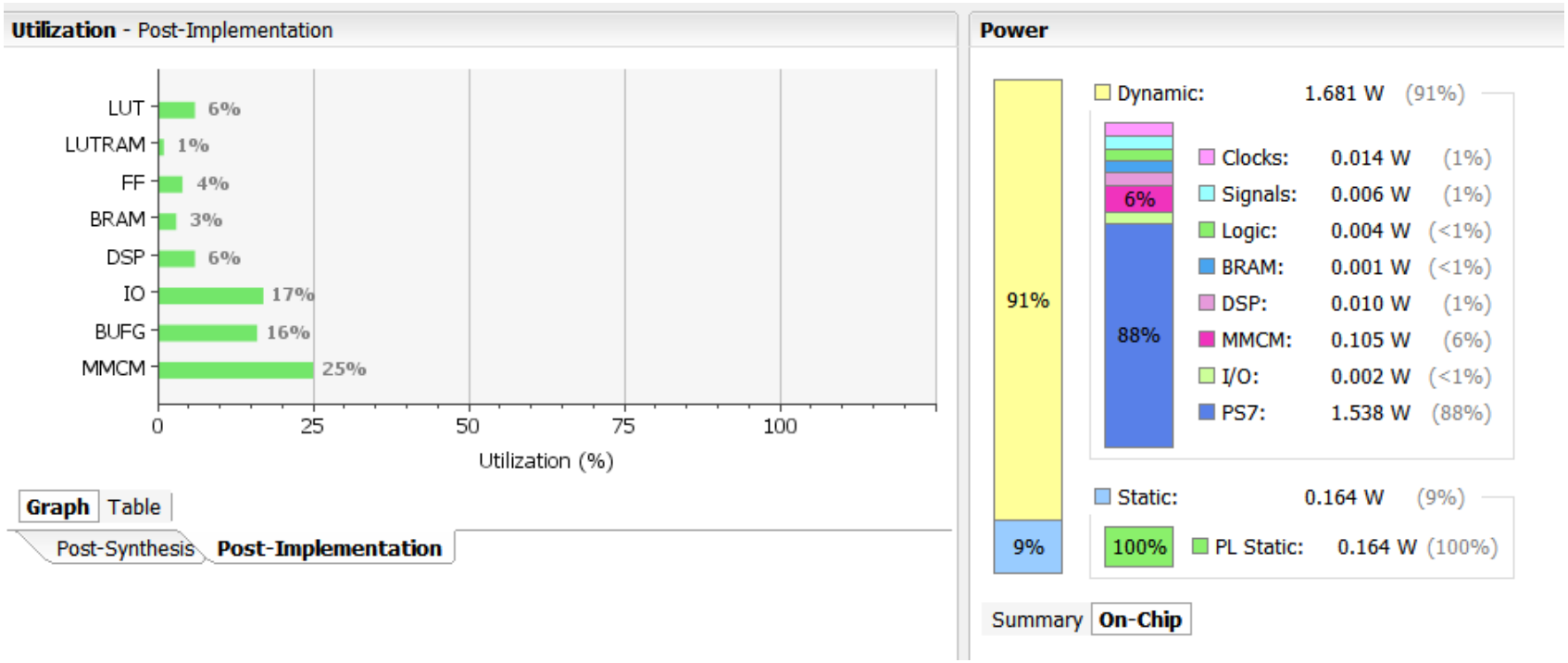
I2C is handled by Zynq (MIO)

Overall design (Zynq PL) 2/2



The overall design utilizes a nehligible portion of the Zynq 7020 device (6% LUT, 1% LUTRAM, 4% FF amd 3% BRAM).

Power consumption < 2 W including ARM core (88%)



ARM: memory and devices 1/2

design_1_wrapper_hw_platform_1 Hardware Platform Specification

Design Information

Target FPGA Device: 7z020
Created With: Vivado 2016.2
Created On: Tue Nov 29 22:25:41 2016

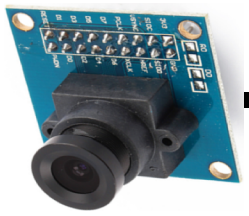
Address Map for processor ps7_cortexa9_0

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
VDMA_axis_to_ddr_writer_0	0x43c00000	0x43c0ffff	s_axi_AXILiteS	REGISTER
VDMA_ddr_to_axis_reader_0	0x43c10000	0x43c1ffff	s_axi_AXILiteS	REGISTER
axi_gpio_frame_intr	0x41200000	0x4120ffff	S_AXI	REGISTER
axi_gpio_pl_reset	0x41210000	0x4121ffff	S_AXI	REGISTER
ps7_afi_0	0xf8008000	0xf8008fff		REGISTER
ps7_afi_1	0xf8009000	0xf8009fff		REGISTER
ps7_afi_2	0xf800a000	0xf800afff		REGISTER
ps7_afi_3	0xf800b000	0xf800bfff		REGISTER
ps7_coresight_comp_0	0xf8800000	0xf88fffff		REGISTER
ps7_ddr_0	0x00100000	0x1fffffff		MEMORY
ps7_ddrc_0	0xf8006000	0xf8006fff		REGISTER
ps7_dev_cfg_0	0xf8007000	0xf80070ff		REGISTER
ps7_dma_ns	0xf8004000	0xf8004fff		REGISTER
ps7_dma_s	0xf8003000	0xf8003fff		REGISTER
ps7_ethernet_0	0xe000b000	0xe000bfff		REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff		REGISTER
ps7_gpio_0	0xe000a000	0xe000afff		REGISTER
ps7_gpv_0	0xf8900000	0xf89fffff		REGISTER

ARM: memory and devices 1/2

ps7_i2c_0	0xe0004000	0xe0004fff	REGISTER
ps7_intc_dist_0	0xf8f01000	0xf8f01fff	REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff	REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff	REGISTER
ps7_ocmc_0	0xf800c000	0xf800cfff	REGISTER
ps7_pl310_0	0xf8f02000	0xf8f02fff	REGISTER
ps7_pmu_0	0xf8893000	0xf8893fff	REGISTER
ps7_qspi_0	0xe000d000	0xe000dfff	REGISTER
ps7_qspi_linear_0	0xfc000000	0xfcffffff	MEMORY
ps7_ram_0	0x00000000	0x0002ffff	MEMORY
ps7_ram_1	0xffff0000	0xffffdfff	MEMORY
ps7_scuc_0	0xf8f00000	0xf8f000fc	REGISTER
ps7_scugic_0	0xf8f00100	0xf8f001ff	REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f0061f	REGISTER
ps7_scuwdt_0	0xf8f00620	0xf8f006ff	REGISTER
ps7_sd_0	0xe0100000	0xe0100fff	REGISTER
ps7_slcr_0	0xf8000000	0xf8000fff	REGISTER
ps7_ttc_0	0xf8001000	0xf8001fff	REGISTER
ps7_uart_1	0xe0001000	0xe0001fff	REGISTER
ps7_usb_0	0xe0002000	0xe0002fff	REGISTER
ps7_xadc_0	0xf8007100	0xf8007120	REGISTER

Acceleration architectures and tools



HLS, OpenCV

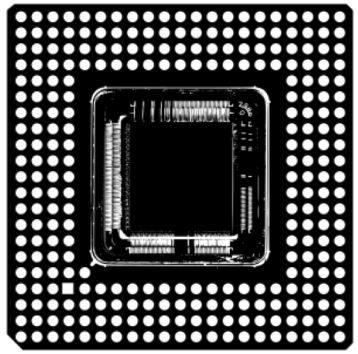
OpenCV, OpenMP
CPU, SIMD

OpenCV, OpenMP
GPU, CPU, SIMD

FPGA

ARM
Cortex A9
800 MHz

ARM
Cortex A9
800 MHz



Zynq, MpSOC or equivalent

Host
(optional)