Avalon Master Interface for the TRDB-D5M Camera
Embedded Systems

Philémon Favrod 218353

Sahand Kashani-Akhavan 201034

November 2014

The goal of this laboratory is to design a programmable interface for the Avalon bus with the ability to acquire an image/video from a Terasic TRDB-D5M camera module and write the result in memory. Later, this image will be displayed on a Terasic LT24 LCD screen, but this report will only be focused on the camera portion of the system. Nevertheless, the end-to-end final product must be kept in mind in order to justify some design choices.

Contents

1 System Overview

A block diagram of the system is depicted in Figure 1. It consists of three main blocks:

- A unit responsible for interfacing with the Avalon bus. It has signals for an Avalon Slave interface and an Avalon Master interface. The Slave signals are used to let the FPGA soft-core processor (or another Avalon Master interface) control/configure the programmable interface and to let the latter interrupt it. The Avalon Master interface is used for direct memory access whenever data from the camera module needs to be stored. The interfaces will be discussed in greater details in sections 2 and 3.

  Note: To distinguish between these two interfaces, the Avalon signals for Slave transfers are prefixed by "`AS_`" while the Avalon signals for Master transfers are prefixed by "`AM_`".

- A *FIFO* queue to synchronize two clock frequency domains, namely the one of the CPU and the one of the camera. It is also used to diminish the data loss probability in case the Avalon bus is not available as soon as data is produced.

- A *camera controller* which handles the acquisition of the raw data from the camera by translating the camera output scheme into the FIFO input scheme. This is done to have a more suitable pixel format than the one directly provided by the camera.

On the right of Figure 1, you can see the `TRIGGER` input pin connected to a button. This button will be used if we want to use the camera to take snapshots instead of continuous video.

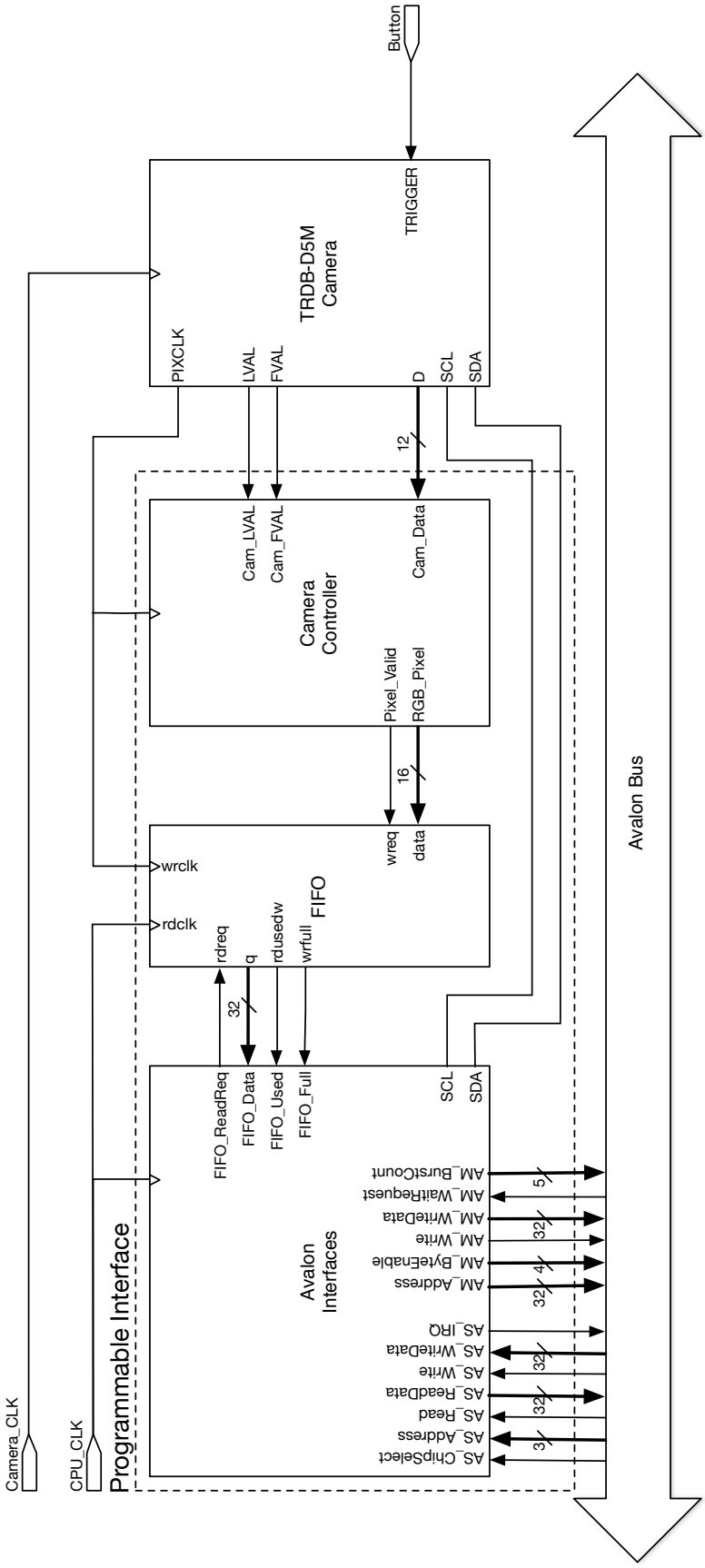

Figure 1: Block diagram of the camera interface

2 The Avalon Slave Interface

To configure the programmable interface, one can access numerous configuration registers through an Avalon Slave Interface. The latter allows the user to control the DMA workflow, and to directly write the internal registers of the TRDB-D5M camera module through I^2C. Its register map is shown in Table 1.

Offset	Name	Access	Description
0	`REG_STATUS`	RO	Status register. See section 2.0.1.
1	`REG_CONTROL`	WO	Control register. See section 2.0.2.
2	`REG_DMA_ADDRESS`	R/W	Base destination write address of the DMA. See section 2.0.5.
3	`REG_DMA_LENGTH`	R/W	Length of memory allocated to the DMA. See section 2.0.5
4	`REG_I2C_DR`	R/W	Transmitted/received data from the camera I^2C configuration interface. See section 2.0.3
5	`REG_I2C_CD`	CD	Clock divisor used by the I^2C interface. See section 2.0.4
6	-	-	
7	-	-	

Table 1: Address/Register map of the Avalon Slave Interface

When writing to a register which is smaller than 32 bits, the most significant bits of `AS_WriteData` are ignored. The same applies in case of a read where only the least significant bits are valid while the remaining bits are zeroed out.

2.0.1 Status register

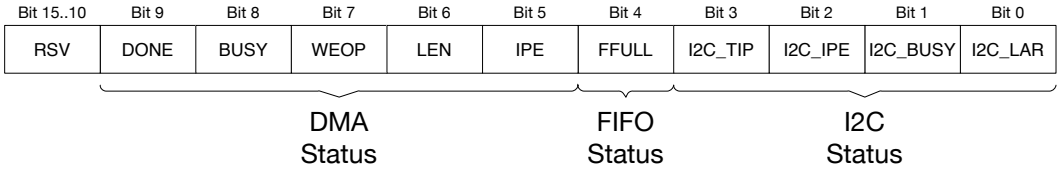

Figure 2: Status register bit map

The status register contains all status flags related to both the DMA unit, and the I^2C unit. The details regarding all its fields can be found in Table 2.

Bit	Description	Generates interrupt
RSV	Reserved/Unused.	N.A.
DMA_DONE	Indicates whether the DMA has finished its current transfer.	On set
DMA_BUSY	Indicates whether the DMA is currently transferring data.	No
DMA_LEN	Indicates that the transfer is DONE and `length` bytes have been transferred.	Indirectly by DONE
DMA_WEOP	Indicates that the transfer is DONE due to the absence of new incoming packets.	Indirectly by DONE
DMA_IPE	DMA Interrupt pending.	On set
DMA_FFULL	Indicates whether the FIFO is full in which case the data written in memory must be considered invalid.	On set
I2C_TIP	Indicates that a I^2C transfer is in progress.	On reset
I2C_IPE	I^2C Interrupt pending.	On set
I2C_BUS	Indicates whether the I^2C channel is busy or available.	No
I2C_LAR	Indicates whether the last acknowledge has been received.	No

Table 2: Detailed status register bit map

2.0.2 Control register

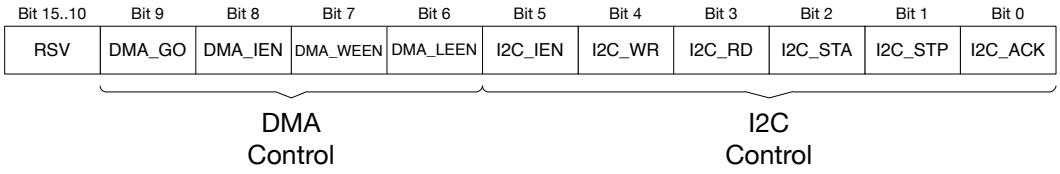

Figure 3: Control register bit map

The details regarding all its fields can be found in Table 3.

Bit	Description
RSV	Reserved/Unused.
DMA_GO	Enable DMA.
DMA_IEN	Enable interrupt.
DMA_WEEN	Enable writing end of packet.
DMA_LEEN	End DMA when REG_DMA_LENGTH register reaches 0.
I2C_IEN	Interrupt Enable.
I2C_WR	Write command bit.
I2C_RD	Read command bit.
I2C_STA	Generate an I^2C Start Sequence.
I2C_STP	Generate an I^2C Stop Sequence.
I2C_ACK	Acknowledge bit for reading.

Table 3: Detailed control register bit map

2.0.3 I^2C Transmit/Receive register

This register contains a byte of data to be written to the camera registers during a write operation, or a byte of data received from the camera after a read to the camera registers. For the purpose of communication with the TRDB-D5M Camera module, the protocol is outlined in Table 4.

Steps	Reading REG	Writing REG
1	Start bit	Start bit
2	Send 0xBA	Send 0xBA
3	Slave ACK	Slave ACK
4	Send ADDR(REG)	Send ADDR(REG)
5	Slave ACK	Slave ACK
6	Start bit	Send new value
7	Send 0xBB	Slave ACK
8	Slave ACK	Stop bit
9	Receive REG per 8-bits block	
10	Stop sending ACK to stop reading	

Table 4: Reading/writing the registers of the TRDB-D5M Camera module

2.0.4 I^2C Clock Divider register

The standard data rate for an I^2C transfer is 100 kbits/s. However, in order to meet the timing constraints of the protocol, the I^2C controller needs to operate 4 times faster.

Therefore, one must set `REG_I2C_CD` to

$$\frac{f_{CPUCLK}}{4 \times 100\ kHz}$$

where f_{CPUCLK} is the frequency of the CPU clock.

2.0.5 DMA Address & Length register

`REG_DMA_ADDRESS` determines the address to which the DMA should write its data. It is incremented by 4 after each memory write. `REG_DMA_LENGTH` holds the number of remaining bytes the CPU/programmer has allocated to the DMA. Similarly, it is decremented by 4 after each memory write. If `DMA_LEEN` is asserted, it stops the DMA when it reaches 0 and generates an interrupt.

2.0.6 Example DMA Configuration Code

```
// Initialze DMA controller to write data from the FIFO to address 0x1000 for a
// length of 16 words.
IOWR_32DIRECT(CONFIGURATION_UNIT_BASE, 2, 0x1000); // DMA dst address
IOWR_32DIRECT(CONFIGURATION_UNIT_BASE, 3, 16);     // DMA length

// Enable DMA interrupts, stop DMA operation when length reaches 0, then start DMA.
IOWR_32DIRECT(CONFIGURATION_UNIT_BASE, 1, DMA_LEEN + DMA_IEN + DMA_GO);
```

Listing 1: Programming the DMA

Note that we do not need to set the source address for the DMA transfer, as the DMA unit has the FIFO hard-wired as its input.

2.0.7 Example I^{2}C Configuration Code

The camera I^{2}C slave interface can be reached at I^{2}C address `0xBA` [1, p.53].

The camera manual has predetermined values for specific target resolutions. The values we chose correspond to [1, p.8, Table 1.7: Standard Resolutions - 640 × 480 with no binning or skipping]. We summarize these values in Figure 8. Note that Figure 8 shows frame rates when `CAMERA_CLK` runs at 96 MHz. In our design, `CAMERA_CLK` runs at 10 MHz. The explanation for this clock frequency can be found in Section 4.

```
#include "system.h"
#include "Interface_i2c.h"

#define TRDBD5M_I2C_ADDR                0xBA

#define TRDB5M_COL_SIZE_DATA            639
#define TRDB5M_ROW_SIZE_DATA            479
#define TRDB5M_SHUTTER_WIDTH_LOWER_DATA (TRDB5M_ROW_SIZE_DATA-1) // Any number lower than
                                                                 // TRDB5M_ROW_SIZE_DATA
#define TRDB5M_ROW_BIN_DATA             0
#define TRDB5M_ROW_SKIP_DATA            0
#define TRDB5MDATA                      0
#define TRDB5M_COL_SKIP_DATA            0

// Initialize clock I2C divider to have 100 kHz
IOWR_I2C_CLKDIV(TRDBD5M_I2C_ADDR, ALT_CPU_FREQ / (4 * 100000));

RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
                             TRDB5M_COL_SIZE_ADDR,
                             TRDB5M_COL_SIZE_DATA);

RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
                             TRDB5M_ROW_SIZE_ADDR,
                             TRDB5M_ROW_SIZE_DATA);
```

```
25
26 RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
27                               TRDB5M_SHUTTER_WIDTH_LOWER_ADDR,
28                               TRDB5M_SHUTTER_WIDTH_LOWER_DATA);
29
30 RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
31                               TRDB5M_ROW_BIN_ADDR,
32                               TRDB5M_ROW_BIN_DATA);
33
34 RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
35                               TRDB5M_ROW_SKIP_ADDR,
36                               TRDB5M_ROW_SKIP_DATA);
37
38 RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
39                               TRDB5M_COL+BIN_ADDR,
40                               TRDB5MDATA);
41
42 RCyc_I2C_WriteDeviceRegister(TRDBD5M_I2C_ADDR,
43                               TRDB5M_COL_SKIP_ADDR,
44                               TRDB5M_COL_SKIP_DATA);
45
46 // Set the camera to snapshot mode so we can use a triggering mechanism.
47 // Need to set bit 8 of register READ_MODE_1.
48 unsigned char read_mode_1;
49 RCyc_I2C_ReadDeviceRegister(TRDBD5M_I2C_ADDR,
50                             TRDB5M_READ_MODE_1_ADDR,
51                             &read_mode_1);
52
53 read_mode_1 |= (1 << 8);
54 RCyc_I2C_ReadDeviceRegister(TRDBD5M_I2C_ADDR,
55                             TRDB5M_READ_MODE_1_ADDR,
56                             read_mode_1);
```

Listing 2: Programming the I^2C interface

3 The Avalon Master Interface

The camera interface contains a DMA unit to write data coming from the FIFO to system memory by going through the Avalon bus. The camera's DMA unit employs a simpler design than traditional units, as it is write-only. Indeed, the DMA unit within the camera interface will never request any read operations on the Avalon bus. The DMA reads 32-bit blocks from the FIFO and write them to memory 16 blocks at a time (burst write). Another peculiarity of our DMA unit is that its "source address" never changes. It always reads data from the FIFO.

3.1 DMA Implementation

The DMA being a Avalon Master Interface, it does not need to wait for the bus to wake up. Instead, once `DMA_GO` is set, the DMA will be triggered by the state of the FIFO. The DMA is designed to perform 16-word burst writes on the bus. That is, it waits for the FIFO to contain 16×32 bits. Then, a clock cycle is spent in order to read the first word from the FIFO. An internal 4-bit counter is reset in order to keep track of the number of writes performed during the burst. The next clock cycles are spent to simultaneously pop data from the FIFO and output them on the bus. On each successful write (when the bus does not make the programmable interface stall), the internal counter is incremented. This goes on until the internal counter overflows in which case 16 words have been written. Figure 4 shows a state diagram depicting this process and figure 5 summarizes its timing.

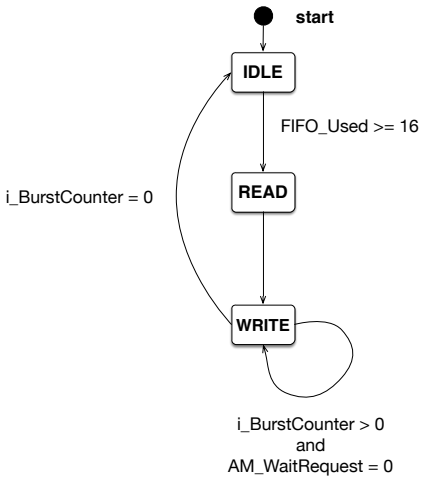

Figure 4: State diagram of the DMA,

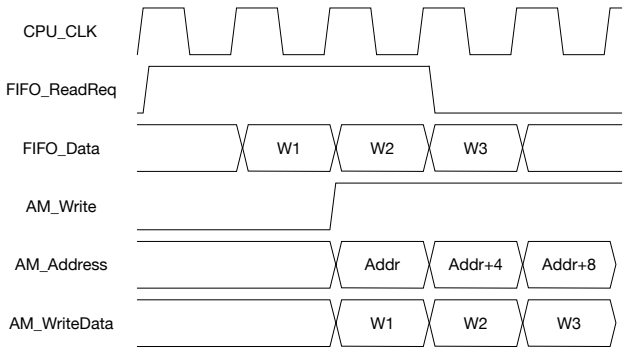

Figure 5: Timing diagram of the DMA.

4 FIFO

The FIFO is the memory element between our camera controller, and our DMA unit. One could ask why this unit is needed. Our camera controller could directly feed the DMA unit with data, which would then be written to memory by going through the Avalon bus.

The reason is actually quite simple. If ever the Avalon bus is busy servicing another units requests, data in the camera controller waiting to be written to memory would be lost. The camera controller doesn't have "extra" pixel space to store anything. It works with the minimum amount of memory elements to be able to reconstruct pixels row by row.

The FIFO introduces a buffering space which can hold a certain amount of data elements before becoming full. However, even with this setup, one can say that the FIFO could overflow if the Avalon bus is too busy. This is true, but the probability of an overflow occurring is greatly decreased due to the FIFO's size.

This probability can be decreased even more by working with two different frequency domain, i.e. clocking the left-hand side of the FIFO (CPU, Avalon Bus/Interfaces) faster than right-hand side of the FIFO (the camera). This justifies why the FIFO in figure 1 is dual-clocked.

The reading clock, called `CPU_CLK` in Figure 1, runs at the same frequency as the CPU and as the SDRAM, namely 50 MHz on the DE0-nano board. This is to have DMA operations run at the same speed as the SDRAM.

The writing clock, called `Camera_CLK` in Figure 1, runs at 25 MHz. The computation of this value was driven by the following analysis: We want to be able to achieve a framerate of 30 FPS on the LCD display. To get such a framerate, the camera needs to be able to output $30 \times 640 \times 480$ pixels per second. The rate at which pixels are sent out by the camera is determined by `PIXCLK`. Therefore, `PIXCLK` needs to have a frequency of $30 \times 640 \times 480 = 9216000$ Hz $= 9.216$ MHz.

In our Qsys design, we didn't want to add another PLL to the system, so we decided to stick with the one that was driving the CPU and the SDRAM. With this single PLL, we were able to create 2 clocks at 50 MHz (for the CPU and SDRAM), but couldn't create a clock at 9.216 MHz. The closest higher frequency we could generate was 10 MHz, so that is what we used as our camera clock.

5 Camera controller

The camera controller is responsible for retrieving information from the camera, transforming the data to a predetermined format, then forwarding the data to the FIFO.

5.1 Interfacing with the camera

The camera module does not immediately output RGB pixels. Instead, it uses a color filter to assign a single color to a pixel on its sensor. To obtain the true color, one must combine the values of multiple neighboring pixels. More precisely, one must convert a Bayer/RGBG pattern into a classical RGB pixel.

Once the data has been captured on the sensor, the camera outputs a padded image starting from the upper left corner down to the lower right one as depicted in figure 6. During such a readout, the rising edge of `PIXCLK` coincides with the output of the next 12-bit camera pixel. As recommended for stability and robustness, the data will be sampled by falling-edge-triggered memory elements [1, p.30]. Figure 7 summarizes this timing.

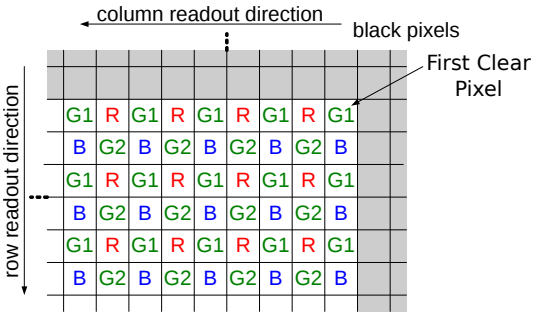

Figure 6: Readout scheme of the camera sensor [1, p.4].

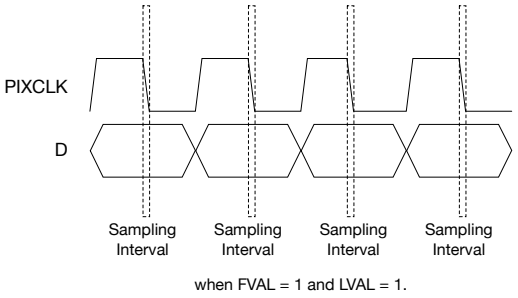

Figure 7: Readout output timing of the camera sensor.

The LCD screen expects to render an image at a resolution of 320×240. To match such a resolution, we have 2 options regarding the camera:

- Capture an image at an equivalently-sized resolution.
- Capture an image at a higher resolution and downscale it to fit on the LCD.

In both cases, the camera is supposed to output a 640×480 image (in terms of camera pixels, since we group 4 pixels to find 1 RGB value). Figure 8 shows some possible configurations (extracted from [1, p. 8]). It is the responsibility of the programmer to configure the camera in one of these modes.

Resolution	Frame Rate	Sub-sampling Mode	Column_Size (R0x04)	Row_Size (R0x03)	Shutter_Width_Lower (R0x09)	Row_Bin (R0x22 [5:4])	Row_Skip (R0x22 [2:0])	Column_Bin (R0x23 [5:4])	Column_Skip (R0x23 [2:0])
640 x 480 VGA	150	N/A	639	479	<479	0	0	0	0
	150	skipping	2559	1919		0	3	0	3
	77.4	binning	2559	1919		3	3	3	3

Figure 8: Some compatible configurations when CAMERA_CLK = 96 MHz

The idea being to target the very format the LCD uses, so our unit must output frames at a resolution of 320×240, each pixel being a 16 bit RGB value. The red and blue colors are

5 bits wide, while the green color is 6 bits wide. Figure 9 illustrates such a conversion.

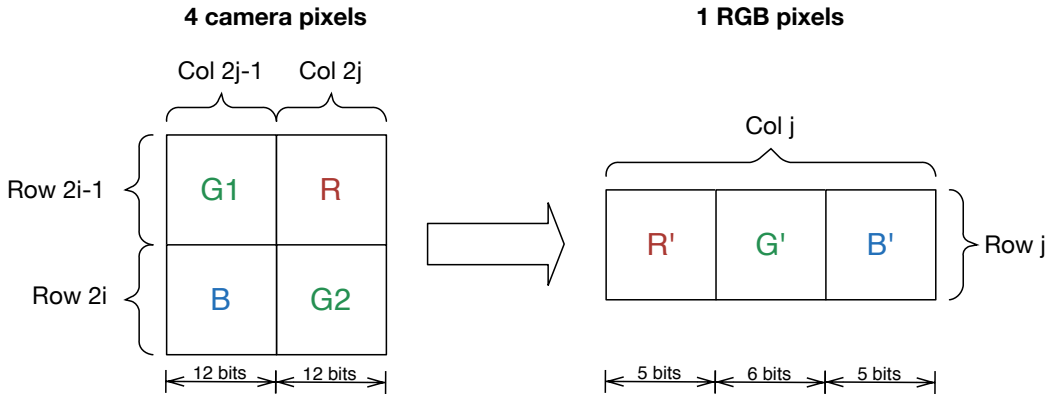

Figure 9: Pixel conversion process

5.2 Implementation details

The camera controller implementation is summarized in figure 10. It mainly consists of a 641-camera-pixel wide, falling-edge triggered, shift register. During a readout of the camera sensor, the idea is to first make it store an *entire* odd row (assuming rows are indexed from 1). Then, the *first* pixel of the following even row is stored in the shift register. As soon as the camera outputs the second pixel of the even row in above, one is able construct an RGB pixel since the two oldest camera pixels stored in the shift registers are the R and G1 camera pixels, while the most recent one is the corresponding B camera pixel, and the current output of the camera is the G2 camera pixel. Therefore, a unit is responsible for the logic that computes

$$RGB = \left\{ R, \frac{G1 + G2}{2}, B \right\}.$$

Note that all the camera pixels need to be scaled to fit into a 16-bit RGB pixel. In addition, a counter and a bit of logic are responsible to decide whether the outputted pixel is valid or not.

References

[1] Terasic. *Terasic TRDB-D5M Hardware specification*, June 2009.

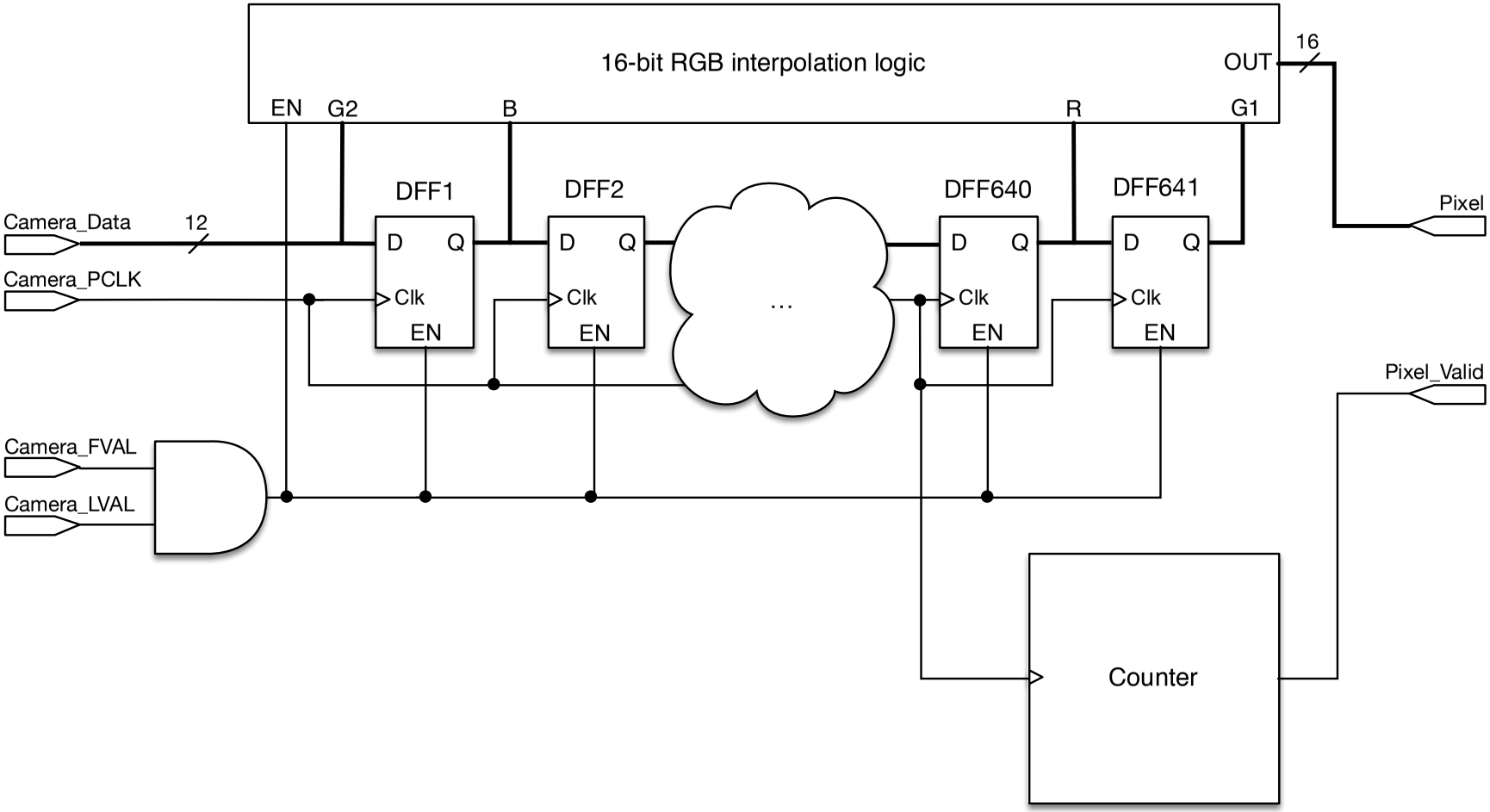

Figure 10: Camera controller block diagram