GOSLOW: DESIGN AND IMPLEMENTATION OF A SCALABLE CAMERA ARRAY FOR HIGH-SPEED IMAGING

CECILL E. ETHEREDGE

Computer Architecture for Embedded Systems (CAES)
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

July, 2016

ABSTRACT

This thesis investigates the viability of using multiple low-cost components to create something much bigger: an embedded system capable of capturing high-speed video using a scalable and configurable array of sensors. Various hard- and software components are designed, implemented and verified as part of a new embedded system platform, that, not only achieves high-speed imaging capabilities by using an array of imaging sensors, but also provides a scalable and reusable design for future camera array systems.

ACKNOWLEDGMENTS

First of all, I would like to thank prof. Marco Bekooij, for his belief in hands-on research and real-world practicality has enabled this research to go forward and develop into this thesis.

Secondly, I would like to thank my family as an endless source of optimism to explore technology, and to venture on research such as this.

A thanks to all colleagues and folks in the CAES group. I would like to thank Oğuz Meteer, whose countless hours of insights and experience provided the fundamentals for without this thesis would not have been possible, and for most of his borrowed electronics involved still have to be returned at the time of this writing.

Thanks goes out to Stephen Ecob from Silicon On Inspiration in Australia, providing the much needed hardware building blocks on more than one occasion and his continued support essential during the production of the prototype. Also thanks to Alice for sourcing the much needed BGA components in Hong Kong, and folks at Proto-Service for providing their field expertise in the final BGA assembly.

CONTENTS

1 INTRODUCTION 1
1.1 Problem definition . . . 2
1.2 Contributions . . . 3
1.3 Thesis outline . . . 4
2 BACKGROUND 6
2.1 Digital image sensors . . . 6
2.2 High-speed imaging . . . 10
2.3 Image sensor arrays . . . 12
3 HIGH-LEVEL SYSTEM DESIGN 14
4 HARDWARE DOMAIN IMPLEMENTATION 17
4.1 Sensor receiver interface . . . 17
4.1.1 Physical signaling . . . 17
4.1.2 Timing characteristics . . . 21
4.1.3 FPGA implementation . . . 24
4.2 Streaming DRAM controller . . . 25
4.2.1 Dynamic Random-Access Memory (DRAM) . . . 26
4.2.2 Access pattern predictability . . . 28
4.2.3 DRAM protocol . . . 29
4.2.4 Peak transfer rate . . . 32
4.2.5 Protocol state machine . . . 33
4.2.6 FPGA implementation . . . 36
4.2.7 Command arbitration . . . 37
4.3 Stream interleaver . . . 38
4.3.1 FPGA implementation . . . 40
4.4 Embedded control processor . . . 41
4.5 Sensor control interface . . . 42
4.5.1 I^{2}C protocol overview . . . 43
4.5.2 Camera control interface (CCI) . . . 43
4.5.3 FPGA implementation . . . 44
4.5.4 Phased start . . . 45
4.6 Readout interface . . . 48
4.6.1 FPGA implementation . . . 48
4.7 Clock domain crossing . . . 49
5 SOFTWARE DOMAIN IMPLEMENTATION 51
5.1 Embedded control . . . 51
5.2 Stream decoder . . . 53
5.2.1 Deinterleaving . . . 54
5.2.2 Bit slip correction . . . 54
5.2.3 Protocol layer decoder . . . 55
5.3 Image rectification . . . 56
5.3.1 Mathematical models . . . 57
5.3.2 Camera intrinsics and extrinsics . . . 58

5.3.3 Rectification 62
5.3.4 Camera calibration 62
6 DATAFLOW ANALYSIS 65
6.1 Throughput analysis 66
6.1.1 Scenario-aware dataflow graph 67
6.1.2 Throughput analysis 70
6.1.3 Effects on array size 71
6.2 Buffer size analysis 71
6.2.1 Latency-rate SRDF graph 72
6.2.2 Real-world case analysis 75
7 REALIZATION AND RESULTS 78
7.1 Sensor array configuration 78
7.2 Choice of hardware 79
7.3 Synthesis results 81
7.4 Measurement setup 85
7.5 Experimental results 86
8 CONCLUSION 90
8.1 Future work 92

I APPENDIX 94
A SADF GRAPH LISTING 95
B STROBOSCOPE PROGRAM LISTING 98

BIBLIOGRAPHY 100

LIST OF FIGURES

Figure 1 Block diagram showing a simplified overview of the processes involved in a modern color image sensor, from incoming scene light to final output image. 6
Figure 2 Example of a Bayer pattern encoded image (left) and the resulting full color image after the debayering process (right). 7
Figure 3 Readout architectures for conventional CCD and CMOS sensors. Photodiodes and capacitive elements are respectively colored dark green and yellow. 8
Figure 4 Rolling shutter mechanism in action. Left shows the shutter's direction of movement in terms of pixel rows. Right contains a plot of row exposure over time, clearly showing the sliding window of time. 9
Figure 5 Simplified view of two active pixels with different shutters in a CMOS sensor at the silicon level. The addition of a memory element causes occlusion of an area of the photodiode. 10
Figure 6 Talbot's high speed photography experiment. A paper disc is captured at standstill (left), spinning and captured with an exposure time of 10 ms (middle), and with an exposure time of 1 ms (right). Images courtesy of [VM11]. 11
Figure 7 The Stanford Multi-Camera array setup as described in [Wil+05]. 12
Figure 8 Abstract system diagram illustrating the hardware and software domain boundaries, their high-level components and the corresponding dataflows in the system. 14
Figure 9 Hardware domain diagram showing its various subsystems and components. Orange indicates a component external to the hardware domain, grey is a custom subsystem implemented by FPGA logic, yellow is a logic element and green represents software. Arrows indicate dataflows, with red representing the capture stage, and blue representing the readout stage. 15

Figure 10 Software domain diagram showing its various subsystems and components, including the embedded control. Orange indicates a component external to the software domain, grey is a custom subsystem implemented in software and yellow is a specific software component. Arrows indicate dataflows, with red representing the primary video dataflow. 16
Figure 11 Standard parallel CMOS sensor interface. Data lines D_0 through D_N represent the N-bit encoded pixels of the output image. 18
Figure 12 LVDS output circuitry: a current-mode driver drives a differential pair line, with current flowing in opposite directions. Green and red denote the current flow for respectively high ("one") and low ("zero") output values. 19
Figure 13 Simplified fragment of a video data transmission using MIPI CSI-2 (top) and HiSPi (bottom). LP indicates logic high and low using single-ended signaling, while all other signals are LVDS. Green respresents image data, all other colors represent control words as defined by the respective protocols. 20
Figure 14 Typical CMOS sensor layout in terms of physical pixels (left) and corresponding structure of the image frame readout data (right), as found in the Aptina MT9M021 CMOS sensor [Apt12b]. 21
Figure 15 Topology within a rank of DRAM devices. A bank consists of multiple DRAMs, each of which supplies an N-bit data word using row and column addressing. 27
Figure 16 Common layout of a DDR3 DIMM, populated at both sides with multiple DRAMs. The DIMM is typically structured in such a way that each side represents a single DRAM rank. 27
Figure 17 Simplified illustration of the pipelined DRAM command timing in case of two BL8 write requests. Green and blue indicate any commands, addresses and data bits belonging to the same request. 32
Figure 18 Original simplified DRAM protocol state diagram as in [JED08]. Power and calibration states have been omitted. 34
Figure 19 Redesigned DRAM state diagram optimized for sequential burst writes. Burst-related states are highlighted in blue. 35

Figure 20 I^2C data transmission using 7-bit slave addressing with two data words as described in [Phi03]. Grey blocks are transmissions from master to slave, white are from slave to master and green depends on the type of operation. Bit widths are denoted below. 44
Figure 21 Example CCI data transmission addressing an 8-bit register using a 16-bit address. Grey blocks are transmissions from master to slave, white are from slave to master and green depends on the type of operation. Bit widths are denoted below. 44
Figure 22 Phased start with 8 sensors plotted against time. Dark regions represent exposure duration, and light regions represent readout duration for all sensors. 46
Figure 23 The two different types of synchronizers used in our system. Green and blue represent the two different clock domains, and red acts as stabilizing logic in between. 50
Figure 24 Simplified fragment of a video data transmission using MIPI CSI-2 (top) and HiSPi (bottom). LP indicates logic high and low using single-ended signaling, while all other signals are LVDS. Green respresents image data, all other colors represent control words as defined by the respective protocols. 55
Figure 25 OpenCV Feature detection using a planar checkerboard pattern as mounted to a wall. The visual marker in the center is not actually used in the algorithm. 63
Figure 26 SADF graph representing the video streaming behaviour in our system and used as a basis for dataflow analysis. 68
Figure 27 Corresponding FSM for the *Dref* node in our dataflow model, representing write and refresh states with write as initial state. 68
Figure 28 Imaginary FSM for the *Dref* node in our dataflow model, using counters to switch between states and scenarios. 70
Figure 29 Simplified latency-rate model of our system, with the SRDF graph at the top and the corresponding task graph at the bottom. Dotted lines represent the imaginary flow of data in the system, and are not part of the actual graph. 72

Figure 30 Stages of development for the camera module PCB, from CAD design to production to pick and placing of SMD and BGA components. . . 80
Figure 31 Final prototype hardware, showing the five individual sensor modules (left) and the final setup as evaluated with the modules and FPGA hardware connected (right). 81
Figure 32 Stroboscope hardware, showing an array of 10 area LED lights driven by 10 MOSFETs and connected to a real-time embedded system. . . 85
Figure 33 Timing characteristics of the stroboscope, strobing at a 1000 Hz rate, showing two LED-driving MOSFETs being subsequently triggered at 1 ms with an error of only 0.02%. Measured using a 4 GHz logic analyzer. 86
Figure 34 Final frames as captured by four sensors of the prototype at 240 Hz after being processed in the software domain, ordered from top-left to bottom-right and showing a seamlessly overlapping capture of the timed stroboscope lights from sensor to sensor. Each subsequent frame is captured by a different sensor, at different subsequent points in time. Stroboscope lights run from bottom left to top right in time, as indicated by the white arrow in the first image. 87
Figure 35 Sensor timing diagram corresponding to our four sensor prototype at 240 Hz and the frames in Figure 34. Our stroboscope LED timing is shown on top, with each number and color representing one of the 10 LEDs that is turned on at that time for 1 ms. Dark green represents a single frame, or the light integrated by a single sensor in the time domain for a single frame exposure, i.e. the first frame captures light from 4 different strobe LEDs. Light green represents the unavoidable time in which a sensor's shutter is closed and image data is read out to our system. When all frames are combined, they in fact form a continuous integration of physical light with an exposure time and capture rate equal to 4.2 ms or 240 Hz as witnessed in Figure 34. 88
Figure 36 Timing characteristics of the I^{2}C exposure start commands as sent to different sensors in case of a hypothetical 1200 Hz (833 us) capture rate. Measured using a 4 GHz logic analyzer. 89

1 INTRODUCTION

Over the past century, the world has seen a steady increase in technological advancement in the field of digital photography. Now more than ever, consumers rely on the availability of advanced digital cameras in personal devices such as mobile phones, tablets and personal computers to snap high resolution pictures and videos, all the while cinematographers in the field find themselves with an ever increasing variety of very high-end cameras. And in between these two markets, we have witnessed the rise of an entirely new "prosumer" segment, wielding action cameras and 4K camcorders in the lower high-end spectrum to capture semi-professional video.

Some of the most obvious advancements in cameras can be found in higher resolution imaging capabilities of sensors, as well as semiconductor production techniques that have seen vast improvements over the years. As a result, sensors have become smaller, more capable, and cheaper to produce, and the cost of including such a sensor of acceptable quality in a new embedded consumer product is relatively low, especially in the lower segment of products. Furthermore, some parts of these sensors have become standardized in the industry, practically turning many imaging sensors and related technology into commercial off-the-shelf components.

Though, one of the areas of videography that has seen increasing demand but quite conservative technological innovation is the field of high-speed imaging. With the advent of television and on-line documentary series revolving around the capture of slow motion footage, consumers and prosumers have been voicing their interest in wanting to create slow motion video at home. While professional high-speed cameras have long been available, their prices are far outside the reach of these markets, often costing up to three orders of magnitude more than conventional cameras. The reason for this is quite simple: high-speed imaging puts extreme demands on an imaging sensor and its surrounding platform in terms of bandwidth, noise and timing as we will see in this thesis, therefore raising the cost price of a single high-speed imaging sensor into the hundreds or thousands of dollars, let alone the required effort and expertise to design the surrounding hardware.

Of course, this does not mean that there is no other way to provide technological innovation in this field. Taking into account that, at the lower end, imaging sensors are becoming more standardized and cheaper, the question arises as to whether it is now possible to use multiple low-end sensors to achieve the same as a single high-end sensor. This trend of combining commodity-class or low-end components is already being applied by big players such as in the online and cloud computing industry, and can prove to be cost-effective if the components can be properly combined by means of a surrounding platform of hardware and software that is scalable in the numbers.

This thesis focuses on this very idea of using multiple low-cost components to create something much bigger: an embedded system platform, that, not only achieves high-speed imaging capabilities by using an array of imaging sensors, but also provides a scalable and reusable design for future systems with increasingly larger configurations. Finally, a small-scale prototype based on this platform is produced and evaluated to assess the real-world viability of such a product.

1.1 PROBLEM DEFINITION

In order to investigate, design, implement and ultimately realize such an embedded system platform, we set out the following objectives for this thesis:

1. Investigate the viability of using a sensor array for high-speed imaging applications —
    a) Determine the trade-offs of using multiple sensors versus a single sensor;
    b) Identify the negative side-effects, and how to mitigate their effect;
2. Research and design an embedded system platform that can be used to realize a scalable array of image sensors —
    a) Determine relevant hard- and software domains and design subsystems that fit within these domains;
    b) Identify any respective bottlenecks in these subsystems and how to mitigate their effect on an implementation;
3. Realize an embedded system using this platform capable of high-speed imaging capture using a sensor array —
    a) Develop a capable hardware design implementing the embedded system platform;
    b) Implement the software domain solutions and integrate these with the hardware design;
    c) Design and implement a hardware setup that can be used to verify the high-speed imaging capabilities of the system;
    d) Verify the high-speed imaging capabilities of the system in a real-world setup.

Based on these objectives, we define our research question to be the following:
Is it viable to use an array of image sensors for high-speed imaging in an embedded form factor, and if so, which hardware and software domain components would be required to implement such an embedded system?
A solution is considered to be *viable* if the following theoretical evaluation criteria are met:

- Capable of interfacing with at least 16 image sensors at a capture rate of 60 Hz leading to an effective total capture rate of 960 Hz.
- Capable of issuing phased start commands to image sensors with a timing accuracy of at least 99%, as further explained in Chapter 4.

These criteria are especially important as they allow the production of a high-speed video using the system, as we will see in this thesis.

1.2 CONTRIBUTIONS

This thesis makes a major contribution in the field of sensor array research by designing a novel embedded system platform that is configurable, scalable and complete. This platform covers all necessary aspects in hardware and software necessary in order to enable high-speed image capture using an array of image sensors. Currently, no other known documented solutions exist describing such an embedded system.

As part of this design, this thesis introduces a number of hardware subsystems that allow for interfacing with a varying number of image sensors and capturing their corresponding data. This includes a novel and specialized streaming DRAM controller specifically targeted at writing streamed data, as well as a custom interleaver that combines multiple streams of sensor image data into single coherent DRAM write commands.

Building upon these hardware subsystems, this thesis also presents corresponding software subsystems that interface with this hardware in order to properly transform the captured image data into high-speed video. This includes a novel HiSPI sensor protocol decoder, an embedded control system used to configure the sensor array, and a video streaming pipeline that includes a custom deinterleaver as well as options for image rectification and camera calibration.

This platform effectively establishes a significant scale reduction of all prior documented efforts into an embedded and potentially handheld form factor. This kind of hands-on description has not been seen before in this field of research.

The platform is further solidified by the actual production of a prototype, showing real-world viability and presenting future opportunities to allow for an even finer product. This real-world exemplifies the actual practical use of the sensor array as an end product, which is often overlooked in existing research.

This thesis does not contribute any new ideas and algorithms in the field of image rectification or distortion caused by sensor arrays, although efforts are made to include current state of the art and proven image techniques in the software domain.

1.3 THESIS OUTLINE

This introductory chapter has so far introduced the reader to the scope and outline of this thesis, as well as the problem definition and research question.

Chapter 2 provides background information on the various topics that are fundamental to this thesis. These topics include the use and technical breakdown of digital imaging sensors, high-speed imaging and camera arrays. Insight is also provided into prior efforts in the field of high-speed imaging using camera arrays to provide perspective in how this thesis contributes to this field.

Chapter 3 describes the top-level design of the system as presented in this thesis. It provides an overview of the system in terms of the various subsystems that are to be implemented in one of the two associated domains of hardware and software.

Chapter 4 deals with the relevant subsystems in the hardware domain, making it possible to interface with a configurable number of sensors. This includes solutions for clock domain crossing, as well as providing an elaborate description of the design of the sensor receiver interface, memory controller, interleaver, embedded processor, sensor control interface and readout interface.

Chapter 5 presents the subsystems in the software domain that deal with transforming the raw data from the hardware domain into useful video data. This chapter describes the sensor protocol decoder, as well as solutions for image rectification and camera calibration.

Chapter 6 is concerned with the theoretical dataflow analysis in which dataflow models are presented that models the primary bottlenecks in the system, and analysis is performed to determine the throughput of the system and any involved buffers while discussing the viability of the system.

Chapter 7 describes the experimental results of this thesis. This chapter provides insight into the real-world hardware prototype that was produced, implementing the presented platform. A setup is used to verify the produced video of this prototype, and timing analysis is done to verify the timing accuracy of the hardware, and thus the real-world viability of the platform as a whole.

Finally, chapter 8 concludes this thesis by providing a brief summary of this thesis and including final thoughts on future work.

2 BACKGROUND

Today, our world is filled with embedded systems with wildly varying applications, often relying on the use of techniques for digital image capture and processing. The decreasing cost of digital imaging allows for more and more imaging systems to appear in different markets, from consumer photography (e.g. digital cameras and camcorders), to industrial or military (e.g. product verification, surveillance and biometrics) and far beyond.

2.1 DIGITAL IMAGE SENSORS

These digital imaging systems, such as the one presented in this thesis, rely on the use of image sensors. And although the basic concept of an image sensor seems simple, e.g. to convert incident light or *image* into a digital representation, the actual implementation of an image sensor varies considerably depending on the technology that is being used.

Figure 1 describes a simple overview of a modern color image sensor. First of all, incident light from the scene is focused by optics, typically one or more lenses. This focusing, like in most imaging systems or cameras, is necessary to ensure that a certain view of the scene is focused onto a capturing plane behind it. The light then passes through a *color filter array* (CFA) that contains a predefined pattern of colors and is then captured and converted into the analog signal domain by photodetectors directly behind the filter. The analog signals are ultimately processed by an on-chip analog signal processor and converted into the digital signal domain, after which a variety of processing techniques can be used to produce a final image.

FULL COLOR IMAGING The photodetectors or photodiodes themselves are semiconductors sensitive to charged particles and photons.

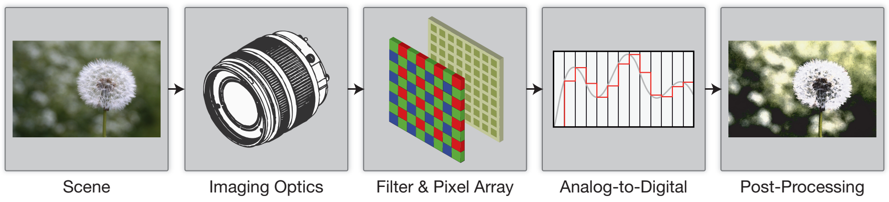


Figure 1: Block diagram showing a simplified overview of the processes involved in a modern color image sensor, from incoming scene light to final output image.

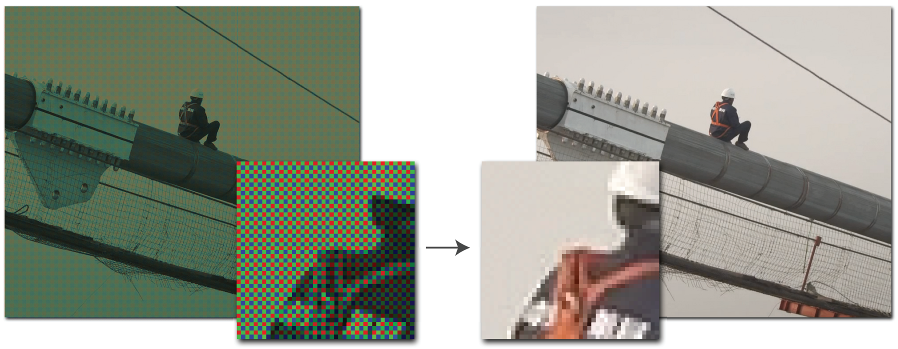


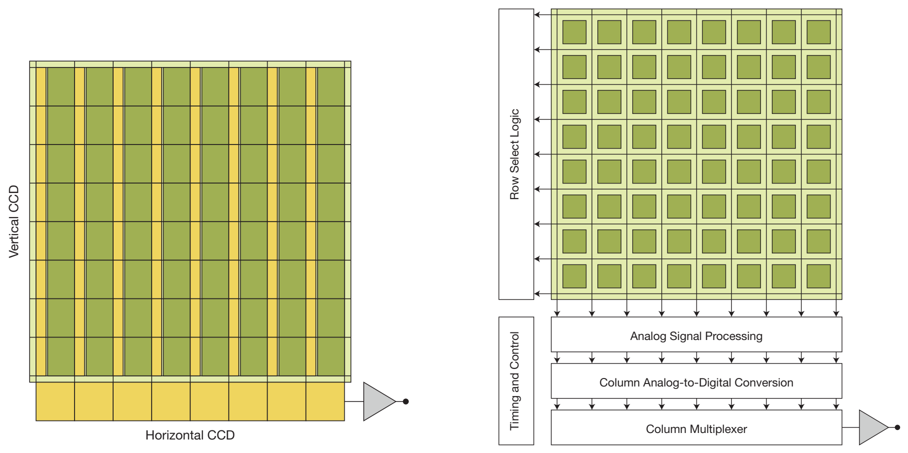
Figure 2: Example of a Bayer pattern encoded image (left) and the resulting full color image after the debayering process (right).

These work by absorbing particles and photons and emitting a voltage proportional to the indicent power, as described in [Big+06], and are thus oblivious to color as they only describe a relation between the amount of light and an analog signal. By placing a filter in front that only passes light in a certain color spectrum range, they can be utilized to only detect the amount of light of a certain color.

In practice, the color filter array that is placed in front of the photodetectors often consists of a so called *Bayer*-pattern that only allows a single color to pass through, making each single photodetector *pixel* sensitive to a predefined color. Each of the pixels in the sensor's output image only encodes the intensity of a single specific color such as red, green or blue. To produce a final full color image, a spatial interpolation process known as *demosaicing* is used, which interpolates multiple single-color pixel values to produce single full color pixels containing respective values for red, green and blue. Demosaicing is an active topic of research, and different interpolation techniques currently exist, as shown in [LGZ08]. An example of a Bayer pattern image and resulting full color image can be seen in Figure 2.

CMOS AND CCD SENSOR ARCHITECTURE As described in [EGE05], the array of photodetectors introduced before varies significantly between different types of image sensors. Many of the difference arise from the way the array is structured and read out into the analog signal domain, also referred to as the *readout architecture*, and these differences can be seen in Figure 3. Modern image sensors generally come in two different types, *complementary metal-oxide semiconductors* (CMOS) and *charge-coupled devices* (CCD), and although CCD-type sensors have traditionally been associated with high quality but high cost, recent advancements in CMOS-technology have allowed for the introduction of lower cost CMOS-type sensors approaching the quality of their CCD-type counterparts.

In a modern CCD-type sensor, the array is typically built out of photosensitive capacitors that *passively* collect and store a charge for



(a) CCD (interline transfer). (b) CMOS (active pixel sensor).

Figure 3: Readout architectures for conventional CCD and CMOS sensors. Photodiodes and capacitive elements are respectively colored dark green and yellow.

each pixel during exposure. During readout, charge is shifted out of the array, step by step, into horizontal and vertical CCDs and converted to the analog signal domain by means of an amplifier circuit, after which it is sampled and converted by the analog signal processor.

In a modern CMOS-type sensor, the array is typically built out of an *active pixel* circuit containing a photodetector and amplifier that produces an analog signal for each pixel while being exposed. During readout, the analog signals of each row of the array are selected, sampled and integrated, one by one, before being converted by the analog signal processor.

The cost-effectiveness of CMOS sensors arises from differences in the manufacturing process, also described in [Fos97], where integration of a significant amount of on-chip VLSI electronics is possible at a lower cost. This makes it possible to create single solution chip CMOS sensors that contain very elaborate analog signal processors, as opposed to expensive off-chip analog signal processing typically required for CCD sensors.

On the other side, CMOS sensors suffer from a high level of *fixed pattern noise* due to variations in the manufacturing process as well as other forms of noise such as *dark current noise* that reduce the effective *signal-to-noise ratio* (SNR) of the sensor.

In practice, CCD sensors are often seen in highly specialized markets such as astronomy and microscopy, while CMOS sensors can be found in a much wider range of applications, including mobile and consumer electronics, due to their low cost and good quality.

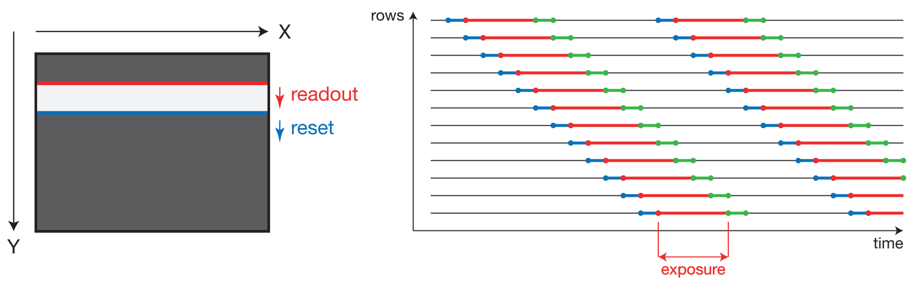


Figure 4: Rolling shutter mechanism in action. Left shows the shutter's direction of movement in terms of pixel rows. Right contains a plot of row exposure over time, clearly showing the sliding window of time.

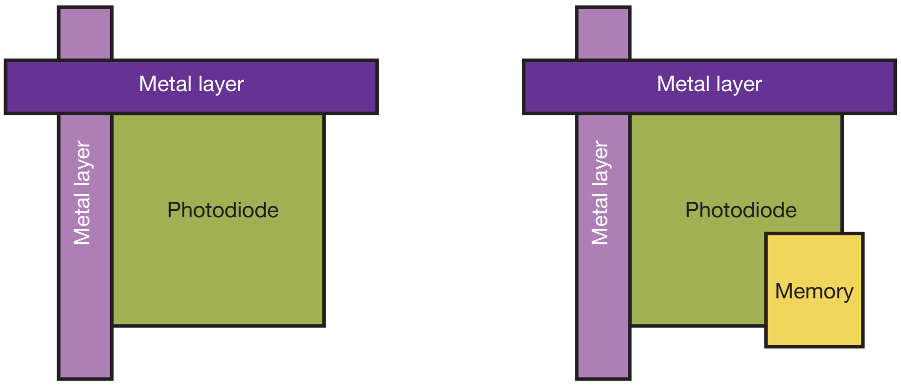
ROLLING AND GLOBAL SHUTTERS Like conventional film cameras, image sensors also require the use of a shutter that controls the exposure of the photodetectors. As mentioned before, each sensor generally has two steps of action: exposure and readout. Both before and after exposure, this shutter prevents any incident light from having an effect on the photodetectors, such that the photodetectors are only exposed to light for a predetermined amount of time.

Differences in the sensor's shutter mechanism can, in some cases, have a profound effect on the final output image of the sensor. As opposed to film cameras, sensor shutters are often electronic and operate by resetting or decharging the individual pixel circuits of the array, but the shutter timing changes significantly depending on the type of sensor used.

Note that the CCD sensor stores a charge for each individual pixel during and after exposure, essentially capturing and storing a complete image inside the sensor until it is read out of the array. This type of shutter mechanism is referred to as a *global shutter* and in a sense resembles that of a film camera. This mechanism differs greatly from that of the conventional CMOS sensor. Due to the way CMOS sensors are manufactured, only a few rows can be selected, exposed, sampled and integrated in the entire sensor at a time. This principle is known is a *rolling shutter* mechanism, and is shown in Figure 4.

The direct result of the rolling shutter is that different rows of sensor pixels are only exposed at different points in time. In other words, not all pixels in a final output image will have been captured in the same window of time. While this shift in time simply does not matter for scenes in which there is little movement, significant distortion artifacts will manifest themselves in the image as soon as fast moving objects are captured.

The introduction of an effective global shutter to CMOS sensors is one of the most sought-after features in the industry, and is an active research topic in digital imaging. Examples of global shutter implementations are [Apt12a] and [Fur+07], in which the active circuitry of



(a) Rolling shutter pixel. (b) Global shutter pixel.

Figure 5: Simplified view of two active pixels with different shutters in a CMOS sensor at the silicon level. The addition of a memory element causes occlusion of an area of the photodiode.

each pixel is augmented with a sample and hold or memory element that stores the analog signal until all rows of the entire array have been read out. Although the CMOS readout architecture still requires a row-by-row readout, the individual pixel memory elements now allow for the entire array to be exposed at the same time.

The difficulty of adding memory elements to the array lies in the fact that the memory element itself introduces varying degrees of signal contamination. In Figure 5, it can be seen that the element itself takes up physical space in the pixel array. As such, incident light can never be fully focused on the photodiode due to optical imperfections, and some may fall on the element as well. This stray light contaminates the analog signal stored in the element, and must be avoided by covering up the element with shielding. Furthermore, as the analog signal is stored in the element, unwanted dark current may accumulate during storage, requiring careful design of the underlying silicon to decrease the negative effect on the signal-to-noise ratio.

2.2 HIGH-SPEED IMAGING

Paramount to this thesis, the concept of *high-speed imaging* is best described as the still frame capture of fast moving objects, which, when used in the context of video, also implies an equivalently high capture frame rate. It is used to analyze real-world events that are difficult to capture with conventional digital cameras, such as automotive crashes, ballistics, golf swings, explosions, and so on.

High-speed photography itself has a long history, which started well before the practical invention of video. The first documented experiments were done in the 19th century, in a time when film camera shutters were still very crude and slow mechanisms, hence freezing the motion of fast moving objects was difficult. As described in

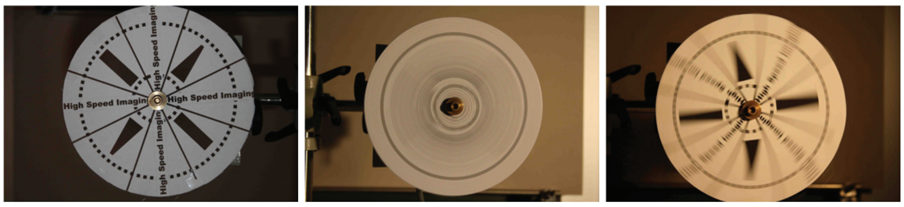


Figure 6: Talbot's high speed photography experiment. A paper disc is captured at standstill (left), spinning and captured with an exposure time of 10 ms (middle), and with an exposure time of 1 ms (right). Images courtesy of [VM11].

[Ram08], these early experiments featured a setup with a fast spinning paper such that the text on it was unreadable by the human eye. However, by using an arc flash with a duration of 1/2000 of a second, still images with sharp readable text could now be reliably produced.

Figure 6 shows a modern version of the spinning disc. In order to capture sharp still images, the *exposure* of the scene's light to the camera must be minimalized: either by opening and closing the shutter very quickly, or by using a single light source that flashes for a fraction of a second. If the exposure is too long, fast moving objects in the image will appear fuzzy and blurred due to the accumulation of light at different positions along the object's frame of movement.

A multitude of different high-speed imaging cameras are available today and include not only high-end cameras for industrial and scientific use, but also low-end consumer cameras, spurred by a recent consumer and professional interest in high-speed photography. Examples include the fps1000 and Edgertronic projects, both of which have been successfully funded and delivered through crowdfunding platforms such as Kickstarter and feature a maximum high-speed capture rate of respectively 550 Hz and 701 Hz at 720p resolution as in [Per15]. In comparison, high-end cameras such as the Photon Fastcam SA series are capable of capturing rates up to 7000 Hz at a 1024x1024 resolution, shutter times in the microseconds, and are often the first choice in scientific research dealing with physical and chemical phenomena such as in [Gül+12] and [BS14]. Nevertheless, these high-end cameras come at a high price and are often only available on a daily rental basis, placing them out of reach for normal consumer and "prosumer" markets. Despite the relatively low-end specifications of products such as fps1000 and Edgertronic, price tags of respectively $1500 and $5495 and successful crowdfunding campaigns have shown that there might be an untapped market demand underneath the existing high-end products.

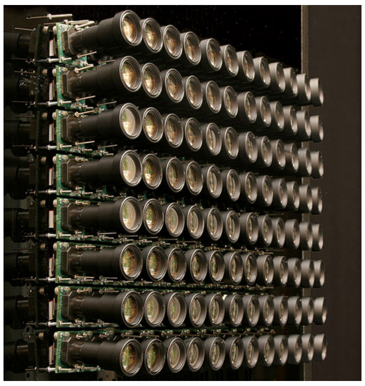


Figure 7: The Stanford Multi-Camera array setup as described in [Wil+05].

2.3 IMAGE SENSOR ARRAYS

One aspect that virtually all high-speed imaging products have in common today is that they are designed around a single high-speed CMOS imaging sensor. Depending on the target market and cost price of the final product, a design choice is made from a variety of high-speed imaging sensors with different low- or high-end performance characteristics and matching cost prices such as the CMOSIS series ([WM15]) and Sony IMX or Exmor series ([Son11]). These sensors are often at the forefront of CMOS technology, incorporating cutting edge technology such as back-illuminated sensors, novel global shutters and capacitive storage elements to mitigate the sensitivity and bandwidth issues that come into play when imaging at high speed and short shutter time. This in turn demands a high sensor unit cost price, which in turn dictate the minimum cost price of these camera products.

Keeping in mind that conventional high-speed camera products are thus still dependent on one or more relatively high cost components, we investigate the possibility of using an alternative design that removes the dependency on these high cost components. We consider an alternative design that does not use a single high cost sensor, but rather multiple low cost sensors, configured in an array, to achieve these high-speed imaging capabilities.

One of the earliest projects involving the use of multiple cameras in a similar field of research is [Wil+01] at Stanford University. In this thesis, light field data is captured using an array of up to 64 custom camera boards connected by IEEE1394 bus to one or more video processing host PCs. Here, the camera boards are custom designed individual single-board computers containing a microprocessor, IEEE1394 chipset, MPEG2 video encoder, Xilinx Spartan FPGA and Omnivision CMOS image sensor. A continuation of this thesis is described in [Wil+04], in which an identical hardware setup, also seen in Figure 7, is used for high-speed imaging and which is practically the first documented research involving a camera array for high-speed imaging. The camera array consists of 52 Omnivision sensors, each capturing at a rate of 30 Hz, providing a video stream that is further processed by two or more host PCs. The research itself is mostly concerned with dealing with the side effects of using a camera array for high-speed imaging, and therefore provides fundamental insights in correcting geometric distortion, rolling shutter distortion and camera timing, which we will also cover in this thesis.

Further in-depth research on this particular project at Stanford is described in [Wil05]. Here, a number of possible applications for camera arrays are researched and include synthetic aperture photography (SAP), spatiotemporal view interpolation (SVI) and high-speed imaging. While SAP and SVI are useful in the sense that they provide image aperture manipulation and improved image quality, we focus on the high-speed imaging research as this closely fits the scope of this thesis. Finally, the Stanford setup is further refined in terms of hardware and system architecture in [Wil+05], and image quality is evaluated by comparing with a conventional digital camera showing encouraging results.

3 HIGH-LEVEL SYSTEM DESIGN

The purpose of the embedded system platform presented in this thesis is to interface with an array of image sensors in order to allow high-speed imaging. The platform, or system, is composed of a number of novel custom hardware and software components that implement the required functionality to ultimately produce a video stream containing high-speed images.

In this chapter, an overview of the system as a whole is presented at increasing levels of detail. We simply begin by establishing the different domains that make up this platform:

- *Hardware domain*. This domain contains all components implemented in hardware, or more specifically, a reconfigurable FPGA platform. The use of a FPGA allows for rapid design and prototyping, and is thus fundamental to the platform.
- *Software domain*. This domain contains all components implemented in software. The components represent the programs that run on CPU-based architectures either embedded in or connected to the hardware domain.

In Figure 8, a system diagram can be seen describing these domains and the components involved at a high and abstract level. In this diagram, the primary dataflow is clearly illustrated and effectively begins in one or more image sensors, flows through the FPGA and RAM in the hardware domain, and ultimately ends up at the host in the software domain. A number of secondary dataflows also exist to ensure that the primary dataflow is controlled. Furthermore, while the separation of hardware and software domain boundaries is generally clear, the software domain also covers a small part of software used within the hardware domain, namely that of the embedded control processor as we will see later on in this thesis.

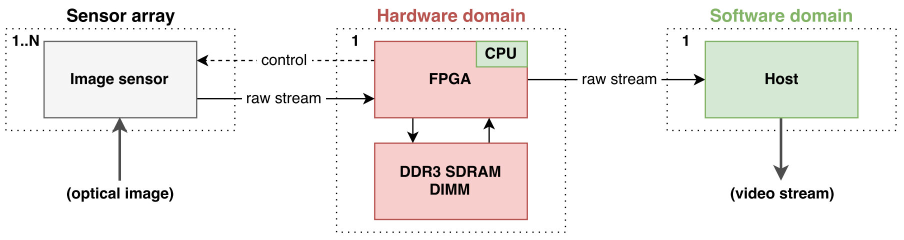


Figure 8: Abstract system diagram illustrating the hardware and software domain boundaries, their high-level components and the corresponding dataflows in the system.

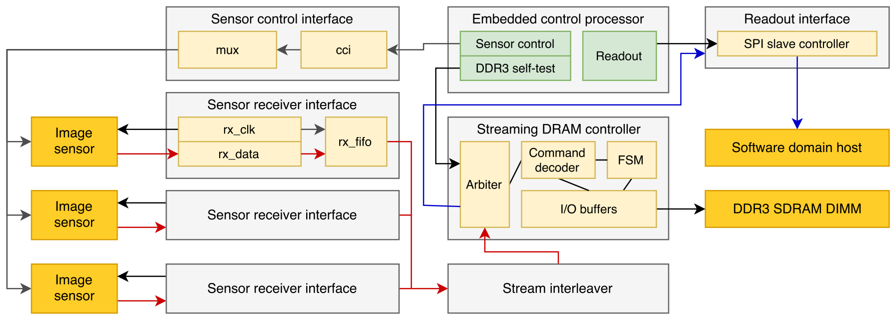


Figure 9: Hardware domain diagram showing its various subsystems and components. Orange indicates a component external to the hardware domain, grey is a custom subsystem implemented by FPGA logic, yellow is a logic element and green represents software. Arrows indicate dataflows, with red representing the capture stage, and blue representing the readout stage.

HARDWARE DOMAIN The hardware domain represents the bulk of components in the system, and for good reason. At the highest level, the hardware simply performs two dataflow stages: first to *capture* as much raw data as possible from all image sensors, and secondly to *read out* and transmit all this captured data to the software domain for further processing. Note that these two stages do not have to occur at the same time due to the nature of video capturing where data is generally first captured to a storage medium, and then offloaded to another system for further use.

The *capture stage* is assumed to be a unique and critical stage that cannot be interrupted or paused, as this may lead to data corruption, making it *hard real-time* by definition as per [But11]. Needless to say, the hardware domain must then provide guarantees that no overflows ever occur in the system in order to avoid fatal data corruption. In contrast, the *readout stage* occurs after the capture stage has completed, and can occur at any speed. Since it is fully off-line and non-critical, this thesis therefore imposes no timing constraints on this stage.

All components involved in the hardware domain can be seen in Figure 9. As has been mentioned before, the components or subsystems in this domain are implemented as FPGA logic using VHDL or Verilog together with vendor-specific primitives. All components are in fact custom implementations especially designed to implement critical functionality for this platform. In Chapter 4, each of these subsystems is described on a functional level. Note that due to the gritty details involved, implementation-specific details such as VHDL or Verilog code do fall outside the scope of this paper.

Next to the FPGA logic (grey components), Figure 9 also illustrates a number of components (in orange) that are external to the domain but nevertheless critical to the functionality of the system. Interfaces to these components are provided by the FPGA logic inside the hard-

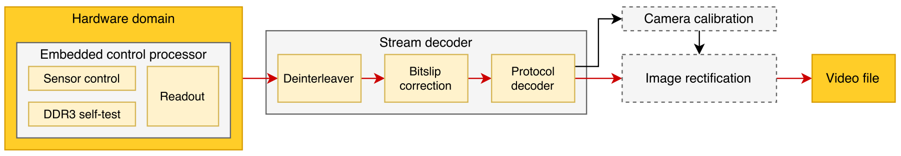


Figure 10: Software domain diagram showing its various subsystems and components, including the embedded control. Orange indicates a component external to the software domain, grey is a custom subsystem implemented in software and yellow is a specific software component. Arrows indicate dataflows, with red representing the primary video dataflow.

ware domain, enabling the primary data to flow from image sensor to the software domain.

SOFTWARE DOMAIN The software domain largely starts where the hardware domain ends, implementing the necessary components in software to produce a video stream or file containing the captured high-speed images. It is also concerned with performing command and control in order to set up the dataflow at various points in the system, as well as providing auxiliary functionality such as DRAM self-testing.

Figure 10 shows the components and subsystems that make up the software domain. Here, all subsystems except for those in the dotted lines represent new custom pieces of software that have been explicitly designed for this platform. Note that the diagram also shows a number of components inside the hardware domain. As mentioned before, the software domain also covers the software specially written for the control processor that is embedded inside the hardware domain. All subsystems except for those in this embedded part run on a host computer and are connected to each other through a video stream pipeline software architecture known as GStreamer. Further details are provided in Chapter 5.

4 HARDWARE DOMAIN IMPLEMENTATION

In the previous chapter, an overview of the system was provided that explained the various domains involved in the design. This chapter will focus specifically on the details of the hardware domain, as illustrated in Figure 9. Recall that each of the subsystems is a custom design specially designed to provide functionality specific to our system. Each of the sections in this chapter will describe the background, design choices and functional implementation of each of the subsystems.

4.1 SENSOR RECEIVER INTERFACE

The very first input of the entire system and thus the hardware domain is in fact the sensor array, consisting of a varying number of image sensors. Each of these sensors are physically connected to the FPGA that implements our hardware domain. We therefore require a subsystem in our hardware domain that implements an interface capable of capturing all raw data as physically received by the FPGA.

This section introduces a sensor receiver subsystem that interfaces with a single image sensor. This subsystem is designed such that it can then be instantiated multiple times, for each individual sensor, such that all raw data coming in from the entire sensor array can be captured.

We first provide some background on the physical signaling standards relevant to the receiver, after which the timing parameters relevant to the receiver are introduced. Finally, the FPGA logic elements necessary for the functionality of the receiver are described.

4.1.1 *Physical signaling*

Conventional CMOS image sensors have traditionally been equipped with a *parallel interface* that utilizes simple single-ended signaling to encode all the necessary video data. This parallel interface, which can also be seen in Figure 11, contains a certain number of parallel data lines with a corresponding clock and synchronization signals. Typical configurations are 8 or 12 data lines, equivalent to the sensor's pixel bit resolution, where each individual line represents a single bit of the image pixel, and full pixels are thus transmitted on every rising edge of the corresponding pixel clock. The advantage of the parallel interface is obvious: it allows for very simple interfacing with low complexity receivers.

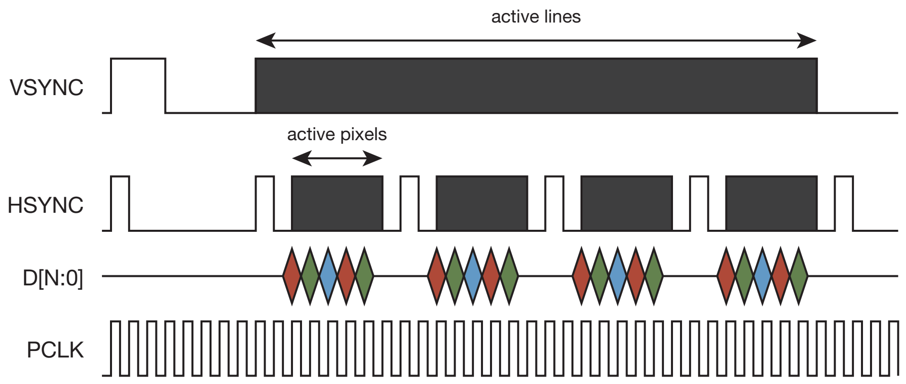


Figure 11: Standard parallel CMOS sensor interface. Data lines D_0 through D_N represent the N-bit encoded pixels of the output image.

The increase of sensor resolution, bandwidth and signal integrity requirements have however led to the adoption of *serial interfaces* such as Low Voltage Differential Signaling (LVDS). A typical differential interface circuit can be seen in Figure 12 and shows a current-mode driver, two equal transmission lines with opposed polarity and equal impedance, and a comparator at the receiving end.

Although serial transmitters require a higher clock frequency, they allow for a significant improvement in signal integrity of the overall system due to a number of critical factors, such as:

- **Common-mode noise.** For differential signaling, common-mode noise is injected on both transmission lines and is rejected at the receiving end as only the differential value is sampled.
- **Switching noise.** The design of differential current-mode drivers allows for a reduction of (simultaneous) switching noise and ringing in the transmission lines as well as the overall system.
- **Crosstalk and electromagnetic interference.** Differential signaling radiates less noise into the system than single-ended signaling since magnetic fields are canceled out by the two transmission lines.

Because differential signaling reduces any concerns regarding noise, it is possible to use lower voltage swings in the transmission lines, typically 350 mV for LVDS and 150 mV for SubLVDS. In turn, reducing the voltage allows lowers the overall power consumption and also allows higher data rates to be used, as data can be switched more quickly. Furthermore, the use of current-mode drivers creates an almost flat power consumption across frequency, such that the frequency can be increased without otherwise exponentially increasing power consumption [Gra04].

The resulting differential serial interface as implemented in an image sensor typically replaces all single-ended signals of the parallel

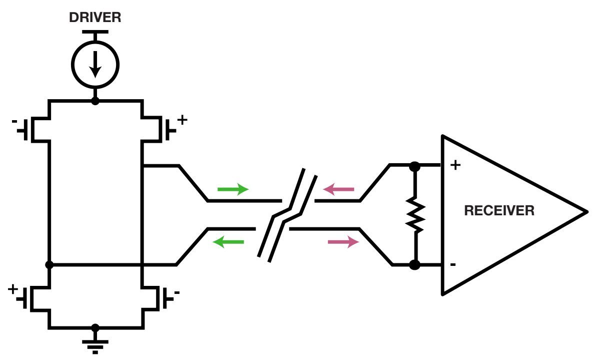


Figure 12: LVDS output circuitry: a current-mode driver drives a differential pair line, with current flowing in opposite directions. Green and red denote the current flow for respectively high ("one") and low ("zero") output values.

interface with just a differential clock line and only one or more differential data "lanes" onto which all relevant data is serialized. The number of required I/O pins per sensor are thus minimized, requiring less complex board designs. The control and data signals of the earlier parallel interface shown in Figure 11 are instead encoded as a continuous serial stream of bits, and specific leading and trailing bit patterns distinguish between different parts of data, as will be explained in Section 4.1.1.

There are many varieties of LVDS on the market today, each with different characteristics suited for certain application or vendor specific solutions. LVDS as defined in the original *TIA/EIA-644* standard is generally considered to be the most common and original specification from which many of these implementations are derived. Varieties include *BLVDS (Bus LVDS)*, *M-LVDS (Multipoint LVDS)*, *SCI (Scaleable Coherent Interface)* and *SLVS (Scalable Low-Voltage Signaling)* and these varieties mainly exhibit differences in electrical characteristics such as voltage swing, common mode voltage, ground reference and output current [Gol11].

Note that this thesis focuses solely on the use of LVDS-capable sensors in its broadest sense, where the use of the term LVDS refers to LVDS and any of its many derivative transmission standards.

INDUSTRY-STANDARD INTERFACES Increased market demand for sensors with higher image resolution, greater color depth and faster frame rates means that current processor-to-camera sensor interfaces are being pushed to their limits. As far as these sensor interfaces go, bandwidth is not the only important design choice: robustness, cost-effectiveness and scalability are among the factors that have an important weight, especially for mobile devices [MIP16a].

As such, the *MIPI (Mobile Industry Processor Interface) Alliance* was established to develop a new industry-standard interface, and has

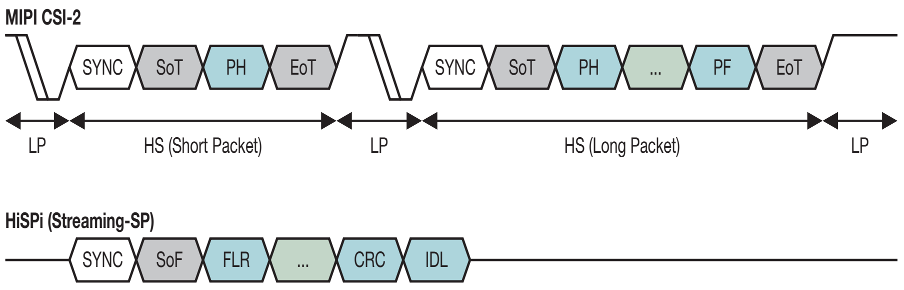


Figure 13: Simplified fragment of a video data transmission using MIPI CSI-2 (top) and HiSPi (bottom). LP indicates logic high and low using single-ended signaling, while all other signals are LVDS. Green respresents image data, all other colors represent control words as defined by the respective protocols.

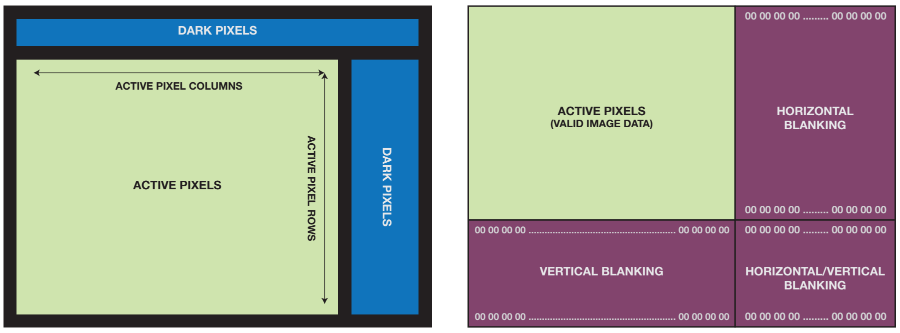
been rapidly replacing conventional parallel interfaces with their *CSI*, *CSI-2*, *CSI-3* (Camera Serial Interface) specifications. These CSI specifications define the use of a physical communication layer and protocol layer capable of transporting arbitrary sensor data to the receiver [Lim+10].

The MIPI Alliance promotes their CSI specifications as open standards, though one should note that all CSI specifications are confidential. The actual degree of openness and risk of infringement for these specifications is therefore unclear[1]. It is possible that this policy has indirectly resulted in the introduction of HiSPi, an alternative interface standard that is otherwise very identical to MIPI CSI-2 and designed by Aptina Imaging for use in their own line of LVDS-capable sensors [Sem15a]. As CSI-2 (and thus HiSPI) is currently the most commonly used standard for the class of sensors most relevant to this thesis, we will not further elaborate on the use of CSI, CSI-3 or any other recently introduced standards. Instead, we briefly elaborate on the physical layers of CSI-2 and HiSPi as far as the sensor receiver interface is concerned. In Section 5.2, we further describe the protocol layer of these standards.

The CSI-2 and HiSPi specifications define a *physical layer* that specify the transmission medium, electrical circuitry and the clocking logic used to accomodate a serial bit stream. In the case of CSI-2, this physical layer is referred to as the *D-PHY* and is composed of between one and four (generally) unidirectional data lanes and one clock lane, capable of transmission in one of two modes known as *Low Power (LP)* and *High-Speed (HS)* mode.

In HS mode, each lane is terminated on both the transmitter and receiver side and is driven by a SLVS transmitter, allowing for high-speed bursts of protocol packets. In LP mode, all lanes are instead

1 MIPI is an independent, not-for-profit corporate entity that requires an active membership for the disclosure of any of the confidential specification documents. Memberships require payment of dues ranging from $4.000 to $40.000, yet any company can apply to join [MIP16b]



(a) Physical CMOS pixel layout. (b) Spatial layout of image frame.

Figure 14: Typical CMOS sensor layout in terms of physical pixels (left) and corresponding structure of the image frame readout data (right), as found in the Aptina MT9M021 CMOS sensor [Apt12b].

driven as single-ended LVCMOS wires, allowing for control signal transmission at decreased power consumption. HS transmissions occur in bursts, and each burst is preceded and followed by various control signals in LP mode to signify the start and end of transmission to the receiver, while the HS transmission itself contains a synchronization sequence of bits used to align the receiver's deserializer to the correct word boundary [Lim+10][Xil14].

The HiSPi standard omits the use of a single-ended low power mode in its entirety and only uses a differential SLVS driver in the physical layer [Sem15b], further reducing the receiver's complexity in case of full compatibility. An example of a data transmission for both standards can be seen in Figure 13.

4.1.2 *Timing characteristics*

As explained in the previous section, image sensors with differential serial interfaces come equipped with an on-chip serializer that transmits binary data at a certain bit rate. This bit rate, referred to as the output serializer frequency, is fully defined by the characteristics of the image sensor, where modern differential image sensors typically operate in the range of 100 and $1000\,\text{MHz}$. Characteristics are often configurable by the developer and include the image resolution, frame rate, bit depth and others.

Since the timing of a sensor is defined by its output serializer frequency, which in turn is defined by the configurable characteristics of the sensor, insight into the sensor timing can be gathered by looking at these characteristics in detail.

Figure 14 shows a typical physical CMOS sensor pixel layout, in which only a smaller region contains active pixels that will actually capture image data. Pixels outside the active region can be used for different auxiliary purposes: so called *dark pixels* are used for black-level calibration, while *barrier pixels* serve no purpose other than to sit in between other types of pixels. This makes it necessary to define a 2D region or *window* of active pixels that will capture the final image.

All physical pixels within the active region are then sampled during image readout and padded with horizontal and vertical *blanking pixels*, as can be seen in Figure 14b. Both the active pixels and blanking pixels make up the final *image frame* as it is transmitted, and the blanking regions serve no other purpose than to ensure correct timing between the sensor capture frame rate and data transmission, as we will see later on.

As part of this layout of sensor pixels, we can explicitly define the following configurable characteristics or sensor parameters:

- *Active pixel columns*, N_{active_cols}: the number of active pixels or horizontal columns in the sensor array that are transmitted for every row.
- *Horizontal blank columns*, N_{blank_cols}: the number of blank pixels or horizontal columns in the sensor array that are transmitted for every row after all active columns.
- *Active pixel rows*, N_{active_rows}: the number of active pixel rows in the sensor array that are transmitted. Together with N_{active_cols}, this defines the 2D region of active pixels.
- *Vertical blank rows*, N_{blank_rows}: the number of blank vertical rows in the sensor array that are transmitted after all active rows.

Note that these characteristics only describe the various imaging regions. Obviously, each image pixel contains an amount of data, and each image frame consisting of pixels is to be captured at a certain exposure time, and transmitted at a certain rate and over a certain physical interface. We thus further identify the following typical characteristics or sensor parameters that directly affect sensor timing:

- *Pixel bit depth*, N_{bit_depth}: the amount of bits per pixel that encode the pixel's color value.
- *Serial lanes*, N_{lanes}: the number of individual differential serial lines used to serialize data.
- *Exposure time*, $t_{exposure}$: the time in which the (global) shutter is opened, all pixels are exposed, sampled and integrated.
- *Pixel clock*, f_{PIXCLK}: the rate at which individual pixels of a frame are transmitted.

As soon as the layout of image frame has been defined, it is fairly trivial to calculate the timing parameters relevant to the frame:

$$t_{row} = (N_{active\_cols} + N_{blank\_cols}) \times \frac{1}{f_{PIXCLK}}$$

Here, t_{row} represents the total time spent transmitting a single row of the image frame, which can then be used for all rows:

$$t_{frame} = (N_{active\_rows} + N_{blank\_rows}) \times t_{row}$$

Where t_{frame} is the total time spent transmitting the entire image frame. Finally, the *frame rate* (in Hz) can then be calculated by adding the actual exposure or integration time:

$$f_{fps} = \frac{1}{t_{frame} + t_{exposure}}$$

Since the frame data is to be transmitted over the serial interface, we calculate f_{SERIAL} or the *serial bit rate* at which individual bits are transmitted over a single serial lane:

$$f_{SERIAL} = \frac{f_{PIXCLK} \times N_{bit\_depth}}{N_{lanes}} \tag{1}$$

TYPICAL EXAMPLE As an example of the above timing calculations, we consider a 1280×720 (*720p*) resolution CMOS image sensor with the following input parameters:

- $N_{active\_cols} = 1280$ pixels, $N_{blank\_cols} = 370$ pixels.
- $N_{active\_rows} = 720$ pixels, $N_{blank\_rows} = 28$ pixels.
- $N_{bit\_depth} = 12$ bits, $N_{lanes} = 2$ lanes.
- $t_{exposure} = 3$ ms
- $f_{PIXCLK} = 74.25$ MHz

The row time t_{row} and frame time t_{frame} are then calculated as follows:

$$t_{row} = (1280 + 370) \times \frac{1}{74.25\,\text{MHz}} \approx 22.22\,\mu\text{s}$$

$$t_{frame} = (720 + 30) \times t_{row} \approx 16.67\,\text{ms}$$

The sensor integration overlaps with the capture period in such a way that as long as $t_{exposure} < t_{row}$, $f_{fps} = t_{row}$. Since this is the case in the above example, the final frame rate f_{fps} and serial bit rate f_{SERIAL} are as follows:

$$f_{fps} = 16.67\,\text{ms} = 60\,\text{Hz}$$

$$f_{SERIAL} = \frac{74.25\,\text{Mhz} \times 12\,\text{bits}}{2\,\text{lanes}} = 445.5\,\text{MHz}$$

As can be seen, careful balancing of the various parameters is required whenever the frame rate is expected to be at a specific final value. As such, the input parameters in this example were tweaked to achieve a final frame rate of 60 Hz.

Since this thesis is concerned with high-speed capture, it is beneficial to maximize the sensor's capture rate f_{fps} as well as the image resolution through N_{active_cols} and N_{active_rows}. In other words, a sensor is expected to capture as much image data as possible at its highest possible frame rate. This is done by setting the input pixel clock f_{PIXCLK} to the highest possible frequency, and setting the blanking pixels N_{blank_cols} and N_{blank_rows} to their minimum values.

4.1.3 *FPGA implementation*

Now that the timing characteristics for a single sensor have been identified, a matching FPGA logic design is necessary in order to capture the raw sensor data coming in from the physical interface.

The required logic we have designed is fairly straightforward and uses vendor-specific primitives to set up an interface with the physical layer. The following generic elements make up this design:

- *Differential clock input* or rx_clock. Buffer logic element that interfaces with the differential clock signal coming in from a sensor.
- *Differential data input buffers* or rx_data. Buffer logic element that interfaces with a single differential DDR data lane coming in from a sensor.
- *Cross clock FIFO* or rx_fifo. FIFO logic element that interfaces with all of the above elements as input, and provides an output in a different clock domain.

Using the elements described above and the typical 2-lane sensor example outlined in the previous chapter, the sensor receiver logic typically consists of a single rx_clock instance, two rx_data instances and a single rx_fifo instance. A rx_clock instance is always necessary, as the rising and falling edge transitions of the differential input clock are used to sample valid data from the differential data lanes in each rx_data instance.

The need for rx_fifo becomes clear if one assumes the likely fact that each sensor operates as a separate unit that is completely independent from the rest of the system. The incoming differential data clock is therefore unrelated to any other clock and the corresponding data has to be moved over from the sensor's clock domain into the system's primary clock domain. In order to achieve this, rx_fifo is implemented as a two-clock FIFO synchronizer, as explained in Section 4.7, clocking the output data safely in the system's own primary clock domain for further use.

The rx_clock and rx_data element implementations make use of vendor-specific primitives due to their low-level interfacing, e.g. in Xilinx systems, where IBUFDS differential buffer primitives and ISERDES2 deserializer primitives are used to realize the sensor receiving interface in the FPGA.

The deserializer primitive deserializes the bit stream coming in from the differential buffer primitives into words of S bits wide, typically 4 or 8 bits dependent on the FPGA vendor. The deserializer primitive is directly connected to the write end of the rx_fifo instance, and therefore imposes an identical FIFO word width of S and arbitrary depth. The read end of the rx_fifo instance is then exposed as an output bus of S bits wide, accompanied by an *enable* signal that indicates that the subsystem connected to this bus is reading from the FIFO. To avoid unnecessary filling the FIFO after the image sensor has started streaming data but before the FIFO is being read out, the enable signal is connected to both the *read enable* and *write enable* signals of the FIFO.

4.2 STREAMING DRAM CONTROLLER

This section focuses on the design and implementation of the system's specialized memory controller. The controller is a novel design in the sense that it is specifically designed to handle linear DRAM access patterns, as opposed to conventional random access as is typical for DRAM controllers.

In [Goo+16], it is shown that the drive behind the developments for conventional DRAM controllers is essentially improvement of the typical or average-case throughput. As explained in [Axe+14], conventional DRAM controllers also do not provide any form of temporal isolation, e.g. their latency is largely influenced by other tasks or history, leading to unpredictable throughput. Since our system operates in the hard real-time domain, the focus lies on improving the worst-case behaviour instead. Combining this with the streaming nature of our system dataflow as hinted in previous sections in fact creates an ideal and fully predictable DRAM use case. The design presented in this section leverages this behaviour in order to achieve optimal sustained throughput that is largely predictable as will later be shown in Chapter 6.

4.2.1 *Dynamic Random-Access Memory (DRAM)*

Today, *Double Data Rate Synchronous Dynamic Random Access Memory* (DDR SDRAM) is commonly used in embedded systems due to its relative low cost per bit when compared to other conventional volatile memories such as SRAM [Kle07]. It is the de facto choice for embedded system boards carrying FPGAs and System-on-Chips, fulfilling the need for mebi- to gibibytes of low cost addressable memory. Manufacturers such as Xilinx, Altera, Lattice Semiconductor provide a variety of options to use DDR memory technology for their devices, ranging from hardware memory controllers to fully synthesizable memory controllers, as in [Lat15], [Alt15a] and [Geo07]. The use of DDR SDRAM technology for our data capture memory storage therefore seems obvious in terms of practicality.

Apart from the obvious benefits of DDR SDRAM, there is a great variety of DDR device generations and system arrangements to choose from when designing a memory controller. Since the number of storage bits contained in a single DRAM device or chips at any given instance is inherently limited, the use of multiple organized DRAM devices provides a necessary degree of storage scalability in the system. From a (traditional) memory controller's point-of-view, these DRAM devices are organized as follows [JNW07]:

1. *Channel*. A channel represents the use of split data buses, typically used to address multiple memory modules, e.g. a dual-channel configuration that combines two 64-bit modules into a single 128-bit data bus.

2. *Rank*. A set of individual DRAM devices, addressed using the available chip select signals.

3. *Bank*. A set of independent memory arrays within a DRAM device, operating independently or concurrently.

4. *Row*. A set of "horizontal" storage cells within a DRAM device, activated in parallel.

5. *Column*. A set of "vertical" storage cells within a DRAM device, the smallest addressable unit of memory, equal to the data bus width.

The organization levels above show that the memory topology is quite complex and the data and address bus are typically structured in such a way that a single *row* or *column* actually spans across multiple DRAM devices, as can be seen in Figure 15.

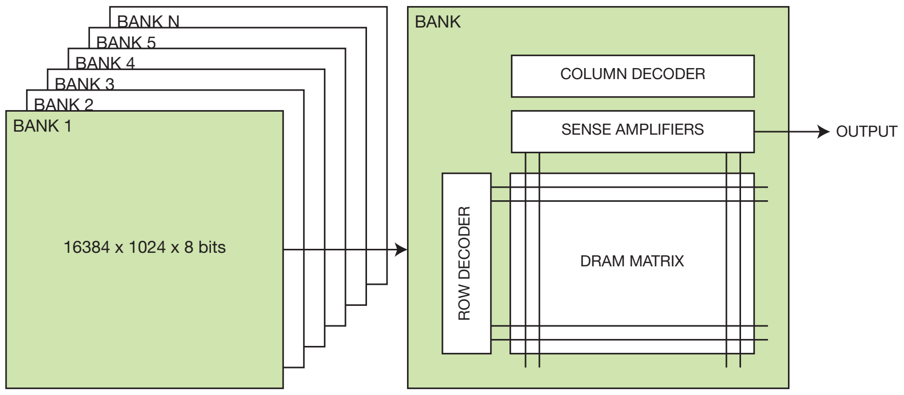


Figure 15: Topology within a rank of DRAM devices. A bank consists of multiple DRAMs, each of which supplies an N-bit data word using row and column addressing.

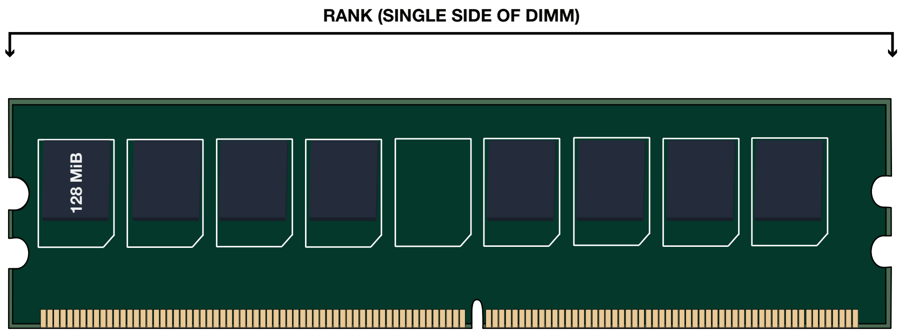


Figure 16: Common layout of a DDR3 DIMM, populated at both sides with multiple DRAMs. The DIMM is typically structured in such a way that each side represents a single DRAM rank.

These DRAM devices exist in many types ranging in capacity, number of rows or columns, but also data bus width (e.g. 4-bit, 8-bit or 16-bit). For a single *rank* of memory, many device configurations are therefore possible, as long as the combined memory data bus is 64 bits.

IN-LINE MEMORY MODULES The first design choice for the memory controller is concerned with the use of *in-line memory modules*. As noted before, single DRAM devices simply do not provide adequate memory storage, and while the use of multiple DRAM devices lessens this problem to a certain degree, it still does not solve the issue of storage scalability.

This is where the use of in-line memory modules such as *Dual In-line Memory Modules* (DIMMs) or *Small Outline DIMMs* (SO-DIMMs) provide a solution: these are standardized boards holding a variable number of DRAM devices and can be plugged into a matching DIMM socket. The DIMM, as used in virtually all modern desktop computers, allows for easy replacement of DRAM devices, thus providing a highly configurable and scalable means of memory storage. These modules are typically populated at both sides, with each side representing a *rank* of DRAM devices, as can be seen in Figure 16.

FPGA and SoC manufacturers generally provide the necessary building blocks to enable the use of DDR SDRAM devices, as seen in [Lat15], [Alt15a] and [Geo07]. Most building blocks are however optimized for conventional memory use in embedded systems: mostly random access, and addressing one or only a few on-board DRAM chips in a single rank. As a result, support for DIMMs is often missing or otherwise undocumented, especially for the lower price class FPGA platforms. Instead, we will choose to design a custom soft-core DRAM controller specifically tailored to the requirements of our system and using a virtually ideal use case.

4.2.2 *Access pattern predictability*

The general purpose of the DRAM controller is to provide an intermediate storage facility with enough bandwidth to be able to temporarily store all incoming data from the interleaver subsystem, up until the point when all data has been consumed by the readout interface. The general flow of storage in the system is therefore quite extraordinary:

1. Data capture (streaming writes, maximum throughput)
   a) Interleaver starts producing data.
   b) Memory controller consumes interleaver data and linearly writes to memory storage.
   c) Maximum memory storage capacity is reached or stream is otherwise aborted.
   d) Interleaves stops producing data.
2. Data readout (streaming reads, controllable throughput)
   a) Memory controller linearly reads from memory storage and starts producing data.
   b) Readout interface consumes memory controller data and writes to external interface.
   c) Maximum memory storage capacity is reached or stream is otherwise aborted.

Throughout steps 1a to 1d, the stream of data is captured and stored linearly (e.g. from front to back) into memory storage. Steps 2a through 2c are concerned with the readout and transmission of the captured data to an external interface.

Here, the distinction between the data capture and readout steps is especially important, as they differ significantly in throughput and thus in bandwidth requirements. The data capture occurs as a continuous data stream running at maximized throughput, where loss of data means a loss of captured images, and must thus be avoided at all cost. This is in stark contrast to the data readout, which is a fully controllable and interruptible data stream that can progress with any desired throughput.

As can be seen from the above flow, the access patterns involved for our DRAM controller are ideal in the sense that any access always exhibits the following characteristics:

1. *Strictly increasing*. The data capture case simply starts writing from the lowest addressable DRAM word and increments by a fixed amount until it is stopped. The readout case behaves identical, but issues read instead of write commands.

2. *Contiguous and aligned*. Capture and readout always write and read entire DRAM words and addressing contiguous DRAM elements. This also implies that addresses are always aligned to DRAM rows and columns. The DRAM data bus is thus always fully and optimally saturated with meaningful data, and accesses never have to be split into two commands.

3. *No interruptions*. Once capture begins, no other command than a write is ever issued until the case is fully finished. For readout, the same holds for read commands. No delays are to be expected due to recovery times since write-after-read or read-after-write cases do not happen.

All of these characteristics ensure that all access patterns for our DRAM controller are fully predictable, making it possible to optimize the DRAM state machine to achieve maximum throughput. No worst-case assumptions have to be made regarding address word alignment and potentially time consuming commands such as row precharging or activation which would normally be the case if random access patterns would have to be supported, as we will see below.

4.2.3 *DRAM protocol*

This section continues with a basic explanation of the DRAM protocol as it is used for virtually all modern DRAM systems including DDR, DDR2 and DDR3 SDRAM. Since the protocol is quite complex and elaborate, we only focus on information relevant to the memory controller and previously described access patterns on a functional level.

Starting off, the specification dictates the use of the following DRAM signals, as seen from the DRAM devices, and described in [JED08] and [JNW07]:

- **CK**. Rising edge clock used to synchronize control and address transmissions (CMD and ADDR).
- **CK**, **CK#**. Dual-edged clock used to synchronize data bus transmissions (DQ, DQS and DQS#).
- CMD. Command input lines (also known as RAS#, CAS# and WE#) that define the command being issued.
- ADDR. Address selection lines, consisting of 16 address bits (A0 to A15) and 3 bank address bits (BA0 to BA2), also muxed as op-code inputs for certain commands.
- DQ. Data input/output bus that works in both directions.
- DM. Data mask that masks input data during a write request.
- **DQS**, **DQS#**. Strobe signals indicating valid input or output on the data bus (DQ).
- CS#. One or more chip selects that allow for external rank selection on systems with multiple ranks.

All of the above signals normally use single-ended signaling, except those denoted in **bold**, which use differential signaling. These signals are used to transmit requests to the DRAM devices in a pipelined or phased fashion as can be seen in Figure 17. The most relevant DRAM commands are described below.

ACTIVE (ROW ACTIVATION) The purpose of the ACTIVE command is to move data from cells in the DRAM into the sense amplifiers, and vice versa, as illustrated in Figure 16. A row in a particular bank is selected by means of the ADDR lines, after which the sense amplifiers in the relevant DRAM devices are set up. This row is then active (or open) for accesses until a PRECHARGE command is issued. The time for a row activation is referred to as t_{RCD}, also known as the RAS to CAS delay.

PRECHARGE (ROW DEACTIVATION) The PRECHARGE command is used to deactivate or close the open row in a particular bank or all banks, and is typically used after all relevant accesses in the currently open row have been done. Although this command is explicit, precharging can also be done automatically when setting up the appropriate flags for a READ or WRITE command, as will be explained below. The time for a row precharge is referred to as t_{RP}, also known as the RAS precharge delay. Note that whenever a write occurs before a precharge, the DRAM devices should be allowed time to recover from this last write command before the precharge command is issued. This time is fixed and known as t_{WR} or the write recovery time.

READ (COLUMN READ) When a READ command is issued, the DRAM device transmits the data from the column and bank as selected using the ADDR lines. Internally, the DRAM device uses an N-bit wide bus made up of sense amplifiers that each move data from individual columns. The outputs of the sense amplifiers are fed into a pipelined circuit consisting of read latches and a multiplexers. This circuit allows the DRAM device to serially output multiple bits from adjacent columns on each data bus line, effectively multiplying the bandwidth for each individual READ request. This mechanism is called *prefetching*, and is of importance when optimizing throughput during data access.

Newer generations of DDR SDRAM devices have seen an increase of number of prefetch bits: whereas DDR SDRAM only supported 2-bit prefetch, DDR2 SDRAM increased this value to 4-bit, while DDR3 SDRAM supports 8-bit prefetch. Note that the prefetch bits are synonymous to the burst length or the minimum number of data bits transmitted per READ or WRITE command, and thus put an upper bound on the respective data throughput.

Specifically, DDR3 SDRAM allows for a configurable burst length (prefetch bits) of either 4 ("Burst Chop 4", BC4) or 8 ("Burst Length 8") bits. The length can either be selected globally, e.g. during initialization, or specified *on-the-fly* by encoding a corresponding bit in the ADDR lines during a READ or WRITE command. Although a 8-bit burst access would maximize throughput, the choice for a 4-bit burst may be convenient for systems that do not require high throughput, as it lowers the amount of words and required logic per data access, as will be explained in subsequent sections.

WRITE (COLUMN WRITE) The WRITE command functions much like a READ command, where data is read and stored into the column and bank as selected using the ADDR lines. An example of two 8-bit write burst commands can be seen in Figure 17. In the case of an equivalent 4-bit write burst, timing would be identical to this example, although the highest 4 bits of each DQ sequence would simply not be used. We assume that the actual write always takes $t_{ROW} = N$ cycles, where N equals the amount of bits being written, e.g. 4 or 8, and is always delayed by DRAM parameter CWL, also known as the CAS write latency.

REFRESH DRAM devices store data as electrical charge inside volatile capacitive cells, and it is inevitable that that this charge will gradually leak out through the memory system. To ensure proper data integrity, all DRAM cells must be periodically restored to full charge in a process called refreshing. During a refresh, the DRAM device initiates an internal process in which a refresh row address register is incremented and this particular refresh row is read out in all banks, which

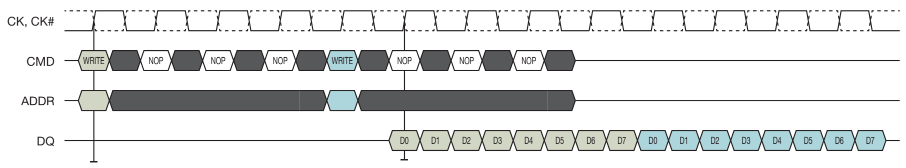


Figure 17: Simplified illustration of the pipelined DRAM command timing in case of two BL8 write requests. Green and blue indicate any commands, addresses and data bits belonging to the same request.

takes t_{RFC} (refresh command time). When the REFRESH command is issued at the average periodic interval t_{REFI} (refresh interval) as required by the DRAM specification, it is guaranteed that all rows are eventually refreshed. However, as the refreshing action obviously stalls the memory system at a periodic interval, there is a measurable impact on throughput as will also be explained in subsequent sections.

4.2.4 *Peak transfer rate*

Given Figure 17, it is easy to see how read and write requests are handled in a pipelined fashion. In ideal circumstances, newly issued commands would appear to the memory controller sequentially and without delay, and all commands in the pipeline would be handled according to a completely fixed schedule as illustrated in the figure and without any interruptions. Commands would be issued once every 4 clocks and data would be issued on every falling and rising edge of the clock or 0.5 clocks. Assuming a standard DDR3 DRAM bus width of 64 bits, this would effectively read or write 512 bits for every issued read or write command. In reality, the pipeline is often stalled by events such as DRAM refreshes and row precharging, as explained in this chapter. Nevertheless, the ideal scenario is often used to calculate the theoretical *peak transfer rate* in terms of MT/s or megatransfers per second, e.g. such as for the DDR3-600 configuration which we will be using later on in our research:

$$\begin{aligned}\text{DRAM clock} \times 2 &= 300\,\text{MHz} \times 2 \\ &= 600\,\text{MT/s}\end{aligned}$$

The peak transfer rate can also be expressed in terms of more conventional gigabits or -bytes per second units when taking the DRAM width into account:

$$\begin{aligned}\text{peak transfer rate} \times \text{DRAM width} &= 600\,\text{MT/s} \times 64\,\text{bits} \\ &= 37.5\,\text{Gbit/s} \qquad (2) \\ &= 4.6875\,\text{GiB/s}\end{aligned}$$

In practice, these peak transfer rates are far from the actual observed throughput of a conventional DRAM controller due to a number of reasons.

First of all, random access means that access may not be *strictly increasing*, *contiguous* and/or *aligned*. This implies that bytes that are to be read or written may have to be spread across multiple DRAM rows or columns, and multiple commands may in fact be required to write the entire data to DRAM. This also means that the DRAM data bus will often only contain a portion of useful data and a bit mask has to be used to mask out the useless bits of data, effectively wasting throughput.

Furthermore, DRAM protocol dictates the use of *recovery* timing constraints in case of write-after-read, read-after-write or other cases where continuous burst is not possible. A burst is always preferred, but can only last as long as identical commands follow each other, e.g. in the case of many subsequent WRITE commands. For random access, no guarantees can be made about the occurrence of reads and writes, so it must be assumed that any read may follow a write or vice versa, and burst behaviour is less than optimal.

As explained in the previous section, our access patterns are in fact not random at all, allowing for a specialized implementation of the DRAM state machine.

4.2.5 *Protocol state machine*

The original JEDEC state machine for the DRAM protocol can be seen in Figure 18. This state machine covers the general case of random memory access behaviour with a clear order of commands when accessing data: a bank is activated, data is read or written, the bank is precharged. This kind of state machine expects that the memory controller is issued with distinct read or write commands, as is the case for convential random memory access patterns.

For our memory controller, there is a specific focus on maximizing throughput for sequential burst writes. In our case, the original state machine has been redesigned, as can be seen in Figure 19. Unlike the JEDEC state machine, in which the idle state always assumes that all banks are closed or precharged, this state machine contains a number of burst-specific states in which the idle state always assumes an already open and active bank. In other words, an incoming write command can almost always be handled immediately, without having to activate the corresponding bank, unless the write occurs in a different (non opened) bank. Obviously, this only holds when burst writes occur strictly sequentially, as is the case in our system.

Note that sequential burst writes are also possible with the JEDEC state machine in Figure 18, as long as the state machine is in the *Writing* state. However, in the case of continuous write burst (e.g. a

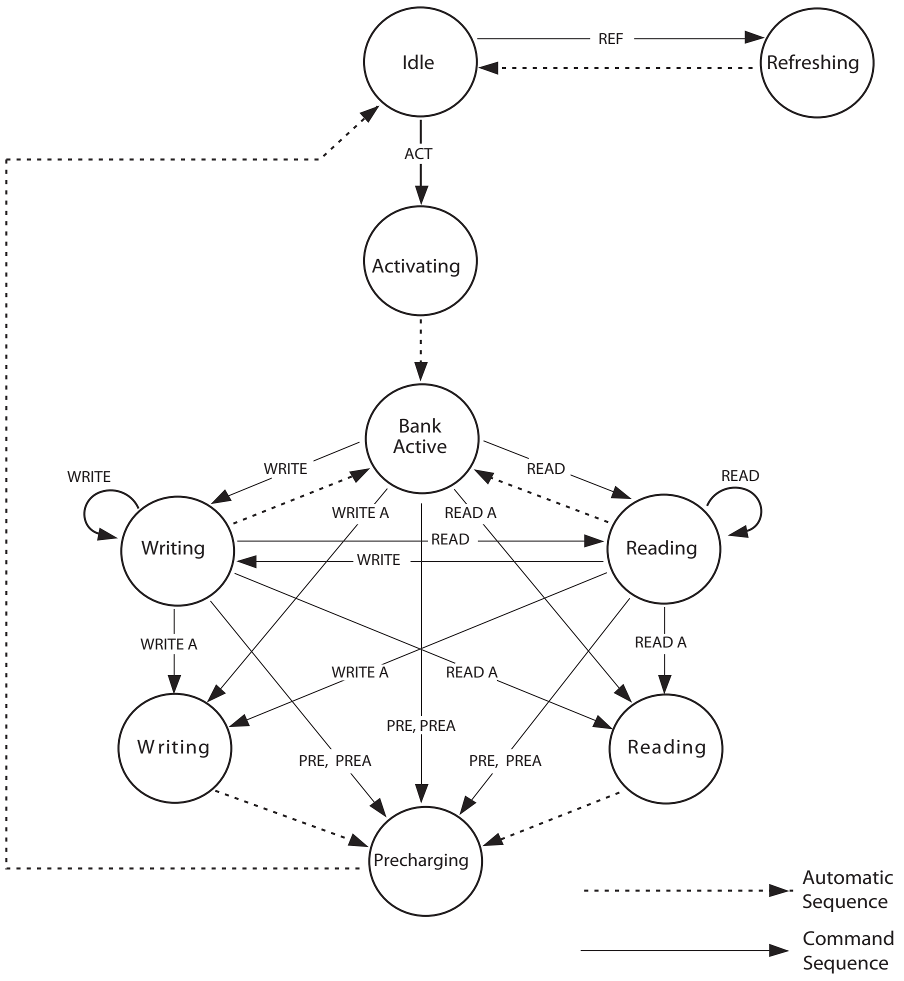


Figure 18: Original simplified DRAM protocol state diagram as in [JED08]. Power and calibration states have been omitted.

long queue of burst writes), the state machine can possibly spend so much time in the *Writing* state that it would violate the refresh interval timing of the DRAM, leading to unpredictable results. The JEDEC state machine therefore only serves as a simplified guideline, on which the actual state machine design in Figure 19 is based.

In our state machine, the following states and associated commands can be seen:

- **Idle**. Idle state in which all banks are precharged. Commands READ and WRITE indicate non-burst reads and writes.
- **Idle B**. Idle state in burst mode, in which a bank is assumed to be active. Commands READ and WRITE indicate non-burst reads and writes, while BURST A indicates a burst write on the currently active bank and BURST B indicates a burst write outside the currently active bank.
- **Activating R**, **Activating W**. Activating states in case of a non-burst read or write. This flow implies an exit of burst mode.

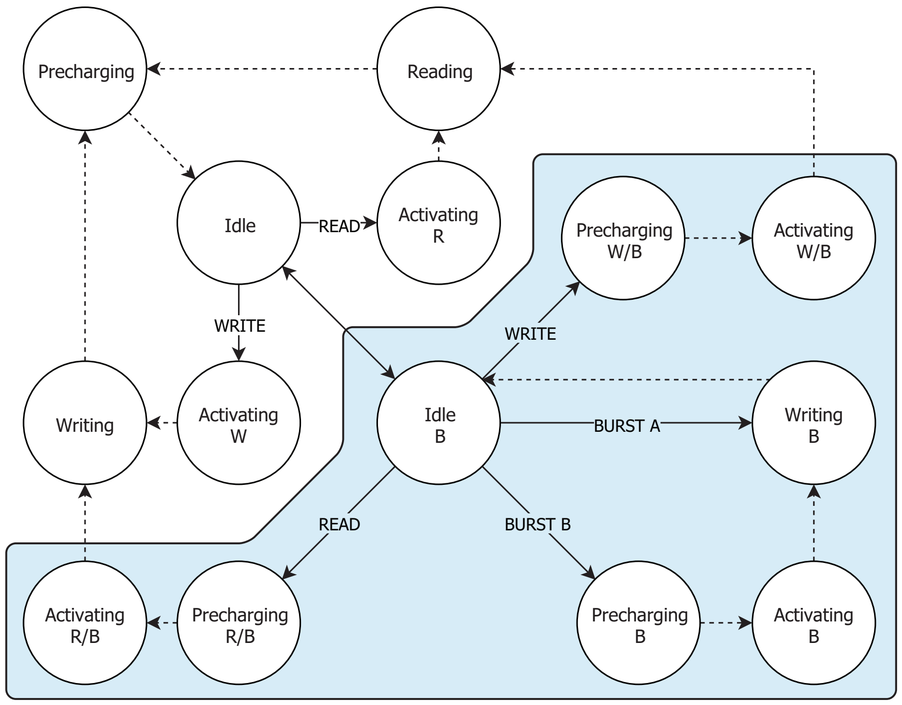


Figure 19: Redesigned DRAM state diagram optimized for sequential burst writes. Burst-related states are highlighted in blue.

- **Reading**, **Writing**. Reading and writing state in case of a non-burst read or write. This flow implies an exit of burst mode.
- **Precharge**. Precharge state after a non-burst read or write using the auto precharge feature.
- **Precharging W/B**, **Activating W/B**. Precharging and activating states in case of a non-burst write (write-after-burst).
- **Precharging R/B**, **Activating R/B**. Precharging and activating states in case of a non-burst read (read-after-burst).
- **Precharging B**, **Activating B**. Precharging and activating states in case of a burst write.
- **Writing B**. Writing state in case of a burst write. Returns to the idle state immediately after completion.

Note that in the above states, auto precharge is always applied in case of non-burst reads or writes. As such, a single non-burst read or write command is always preceded by a bank activation and followed by a bank precharge. Although this is not ideal in terms of non-burst access performance, it does simplify the required DRAM state machine logic. Rather than optimizing the performance for non-burst access in our system, we have chosen to focus primarily on maximizing throughput for burst access, as has been stated before.

4.2.6 *FPGA implementation*

The actual FPGA implementation of the DRAM controller is composed of a number of elements:

- *DRAM physical layer I/O buffers*. All buffer logic elements concerned with interfacing to the DRAM DIMM physical layer. This includes:
  - *64 DQ I/O buffers*. The DQ bus consists of 64 bidirectional single-ended wires and requires matching I/O buffers to interface with these, e.g. 64 IOBUFs for Xilinx platforms.
  - *8 differential DQS I/O buffers*. There are 8 bidirectional differential DQS strobe signals. These are to be interfaced with matching I/O buffers, e.g. 16 IOBUFs or ISERDES2 combined with 16 OSERDES2 for Xilinx platforms. Since all DRAM timings in our implementation are known, the input strobe signals can however be safely ignored.
  - *8 DM I/O buffers*. There are 8 bidirectional DM single-ended data mask signals, interfaced with matching I/O buffers, e.g. 8 IOBUFs for I/O buffers. However, since our implementation does not make use of data masking since the entire width of the data bus always used, DM input is ignored, and output is fixed to all-ones.
  - *16 ADDR, 3 BA and 2 CS# output buffers*. These 20 single-ended outputs make up the complete DRAM address bus including DIMM rank selection, and are driven by matching output buffers, e.g. 20 OBUFs for Xilinx platforms.
  - *3 CMD and 2 DCE output buffers*. These 5 single-ended outputs make up the DRAM command bus, and are driven by matching output buffers, e.g. 5 OBUFs for Xilinx platforms.
  - *2 CK differential output buffers*. The complete DIMM clock net is driven by two matching differential output buffers, e.g. 2 OSERDES2 for Xilinx platforms.
- *DRAM protocol FSM*. This FSM implements the state machine logic as previously outlined in this section. The FSM's only stimuli for transitions are the signals from the *command interface* when it is in either of the idle states. It is otherwise fully self-timed, using the fixed timings as specified in the DRAM protocol, also illustrated by the dotted lines in Figure 19.
- *Command interface*. This element provides the necessary logic between the DRAM controller component I/O signals and the idle state transitions in the FSM. As such, the DRAM controller exposes the following signals to the rest of the system:

- *20 address input bus signals* mirroring the DRAM address bus.
- *512 data bus I/O signals* mirroring the maximum DRAM data throughput per write command, when considering a burst size of 8 and 64 bits per clock.
- *2 read/write strobe input signals* indicating that the data and address bus bits are valid, and a read or write command should be issued.
- *1 ready strobe output signal* indicating that any data on the data bus is valid, e.g. after a read command, and the controller is ready to issue new commands.
- *1 clock input signal* tied to the system's primary clock domain that is multiplied to the DRAM command clock. Needless to say, this requires the system's primary clock to be an integer division of the DRAM command clock. For the DDR3-600 case outlined in this thesis, a clock of 150 MHz is to be used and is quadrupled to achieve a 600 Mhz data clock.
- *1 reset input signal* tied to the reset inputs of all elements inside the DRAM controller.

The above elements allow for a FPGA implementation of the specialized streaming DRAM controller as introduced in this section, which is then connected to the arbitration subsystem described below.

4.2.7 *Command arbitration*

The hardware domain diagram in Figure 9 of Chapter 3 shows that the DRAM controller actually interfaces with two different subsystems: the stream interleaver, and the embedded control processor. This requires the use of an arbitration component, also referred to as an *arbiter* that allows these two subsystems to exclusively issue commands to the DRAM controller.

Our implementation of this arbiter is simply a multiplexer that multiplexes the signals from these two subsystems into a single set of signals for the DRAM controller. First, the DRAM command interface signals outlined in the previous section are mirrored for each of the subsystems. A simple multiplexer logic component is then added to multiplex the input DRAM command interface signals from either subsystem into output DRAM command interface signals, based on a selector.

In order to select one of the two subsystems, we first assume that the two subsystems exhibit exclusive behaviour and never access the memory concurrently: the stream interleaver simply starts writing captured image sensor data, ultimately finishes and is then followed by the embedded control processor that starts a readout of the DRAM that finally finishes when all its contents are read. Due to this exclusive access behaviour, it is easy to model the multiplex selector behaviour in our arbiter:

1. If the stream interleaver issues a write (write strobe is high), select the DRAM command interface signals from the stream interleaver.
2. Otherwise, if the embedded control processor issues a read or write (read strobe OR write strobe is high), select the DRAM command interface signals from the embedded control processor.

Note that the above logic is very naive, but guarantees correct functioning of the arbiter as long as it can be guaranteed that the stream interleaver and embedded control processor never issue any commands at the same time. As we will see in Chapter 5, we can easily enforce this by letting the embedded control processor control the behaviour of the stream interleaver.

4.3 STREAM INTERLEAVER

Our system is designed to interface with multiple sensors in order to facilitate high-speed capture. It follows that for each of these sensors, we require a sensor receiver interface as described in Section 4.1. While this sensor receiver interface covers the input for every sensor in our system, we actually need a block of logic to combine the raw data from all of these sensors into a single stream that can be connected to the DRAM controller described in the previous section. As such, all raw data coming in from the sensor receiver interfaces needs to be combined into a stream of DRAM write commands by means of appropriate address bus, data bus and write strobe signals. This block of logic is also known as the *interleaver*, and its exact working is dictated by the data bus width of the memory controller $N_{memory\_data\_width}$ as well as the total number of sensors N and lanes N_{lanes} per sensor, such that:

$$N_{lines} = \frac{N_{memory\_data\_width}}{N \times N_{lanes} \times S} \tag{3}$$

Where S represents the deserializer ratio or data bus width of the sensor receiver interface. In our case, we assume a convenient $S = 8$ for our entire system to allow for effortless alignment and compatibility with the memory controller. Note that all sensors connected to the interleaver are expected to have identical configurations in terms of S, N_{lanes}, f_{SERIAL} and so forth.

In Equation 3, $N \times N_{lanes} \times S$ represents the amount of total bits coming in from all sensor receiver interface outputs per system clock cycle if data is available, which we will refer to as a *line*, otherwise defined as:

$$N_{line\_width} = N \times N_{lanes} \times S \tag{4}$$

The outputs of the receiver interfaces or lines are fed into a shift register with a capacity of $N_{memory\_data\_width}$ and as soon as this shift register is saturated, a write request with its contents is dispatched towards the memory controller.

In any case, N_{lines} represents the amount of lines that are required before the shift register, or the DRAM data bus, is saturated. We thus assume that $N_{lines} >= 1$. If this is not the case, multiple write requests are instead necessary in order to store a single line of data, which would require a different interleaving implementation and is beyond the scope of this thesis.

If $N_{lines} \notin \mathbb{N}^+$, feeding a new complete line into the shift register will eventually result in shifting out part of the oldest line. To avoid loss of data in this case, data is shifted in bit-by-bit until the register is saturated, in which case the newest line is broken up and the remainder of bits is only shifted in after the write request has been made, after which the entire course of action is repeated. In the end, the shift register logic ensures that incoming data is translated into write requests for the memory controller, such that its data bus is always optimally saturated with data to allow maximum sustained throughput.

The rate at which lines will flow into the interleaver is fully determined by the serial bit rate f_{SERIAL} of the sensors. We define this rate simply as follows:

$$f_{LINE} = \frac{f_{SERIAL}}{S} \tag{5}$$

Here, f_{LINE} is the rate at which N_{lanes} data words of size S appear at the output of each sensor receiver interface, to be consumed by the interleaver, making up a single line of N_{line_width} bits. For simplicity, we assume that all sensor receiver interfaces in the system present this data at the same time on their respective outputs, despite the fact that in practice sensors may exhibit slight relative phase differences in any associated clocks. All sensor data is therefore also assumed to be consumed at the same time by the interleaver.

By combining the aforementioned equations, a relation can be established between the line rate f_{LINE} and the number of lines required to saturate the DRAM data bus N_{lines}, allowing one to calculate the rate at which the interleaver will issue DRAM data write requests:

$$f_{WRITE} = \frac{f_{LINE}}{N_{lines}} = \frac{f_{SERIAL} \times N \times N_{lanes}}{N_{memory\_data\_width}} \tag{6}$$

It is important to note that in case lines will not perfectly align in the DRAM data bus, e.g. N_{lines} is a fraction, f_{WRITE} only represents the *average* rate at which DRAM write requests are issued. The worst-case rate, or the highest possible write rate, is then calculated by explicitly substituting with $\lfloor N_{lines} \rfloor$. For all other cases, the worst-case rate is equal to the average rate. The worst-case rate is therefore defined as follows:

$$f_{WRITE_{wc}} = \begin{cases} \frac{f_{LINE}}{\lfloor N_{lines} \rfloor} & \text{if } N_{lines} > 1 \text{ and } N_{lines} \notin \mathbb{N}^+, \\ & \text{e.g. } N_{lines} \text{ is a fraction larger than 1} \\ \frac{f_{LINE}}{N_{lines}} & \text{otherwise} \end{cases} \tag{7}$$

The worst-case interleaver rate $f_{WRITE_{wc}}$ is be one of the primary factors when analyzing system throughput. As we will later describe in Chapter 6, this is due to the fact that our system has hard real-time requirements, making the worst case behaviour the decisive factor in guaranteeable bandwidth as also stated in [Goo+16].

4.3.1 *FPGA implementation*

The VHDL or Verilog implementation of the interleaver is not burdened by the use of vendor-specific primitives as it is a purely internal FPGA component that only interfaces with other components. This section therefore attempts to explain the logic involved in the stream interleaver implementation in a generic way.

As mentioned before, it is assumed that $N_{lines} >= 1$ such that multiple lines are necessary in order to saturate the DRAM data bus. Our implementation then exhibits the following functionality:

- A *shift register* of width $N_{memory\_data\_width}$ is used to shift in bits of data from the new lines as they become available. A *shift counter* is used to record the number of shifts.
- As soon as the oldest valid bit in the shift register reaches the final bit of the shift register, e.g. in case the counter will equal $N_{memory\_data\_width}$:
  - The shift counter is reset to 0.
  - A *DRAM write command* is issued by signaling the write strobe signal, copying the current shift register contents to the data bus, and setting the address bus to the value of an *address counter* that is incremented after each write.

  - Any remaining (yet to be shifted) data in the new lines are shifted into the shift register.

- As soon as the address counter reaches a predefined address, e.g. the end of the available DRAM space, all input is ignored from thereon and a *done* signal is set.

The above implementation ensures that new lines of raw image data are stored *strictly increasing*, *contiguous* and *aligned* according to the characteristics put forth in Section 4.2.2.

The stream interleaver also exposes a *reset* signal that resets the internal logic elements, and an *enable* signal that enables the functionality of the stream interleaver. The interleaver is connected to the S-bit wide FIFO output bus of each sensor receiver interface. The enable signal of the interleaver is directly routed to the read enable signal of the sensor receiver interface's FIFO, and starts up the flow of raw video data in the system's primary clock domain as soon as the interleaver is enabled.

4.4 EMBEDDED CONTROL PROCESSOR

In our system, the presence of an embedded processor is important as it facilitates a number of critical control functions that ensure that the system operates correctly:

- DRAM controller configuration, notably self-testing.

- Enabling and disabling of stream interleaver and readout interface.

- Sensor control interface drivers for sensor configuration and phased start functionality.

While all of the above functionality could have been implemented purely in hardware, the choice to move these functions over to a software solution, also known as the *embedded domain*, makes it possible to dynamically reconfigure various parameters of the system. Furthermore, a software-based solution eases development and debugging of the system, as the embedded control processor is also capable of providing auxiliary debugging and output facilities. The exact implementation of the above control functionality is in fact part of the software domain, as we have explained before, and is thus further described in Chapter 5.

To accommodate a software implementation, the system requires an embedded processor with a straightforward means of connecting to the other subsystems. Furthermore, as the memory controller addresses up to several gibibytes of DRAM, the processor should be capable of conventional 32-bit memory addressing. For the sake of software development, the availability of a well-supported C/C++ toolchain with a proper means of debugging is also a sensible requirement. The presence of a memory management unit is however not necessary, and by avoiding the use of a conventional operating system, the processor can be kept fairly small and lightweight. The choice for a soft-core processor therefore fits well in our system.

A variety of commonly available lightweight soft-core processors exist today, each of which may be more or less of an obvious choice depending on the actual platform being used to realize the system. Given that the functionality in the embedded domain can be easily developed on most of these processors, the specific choice of which soft-core IP to use is to be be based on practical considerations as well as real-time performance. The embedded processor is expected to operate in real-time, issuing instructions related to system control with minimal delay, especially those related to timing-sensitive operations, also described beyond this section.

For Xilinx-based platforms, the use of the proprietary *MicroBlaze* IP seems most straightforward, whereas one would likely prefer the proprietary *Nios II* for Altera-based platforms[2]. In addition, there are a number of open-source and freely available projects that provide an alternative to the well-known proprietary soft-core processors, such as the *OpenRISC*, *LEON3* and the MicroBlaze-compatible *MB-Lite* as described in [Dor+09] and [KL10]. The open-source nature of these projects also allow fine tuned customization in case more functionality is needed.

All of the above soft-core processors are suitable for real-time operation as described in [TAK06], and thus for our system. Though, [Axe+14] explains that the use of caches and pipelines, as implemented on in all of these processors, impose instruction times that are inherently not fixed and may vary depending on execution history. One might therefore consider a processor with a minimal or no amount of pipeline stages as a quick indicator for a consistent latency across instructions, suited for real-time operation. As summarized in [TAK06], the OpenRISC, LEON3, MB-Lite, Nios II and Microblaze respectively have 5, 7, 5, 6 and 3 stages each, making the Microblaze a seemingly good choice in this case. Further instruction latency consistency optimization is achieved by disabling instruction and data caches for more consistent instruction timing, where possible.

4.5 SENSOR CONTROL INTERFACE

This section describes the sensor control interface, which implements a I^{2}C-compatible driver that interfaces with all the sensors in the system by using a *multiplexer*. It also gives an overview of the I^{2}C proto-

2 For more information on the details of the MicroBlaze and Nios II soft-core processors, refer to the respective manufacturer's references guides [Xil08] and [Alt15b].

col itself, deemed necessary in order to understand the timing aspects behind the *phased starting* of sensors.

As the MIPI CSI-2 defines a separate bidirectional I^{2}C-compatible control interface, we can safely assume that the CSI-2 sensors in the market typically have indeed such an interface [MIP16a]. This also holds for sensors with similar CSI-2 interfaces such as HiSPi as can be seen in [Sem15a] and [Sem15b]. In this thesis, we therefore define the presence of a I^{2}C-compatible control interface for all sensors that are attached to the system.

4.5.1 *I^{2}C protocol overview*

The I^{2}C protocol is a quite straightforward: the physical bus consists of two bidirectional active wires, notably SDA (Serial Data Line) and SCL (Serial Clock Line), connected in an *open drain* topology using pull-up resistors. With this open drain design, the bus drivers in both the master and slave devices can only ever pull the signals to a logic low state instead of driving it to logic high, avoiding potentially damaging scenarios in which multiple devices are driving opposite logic levels [Lee09]. Any number of slaves or masters can connect to this bus, as long as they do not interfere with each other.

A typical data write action, as initiated by the master device, can be seen in Figure 20. This transmission starts off with a START condition, followed by 7 address bits that describes a specific slave device on the bus responding to this address. An extra R/W (Read/Write) bit is used to indicate whether the master is requesting a read or write from the slave. In any transmission, words are 8-bit long and every 8-bit word sent from the master to the slave is always followed by an acknowledgement or ACK from slave to master. After the initial addressing data, the transmission is followed by an arbitrary amount of data words and is finalized by means of a STOP condition. For a data write action, the R/W bit is low, and data words are sent from master to slave. In case of a data read action, R/W is high and data words are sent from slave to master instead.

4.5.2 *Camera control interface (CCI)*

The sensor control interface is also known as the *Camera Control Interface (CCI)*, and generally consists of a half duplex, bi-directional, two-wire serial bus that is largely compatible with the I^{2}C bus specification as originally documented in [Phi03].

As the original I^{2}C standard is quite elaborate, certain non-critical features deemed unnecessary for sensor control are usually left out of the actual CCI implementations. Support for multiple master devices, legacy transfer rates and clock stretching is often omitted in favor of a

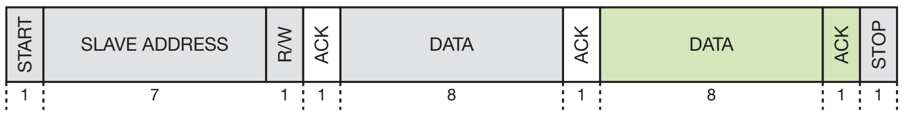


Figure 20: I²C data transmission using 7-bit slave addressing with two data words as described in [Phi03]. Grey blocks are transmissions from master to slave, white are from slave to master and green depends on the type of operation. Bit widths are denoted below.

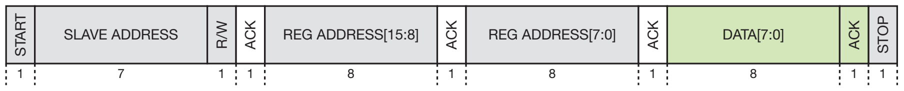


Figure 21: Example CCI data transmission addressing an 8-bit register using a 16-bit address. Grey blocks are transmissions from master to slave, white are from slave to master and green depends on the type of operation. Bit widths are denoted below.

more straightforward single master multiple slave topology running in I²C's *Fast-mode* with a maximum transfer rate of 250 KHz.

Sensors are positioned on the bus as simple slave devices, controllable by a single master or host that sends out commands using conventional 7-bit slave addressing. The actual command protocol is purposely left open to allow manufacturers to implement their own set of commands most relevant to the sensor.

The CCI specification extends the I²C data protocol with support for registers that store either 8, 16, 32 and 64-bit wide data words, and are accessed using 16-bit addresses that define a 64 KiB register space. An example of a CCI data operation addressing an 8-bit register can be seen in Figure 21, and this operation differs only in the amount of transmitted data words for the other register widths.

The register space allows for manufacturer specific sensor configuration and control, and typically includes all critical timing parameters such as exposure, PLL configuration and row timing, as well as sensor resolution, test patterns, stream control and many other options as can be seen in [Sem15a][Sem15b].

4.5.3 *FPGA implementation*

The sensor control interface subsystem consists of a single I²C-compatible master driver that supports the aforementioned CCI. Because the interface transmits at a rate of 250 kHz in its own clock domain, logic within the interface has to properly facilitate the crossing of clock domains.

However, since the interface drives the SCL clocking signal as a master, and is thus not dependent on any input clocks, the clock domain can be generated by means of an arithmetic counter that divides the system clock down to the appropriate output clock frequency. Using these counters, the SCL clock domain is internally kept in phase with the system clock and effectively does not require any of the metastability mitigation described in Section 4.7.

To accommodate the command and control of an arbitrary amount of sensors, the master driver logic is connected to a 1-to-N multiplexer where N is the number of sensors in the system. The multiplexer as well as the master driver are fully controllable from the embedded domain, such that complex sensor control sequences can be executed from the embedded processor. More specifically, the sensor control interface exposes a number of signals specifically designed for easy embedded control:

- $\log_2(N)$ *selector signals* controlling the multiplexer selector functionality, where N is the total number of connected image sensors. The selector signals uses a simple two's complement integer representation.
- *1 read/write signal* indicating whether the interface should issue a read or write command.
- *7 address input bus signals* corresponding to the address part of a I^2C command. This address is internally encoded and subsequently transmitted on the SDA output signal.
- *8 data input bus signals* containing the 8-bit data in case a write command is issued.
- *8 data output bus signals* containing the 8-bit data in case a read command is issued.
- *1 ready signal* indicating whether the data bus contains valid data and/or a new command is ready to be issued.
- *1 clock input signal* tied to the system's primary clock domain or embedded clock domain.
- *1 reset input signal* that resets all internal FPGA logic.

4.5.4 *Phased start*

This complex control from the embedded domain is especially necessary, if we consider the requirement of high-speed video capture for our system. In the case of high-speed capture, the system is connected to multiple sensors, and these sensors will be exposing and integrating their pixel arrays at specific moments in time. This way, different sensors capture images in different but adjoining regions of time, forming a continuous and temporally linear stream of images when ultimately combined.

A visual illustration of this timing regime can be seen in Figure 22. In this figure, each sensor is limited by a minimum readout duration

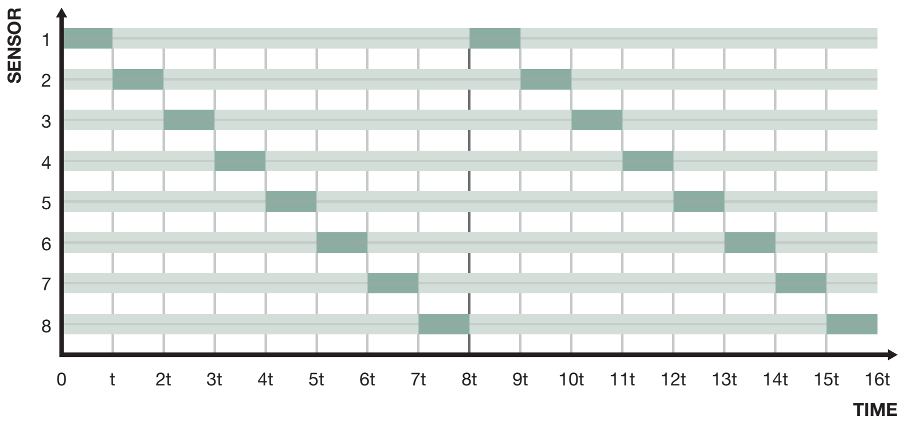


Figure 22: Phased start with 8 sensors plotted against time. Dark regions represent exposure duration, and light regions represent readout duration for all sensors.

which is enforced by sensor limitations such as bandwidth. However, the exposure duration is not limited by bandwidth, and can thus be set at a considerably shorter duration. By using multiple sensors and temporally adjoining their individual exposure durations, and ultimately combining the images captured at those specific times in the same order, video capture at a far higher rate than that imposed by a single sensor's readout duration can be achieved.

TIMING CRITERIA From the figure, it follows that for every sensor i, the exposure start time $t^i_{exposure\_start}$ must be configured to coincide with the end time $t^j_{exposure\_end}$ of another sensor j, where $t_{exposure\_start}$ is configured by means of the sensor control interface, and $t_{exposure\_end}$ is generally defined as follows:

$$t_{exposure\_end} = t_{exposure\_start} + t_{exposure}$$

In an ideal system, the difference between the start and end times is ideally 0 or otherwise minimized:

$$|t^i_{exposure\_start} - t^j_{exposure\_end}| = \epsilon \to 0$$

If N sensors are available in the system, i and j represent two unique sensors in such a way that the above equation holds for each sensor $i, j = [0, N)$ where $i \neq j$.

Assuming a fixed f_{fps} for each sensor i such that $f_{fps} = f^i_{fps}$, and if the images of all sensors are ultimately combined, the effective capture rate of the system can be defined as:

$$f_{fps\_system} = N * f_{fps}$$

As an example in the case of Figure 22, where $N = 8$ and $f_{fps} = 60\,\text{Hz}$, the effective system capture rate is then simply:

$$f_{fps\_system} = 9 * 60\,\text{Hz} = 540\,\text{Hz}$$

Recall that the research introduction in Chapter 1 introduced two evaluation criteria, one of which stated that the timing accuracy of the phased start must be at least 99%. The timing accuracy here refers to the exact time period between two subsequent phased start periods, e.g. a single unit of time t as illustrated in Figure 22. The use of one or more clocks in the implementation of the sensor control interface implies that t is in fact discrete and may therefore exhibit an error ϵ in reference to the ideal period of t.

Recall that there are two clocks involved in the sensor control interface: a *system clock* used to transfer command data to and from the subsystem, and a *I^{2}C clock* used to transfer command data to and from a sensor. These two clocks have different timing effects on the commands as they are sent to the sensor.

The system clock issues the actual commands and therefore has a direct effect on the exact moment in time the sensor sees a new command: a new issue of a command can be delayed by a system clock period, causing an error ϵ equal to the system clock period. However, since the system clock typically operates in the tens or hundreds of MHz, and the effect of ϵ only becomes significant in the order of magnitude of f_{fps_system}, e.g. several kHz, the effect is therefore negligible.

The *I^{2}C clock*, e.g. at 250 kHz, does have a significant effect on the command timing. Since this clock is in fact generated inside the sensor control interface, we assume that this clock ideally starts at the moment a new command is issued by the system. Needless to say, the clock has an effect on the total time it takes for the interface to send the command as it defines the rate at which bits can be sent. However, since all start commands issued to sensors have an equal bit length, the command duration is fixed and equal for every sensor. This clock therefore does not have any effect on ϵ and its effect can also be disregarded.

What is left is to ensure that the system actually issues the start commands with minimal delay. The start commands are handled by the sensor control driver running on the embedded processor, further detailed in Chapter 5. The embedded processor must be (near) real-time, ideally executing the task's necessary instructions according to a fixed schedule that is known a priori.

4.6 READOUT INTERFACE

On the highest level, the readout interface serves as a generic channel between the hardware domain (or DRAM contents) and the software domain (host), respectively containing the raw captured video data and the corresponding components to process this data.

Our readout interface is designed in a way that connects its input to the DRAM controller while its output is physically connected to the software domain host by means of a data bus supported by both the FPGA platform and the host. The implementation of the readout interface then contains the signaling IP to physically interface with the given readout data bus, while control is provided by the embedded control processor.

The choice of data bus is arbitrary as it largely depends on the capabilities of the software domain host. In this thesis, we have chosen to use the *Serial Peripheral Interface* (SPI) bus due to its simplicity and wide support among host platforms, though its bandwidth is limited. By using the SPI bus, the readout interface is simply exposed as four single-ended wires on the FPGA platform carrying our hardware domain. These wires are then be connected to any SPI capable host, realizing the channel between the hardware and software domains, and allowing the software domain to download and further process the raw video data.

4.6.1 *FPGA implementation*

As stated, the input is provided by the DRAM controller. The input is in fact a data bus of $N_{memory\_data\_width}$ bits wide, e.g. 512-bit, that connects directly to the DRAM controller's data bus. Facilitating this input is a matching 20-bit address bus that connects to the DRAM address bus and specifies the address to read from DRAM, as well as a read strobe signal that issues the actual DRAM read command and the data ready signal from the DRAM controller.

As soon as the readout interface is enabled, a linearly increasing address is put on the address bus and the read signal is strobed. It then waits for data to appear on the input DRAM data bus and subsequently forwards all data bits to the readout data bus IP, i.e. the SPI slave controller, such that this data is then physically transmitted over the readout data bus. As soon as the bus IP indicates that all $N_{memory\_data\_width}$ bits have been transmitted, the entire flow is repeated until a predefined address such the end of the DRAM region has been reached.

The readout interface further provides an enable, a reset and a done signal that serve obvious purposes and are connected to the embedded control processor. Control of the readout interface in terms of enabling, disabling and resetting the FPGA logic is therefore exercised from the embedded control software, completing the readout interface functionality.

4.7 CLOCK DOMAIN CROSSING

It is clear that the datapaths and interfaces within the system span over multiple individual hardware systems each of which have individual clocks, i.e. the sensor, FPGA, DRAM and readout interface. Because of this, the datapaths cannot be synchronized by only a single clock, as would be the case in an ideally synchronized hardware design. This raises the issue of multiple clock domains and corresponding clock domain crossing, which is to be addressed properly to ensure stable signal integrity within the overall system. The use of on-chip clock domain crossing memories is common in embedded systems [Goo+05], and our platform is no exception. Here, two assumptions are made to further clarify the definition of a clock domain: any two clock domains contain clocks that are unrelated for which a clock phase relationship can not be guaranteed, and by extension, any clock domain may contain multiple clocks at different frequencies as long as all of these clocks have a guaranteed phase relationship.

PRIMARY CLOCK DOMAINS In our system, a minimal number of important clock domains can be identified that operate at a highest frequency range, i.e. in the 10 to 1000 MHz range. In our system, we call these the *primary* clock domains:

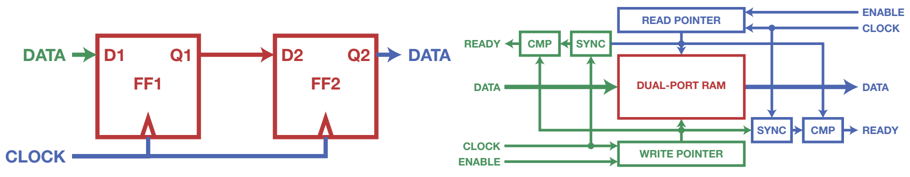
- Sensor receiver interface (e.g. 700 MHz).
- System logic: sensor controller, interconnect, memory controller and embedded processor (e.g. 150 MHz).
- Readout interface (e.g. 16 MHz).

SECONDARY CLOCK DOMAINS The *secondary* clock domains represent those that operate at slower speeds, i.e. below 1 MHz, or do not require any strict form of data communication:

- Sensor controller and I^{2}C interface (e.g. 250 KHz).
- Sensor clock and reset logic (e.g. 27 MHz).

The distinction between primary and secondary clock domains is important as it corresponds with the complexity of the circuitry required to safely communicate data across these clock domains boundaries.

As described in [Gin11], the issue of clock domain crossing lies in the metastability events that appear in digital circuits whenever



(a) Two flip-flop synchronizer. (b) Two-clock FIFO synchronizer.

Figure 23: The two different types of synchronizers used in our system. Green and blue represent the two different clock domains, and red acts as stabilizing logic in between.

asynchronous signals (or unrelated clocks) are used, as is the case in our system. Transferring data across these clock domain boundaries requires the use of *synchronizers* that effectively eradicate any metastability issues.

The two most important implementations for synchronizers in the context of our system are the *two flip-flop synchronizer* and *two-clock FIFO synchronizer*, as can be seen in Figure 23. Both of these are simple and straightforward to use:

- *Flip-flop sychronizer*. This design allows for single bits of data to pass through a clock domain boundary by simply chaining two flip-flops.
- *Two-clock FIFO synchronizer*. This design uses a dual-port RAM to pass entire words of data. Complex to implement, but generally available in vendor-specific HDL libraries such as those from Altera and Xilinx.

Both of the aforementioned synchronizers are necessary. The two-clock FIFO synchronizer is used whenever there are multiple words of data that need to be synchronized between two different clock domains at high throughput, as is the case for the sensor receiver interface in which words of deserialized sensor image data are to cross from the sensor clock domain into the system logic clock domain. The flip-flop synchronizer is used in all other cases whenever there are single bits or signals that need to pass between two clock domains, e.g. used for the sensor clock and reset logic, for which signals are generated by the system logic, but must be clocked in the sensor's clock domain. Note that while it is possible to use flip-flop synchronizers for larger data words, this typically makes the implementation unnecessary complex as FIFOs are commonly available in synthesis tools and better suited in this case.

5 SOFTWARE DOMAIN IMPLEMENTATION

This chapter continues with the description of the system as it was introduced in Chapter 3, focusing on the software domain and all its subsystems, illustrated before in Figure 10. Most of these subsystems are in fact custom software programs, implemented either on the embedded control processor in the hardware domain, or an external host, and providing the off-line functionality to transform the raw video data produced by the hardware domain. The result of this transformation is a video file or stream containing the high-speed images.

The stream decoder, image rectification and camera calibration as described in this chapter are designed to be part of a video streaming pipeline using *GStreamer*. The advantage of this approach is that each of the transformation steps can simply be chained together, starting with an input connected to the readout interface connected to the hardware domain, and ending with an output that simply writes a video file or stream. The entire streaming pipeline software is designed to be executed on an external host that is interfaced with the hardware domain, e.g. a conventional Linux-based computer.

The embedded control software runs on the embedded control processor in the hardware domain instead, and is thus not part of this video streaming pipeline. The control software is a separate subsystem that contains the necessary functionality to set up and control the video dataflow in the hardware domain, and to provide auxiliary functionality such as a DRAM self-test program during startup of the hardware.

5.1 EMBEDDED CONTROL

The software domain covers a small part of the hardware domain, notably that of the embedded control processor. While the hardware domain is concerned with the implementation of the processor, the actual program code or software running on the processor naturally falls within the software domain. As stated in the previous chapter, the presence of the embedded control processor and the programs executed by it actually play a critical role in setting up and controlling the dataflow in the system. Its software is composed of the following components or programs:

- DRAM controller driver.
- Stream interleaver driver.

- Sensor control drivers for sensor configuration and phased start functionality.
- Readout functionality and interface driver.

DRAM CONTROLLER The DRAM controller driver is fairly straightforward, and includes a self-testing routine and a DIMM-specific configuration routine. The self-testing routine simply performs a number of subsequent writes and reads to a predefined DRAM region, writing specific bit patterns and checking for any bit errors when reading back the data. It is used to ensure that the DRAM is functioning correctly once after powering up the FPGA platform.

The configuration routine is used to configure the DRAM devices on the DIMM by making use of specialized DRAM commands to set specific timing, impedance and burst settings required for continuously reliable functioning of the DRAM devices. Its implementation simply follows best practices for DDR3 DRAM configuration and depends on the DIMM module being used. For further details on the DRAM configuration, please refer to the DRAM protocol specification in [JED08].

STREAM INTERLEAVER AND READOUT INTERFACE The drivers for the stream interleaver and readout interfaces are quite straightforward. The FPGA implementation of these two subsystems contain the conventional enable and reset signals that serve obvious purposes. These two input signals are wired up to four corresponding output ports on the embedded control processor, such that they are accessible from within the software. The drivers simply implement three functions: enable, disable and reset which set the respective signals of the connected subsystem. Finally, both subsystems route their individual done output signals to the embedded control processor, such that the embedded control program can detect when the stream interleaver or readout interfaces are done.

SENSOR CONTROL The sensor control driver interfaces with all of the connected image sensors through the sensor control interface described in the previous chapter. The interface subsystem signals, first described in Section 4.5, are all connected to corresponding I/O ports on the embedded control signal. This allows the processor to issue read or write commands on this interface using an 7-bit address, 8-bit data, a ready signal and an integer selector that selects the image sensor to communicate with. The actual commands being sent are sensor-specific and in compliance with the image sensor vendor's specification. They include commands to set the parameters described in Chapter 4 such as image pixel regions, shutter and exposure times, capture rate and serializer frequency, among other sensor-specific parameters. Next to commands related to configuration, capture start and stop commands are also available and allow for the phased start functionality to be implemented.

Note that the commands related to the phased start functionality are ideally executed in real-time with minimal instruction latency to ensure a phased start timing accuracy of at least 99%.

PROGRAM FLOW Each of the previous individual programs or drivers are invoked from a main program. This is simply a single function flashed into the soft-core processor ROM region that executes after the system is powered on and the soft-core processor hardware is initialized.

Excluding any soft-core specific initialization, the main program performs the following operations in order:

1. Disable and reset stream interleaver.
2. Configure DRAM (DIMM-specific commands).
3. Self-test DRAM.
4. Configure sensors (sensor-specific commands).
5. Phase start all sensors.
6. *Capture*: enable stream interleaver.
7. Wait on done signal, disable stream interleaver.
8. *Readout*: enable readout interface.
9. Wait on done signal, disable readout interface.

For this thesis, the above program flow is simple but adequate to perform *capture* of raw video data followed by *readout* of this data. The program simply ends (or waits indefinitely) after the readout has completed. A single run of the above program will produce a raw video stream on the readout interface that is transferred to the software domain host and is further processed as we will read in the next section.

5.2 STREAM DECODER

In Chapter 4, the hardware domain subsystems involved in capturing the raw video data from each of the image sensors have been described in detail, though any details on the actual structure of the raw video data have purposely been omitted. In fact, the entire capture functionality has been explicitly designed to be impervious to the actual content of whatever is to be captured and stored to DRAM. Instead, the entire task of parsing, decoding or otherwise processing this data is delegated to the software domain. Moving the processing over to the software domain aids the scalability of the platform, as it allows for easier and faster development and the possible use of existing software libraries for specific processing steps.

5.2.1 *Deinterleaving*

The first task of the stream decoder is to undo the interleaving that was performed by the stream interleaver. Described in Chapter 4, the purpose of the stream interleaver was to combine the available data words from all N connected image sensors into a single DRAM write of $N_{memory\_data\_width}$ bits. In effect, each available data word of S bits from every successive image sensor is stored one after another in DRAM, and thus also read out the same way. The deinterleaving is then done in software, where each of the individual sensor data words are split into separate streams to reconstruct the separate streams as they were transmitted by each individual image sensor. The implementation of the deinterleaving functionality is quite straightforward: the software simply parses the entire stream of raw data, reading S bits at a time and streaming them into N respective separate data streams.

5.2.2 *Bit slip correction*

One effect of the hardware domain not knowing about the raw video data is that the actual alignment of bits in the raw data is undefined. Due to potential clock drift in the differential transmission from the image sensor, data bits may appear sooner or later to the sensor receiver interface than the clock edge transitions that belong to them. The entire data stored in the internal FIFO of the receiver interface, whenever it is enabled, may therefore be shifted by one or more bits.

Correction of this effect is generally called *bit slip*, and is sometimes implemented in hardware, e.g. as part of the deserializer logic. However, correction is only possible if there is any information available about how particular words of data should look, e.g. so called fixed training patterns at a start of a data transmission. Without the presence of a protocol decoder in the sensor receiver interface, it is impossible to know if bits are out of alignment inside the hardware domain.

In practice, the effect of bit slip can easily be corrected once a fixed bit pattern is recognized in the raw data. Since the entirety of data is only shifted by a maximum of $S - 1$ bits, the correction is done by effectively deshifting with this exact amount. Since our protocol decoder is only present in the software domain, the deshifting is to be done in the software domain as well and can be integrated as part of the protocol layer decoder, simply deshifting with the required amount of bits before attempting to decode the protocol words.

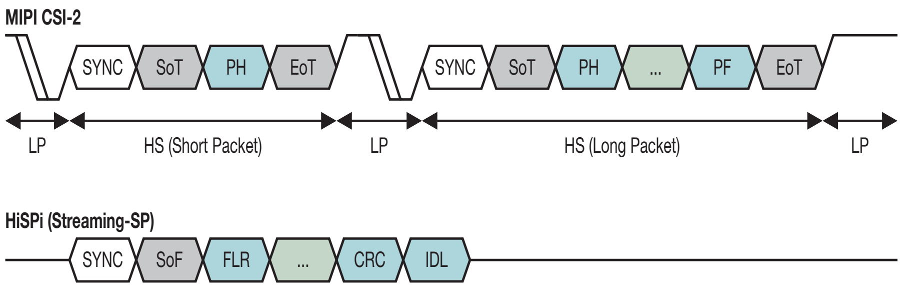


Figure 24: Simplified fragment of a video data transmission using MIPI CSI-2 (top) and HiSPi (bottom). LP indicates logic high and low using single-ended signaling, while all other signals are LVDS. Green respresents image data, all other colors represent control words as defined by the respective protocols.

5.2.3 *Protocol layer decoder*

The protocol layer defines the necessary control data to facilitate proper data transportation to the host or receiver. In case of CSI-2, the common concept of *data packets* is used: control words that describe certain synchronization events and data payload (or active image data) are all encapsulated in packets. HiSPi omits the use of packets and uses fixed-length words and special encoding to mark different payloads. Both specifications use *sync words* to mark the starting boundaries of data, allowing the receiver to properly lock on these word boundaries. This section provides an overview of the protocol layer as it is defined in these standards, and of which a simplified example can be seen in Figure 24.

MIPI CSI-2 INTERFACE The CSI-2 standard protocol layer is only active during HS (High-Speed) mode, in which high-speed serial data is actively traversing the data lanes. Each transmission consists of one or more packets, and always begins with a SoT (Start of Transmission) and ends with a EoT (End of Transmission) sync sequence. These are fixed sequences of bits with a known pattern otherwise known as sync words, allowing the receiver to correctly identify the start and end of a transmission in a stream of bits.

The SoT is followed by at least one packet, which may either be a *Short packet* or a *Long packet*. Long packets consist of a 32-bit packet header followed an arbitrary length data payload and ending with a 16-bit packet footer or checksum, and are used for transmission of active (image) data. On the other hand, short packets are used for synchronization events without payload and are thus only composed of a 32-bit packet header. The packet header in itself contains a data identifier identifying the type of packet, as well as a 16-bit word count or data field and a 8-bit Error Correction Code (ECC) to detect errors in the header.

Note that the standard uses 8-bit alignment for all data during its transmissions, and this includes packet headers, footers and payloads. As the bit depth of sensors is generally larger than 8-bit (e.g. 12-bit), MIPI-CSI2 supports a variety of data encodings that pack image data with different bit depths into the 8-bit data words. This however does require the use of extra unpacking logic in the protocol decoder.

HISPI INTERFACE The HiSPi standard can instead be seen as a continuous stream of data, using only fixed-length words and predefined bit patterns to define its data boundaries or packets. As such, it is somewhat similar to MIPI-CSI2 in the sense that it essentially fuses all its data packets together in a single continuous N-bit word transmission. It operates in four different operating modes, notably *Packetized-SP*, *Streaming-SP*, *Streaming-S* and *ActiveStart-SP8*, which mainly control the type and order of synchronization events that are transmitted. For our implementation we focus on the most common mode, Streaming-SP, which simply encodes the SoF (Start of Frame), SoL (Start of Line) and SoV (Start of Vertical Blanking Line) synchronization events.

For HiSPi, all transmissions are started with a four-word sync code consisting of an all-ones word, followed by two all-zeroes words, and finally a word describing a synchronization event such as SoF, SoL or SoV. In Streaming-SP mode, this is then followed by an optional filler word, after which the actual data payload (active image line) is located. The standard guarantees the absence of any all-zeroes words within the payload to avoid the creation of false sync codes.

The four-word sync code is used to derive the necessary alignment of words in the data stream, as mentioned in the previous section. As a first step in protocol decoding, a bit-for-bit search of the sync code is performed, and its exact bit location is used to determine the start location and alignment of the entire data stream. If the data stream is not aligned to bytes, the previously described bit slip or deshifting step is then necessary to realign the data bits back to byte alignment such that further protocol decoding can take place.

The FPGA deserializer primitives involved in interfacing with the differential transmission line may typically have a deserialization ratio S that does not match the exact word length of the image sensors or HiSPi standard. For example, deserialization may occur at $S = 8$ bits, while the image sensor sends words of 12 bits wide as is common for HiSPi-capable image sensors.

5.3 IMAGE RECTIFICATION

After the steps in the previous sections have been performed, a data stream with conventional video data should be available for further processing. In this chapter, we will describe the image rectification steps that are performed in order to minimize the distortion effects caused by the image array, using existing techniques.

5.3.1 *Mathematical models*

In order to understand and implement any form of image rectification, it is necessary to first define the fundamental mathematical models involved in camera and image processing.

We simply start off with the basic definition of a *camera*, as stated in [HZ04]:

> A camera is a mapping between the 3D world (object space) and a 2D image, and is quite simply represented by a matrix which maps from homogeneous coordinates of a world point in 3-space to homogeneous coordinates of the imaged points on the image plane.

Note that when using a CCD or CMOS sensor, light is projected onto a flat plane containing photosensitive sensors. The camera model that best describes the physical properties of this kind of projection is generally called the *basic pinhole model*, and mathetically describes the central projection (projection through a single central point) onto a plane.

BASIC PINHOLE MODEL In the basic pinhole model, we define a Euclidean coordinate system with its origin defined as the center of projection $\mathbf{C}$, and a plane Z which we call the *image plane* located at the focal point f such that $Z = f$. Any point in $\mathbb{R}^3$ described as $\mathbf{X} = (X, Y, Z)^T$ is mapped into the image plane by defining a line between this point $\mathbf{X}$ and the center of projection $\mathbf{C}$ such that the corresponding intersection point on the image plane can then be defined as $(fX/Z, fY/Z, f)^T$. Since the image plane defines a 2D space, we can describe the model as a mapping from 3D *world space* in $\mathbb{R}^3$ to 2D *image space* in $\mathbb{R}^2$ as follows:

$$(X, Y, Z)^T \mapsto (fX/Z, fY/Z)^T \tag{8}$$

EUCLIDEAN SPACE The Euclidean space has an inherent limitation that needs to be resolved when dealing with projective transformations. In particular, the concept of *infinity* poses a theoretical challenge: consider the simple case of two 2D parallel lines in $\mathbb{R}^2$. As these lines are parallel, they will never intersect. When dealing with projective geometry, however, calculations can be drastically simplified if it is guaranteed that two lines always intersect in a single point. To ensure this, the Euclidean space $\mathbb{R}^n$ is simply extended by adding the notion of points at infinity, called *ideal points*, in which (parallel) lines that would not intersect earlier will now intersect. This new space is called *projective space* or $\mathbb{P}^n$.

HOMOGENEOUS COORDINATES The extension to projective space is quite simple and made possible by the use of *homogeneous coordinates*. As described in [Fol+94], homogeneous coordinates can be defined as the extension of a given tuple of coordinates by addition of an extra coordinate that represents the scaling of all the coordinates. For example, consider a point in $\mathbb{R}^2$ as $(x, y)^T$. The homogeneous coordinates of this point are now defined by the vector $(kx, ky, k)^T$, and it is obvious that the original point can be represented by multiple homogeneous vectors, e.g. $(x, y, 1)^T$ and $(2x, 2y, 2)^T$. The use of this representation is obvious when considering the case for vector $(x, y, 0)^T$: an equivalent original point in $\mathbb{R}^2$ simply does not exist, as $x/0 = y/0 = \infty$. Using $k = 0$ for homogeneous coordinates thus serves as a means of describing the *ideal points* of projective space, and cements the use of these homogeneous in extending Euclidean space to projective space, which holds for any dimensional extension from $\mathbb{R}^n$ to $\mathbb{P}^n$. Other interesting properties of homogeneous coordinates include:

- Origin points in Euclidean space are represented as non-infinite homogeneous coordinates, e.g. $(0, 0, 1)$ for $\mathbb{R}^2$.
- Points at infinity $(k = 0)$ in $\mathbb{P}^2$ form a line "at infinity".
- Points at infinity $(k = 0)$ in $\mathbb{P}^3$ form a plane "at infinity".

From hereon we will explicitly denote homogeneous vectors, such that any 3D point $\mathbf{X} = (X, Y, Z)^T$ can also be described as the homogeneous vector $\widetilde{\mathbf{M}} = (kx, ky, kz, k)^T = (X, Y, Z, 1)^T$ with homogeneous coordinate $k = 1$.

By using homogeneous vectors, k may change after certain matrix multiplications. By ensuring that the vector is always scaled back to $k = 1$, the so called "perspective division", which accounts for the perspective scaling of objects in the image, is inherently taken into account.

5.3.2 *Camera intrinsics and extrinsics*

Going back to the basic pinhole model, we can now generally describe the operation of a projective camera as a linear mapping using homogeneous coordinates:

$$\begin{pmatrix} x \\ y \\ w \end{pmatrix} = \mathsf{P} \begin{pmatrix} \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Z} \\ W \end{pmatrix} \tag{9}$$

or even more compact as:

$$\mathbf{x} = \mathsf{P}\mathbf{X} \tag{10}$$

Where P is a 3×4 matrix called the *camera matrix* and describes the basic pinhole model using central projection, or the relationship between any arbitrary 3D point $\mathbf{X}$ and its projection on the image plane $\mathbf{x}$.

The camera matrix contains all information on the camera's parameters, and these parameters are generally split into two convenient groups:

- *Instrinsic* parameters describing the internal geometric properties of the camera such as focal length, principal point coordinates, image scaling and skew.
- *Extrinsic* parameters describing the external orientation of the camera, including its translation and rotation.

The explicit distinction between the internal and external properties of the camera makes sense, as it allows for the further decomposition of the camera matrix into the product of two separate matrices for these properties:

$$\mathsf{P} = \mathsf{K}[\mathsf{R}\ \mathbf{t}] \tag{11}$$

Where matrix K is a 3×3 matrix describing the intrinsic parameters of the camera, R is a 3×3 rotation matrix describing the extrinsic rotation of the camera, and $\mathbf{t}$ is the translation vector describing the position of the world origin in camera coordinates. $\mathbf{t}$ can be further decomposed to make use of the homogeneous center of projection we have defined earlier:

$$\mathbf{t} = -\mathsf{R}\widetilde{\mathbf{C}} \tag{12}$$

Note that in Equation 11, P has been written as a block matrix for convience, where R and $\mathbf{t}$ represent a left-to-right concatenation columns of matrix P.

INTRINSIC MATRIX The camera intrinsic matrix can be broken down into a number of individual internal parameters that will generally vary between different types of cameras. As we will see later on, the variation of these internal parameters are certain to cause a degree of unwanted distortion in captured images, making the decomposition into individual parameters very important. We therefore use the intrinsic matrix form for K as described in [Zhaoo]:

$$K = \begin{pmatrix} \alpha & \gamma & x_0 \\ 0 & \beta & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{13}$$

Where x_0 and y_0 are the coordinates of the *principal point* or the origin of coordinates in the image plane, α and β are the *scale factors* in the two axes x and y of the image plane, and γ is the *skew coefficient* of these two image axes. More specifically:

- The *principal point* $\widetilde{\mathbf{x}} = (x_0, y_0)^T$ is simply the point at which the *principal ray*, or the ray that is perpendicular to the image plane and passes through the center of projection **C**, intersects with the image plane. This intersection point is otherwise interpreted as the origin of coordinates in the image plane of the camera. The principal point is a 2D point in *image space* and is described in terms of the two image plane axes x and y.

- The *scale factors* α and β describe the amount of scaling applied to the two image plane axes x and y. Whereas the most simple basic pinhole model assumes equal scales along both of these axes, in practice, this scaling varies among different cameras. Examples include sensors with non-square pixels, the use of anamorphic lenses or distortion due to fabrication flaws in the sensors and lenses.

- The *skew coefficient* γ defines the skew of the image plane in terms of the deviation of the normal perpendicular angle between the two image plane axes x and y. γ is ideally 0, but skew can be introduced by non-perpendicular mounting of lenses in relation to the sensor. Since this is a problem that can be avoided in practice, we will assume that from here on $\gamma = 0$.

HOMOGRAPHY Rotation matrix R and translation vector $\mathbf{t}$ represent the extrinsic parameters or orientation of the camera. Using the rotation matrix, the projection transformations in Equations 10 and 11 can be simplified even further when assuming that the image plane Z noted earlier is located at 0:

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = K[\mathbf{r_1}\ \mathbf{r_2}\ \mathbf{r_3}\ \mathbf{t}] \begin{pmatrix} X \\ Y \\ 0 \\ 1 \end{pmatrix} = K[\mathbf{r_1}\ \mathbf{r_2}\ \mathbf{t}] \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} \tag{14}$$

Where $\mathbf{r_i}$ represent the individual columns of the rotation matrix R. Note that the third column of this matrix is removed from the equation due to the fact that $Z = 0$. A new matrix can then be defined:

$$H = K[\mathbf{r_1}\ \mathbf{r_2}\ \mathbf{t}] \tag{15}$$

Here, H is a 3×3 matrix called the *homography matrix* and describes the relationship between an arbitrary 3D point **X** and its projection on the image plane **x** at $Z = 0$.

INTRINSIC LENS DISTORTION One important intrinsic parameter that is still missing from the aforementioned intrinsic matrix is *lens distortion*. As mentioned in [Bou08], this lens distortion is defined as a number of image distortion coefficients k_i describing the radial and tangential lens distortion parameters of the lens mounted in the camera.

To model this type of distortion, Equation 14 is rewritten as follows:

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = K[\mathbf{r_1}\ \mathbf{r_2}\ \mathbf{t}] \begin{pmatrix} X_d \\ Y_d \\ 1 \end{pmatrix} \tag{16}$$

Where X_d and Y_d the distorted counterparts of coordinates X and Y, also defined as:

$$\begin{pmatrix} X_d \\ Y_d \\ 1 \end{pmatrix} = (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + k_5 \cdot r^6) \begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} + \mathbf{dx} \tag{17}$$

And **dx** is the so called *tangential distortion vector* defined by:

$$\mathbf{dx} = \begin{pmatrix} 2k_3 \cdot X \cdot Y + k_4(r^2 + 2X^2) \\ k_3(r^2 + 2Y^2) + 2k_4 \cdot X \cdot Y \end{pmatrix}$$

Here, X and Y are the homogeneous coordinates from Equation 17, k_i are the lens distortion coefficients and $r^2 = X^2 + Y^2$ describes the radial relationship that accounts for the radial distortion. In [Bou08] and [Zha00] it is established that the lens distortion is dominated by only the first two coefficients k_1 and k_2.

For convenience, we will define the relation from a 3D point $\mathbf{X}$ to its projection $\mathbf{x}$ on the image plane as the function $\widetilde{\mathbf{x}}(K, k_1, k_2, R, \mathbf{t}, \mathbf{X})$ where all parameters correspond to the aforementioned definitions.

5.3.3 *Rectification*

As shown, any camera is inherently bound by its intrinsic and extrinsic parameters. These parameters obviously manifest themselves in the images taken by the sensor of that camera by means of influence on the projection. Thus, the ultimate goal of image rectification is to ensure that all parameters are constrained, such that the images conform to an expected projection transformation.

In practice, this means that image rectification is essentially retroactive, where already existing images are corrected in such a way that any unwanted influence of the parameters on the projection of these images is made undone through the following general steps:

1. Estimation of all intrinsic and extrinsic parameters of the camera, also called *camera calibration*.

2. Calculation of a mapping function describing the difference between a projection transformation with the expected and actual parameters. This function maps a distorted image to an undistorted ideal image.

3. Remapping of all images with the correct parameters using the aforementioned mapping.

The result of the image rectification on existing images taken by the camera is a reprojected set of those images that match as closely as possible to the expected intrinsic and extrinsic parameters. Note that due to the nature of the parameters, separate mapping functions will exist for both intrinsic and extrinsic parameters.

5.3.4 *Camera calibration*

The process of camera calibration that we use was first described in [Zha99] and involves estimating the intrinsic and extrinsic camera parameters that have been described in the previous section. This camera calibration is performed using the following general steps:

1. Initial estimation of the intrinsic and extrinsic parameters, as further described in [Zha99].

2. Refinement of all parameters by minimizing the difference between a set of known points on the image plane, and a set of

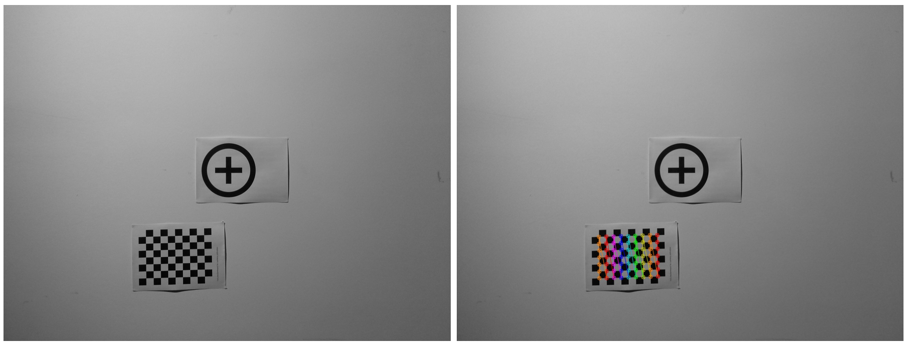

(a) Raw grayscale input image. (b) Output image with internal corners marked in color.

Figure 25: OpenCV Feature detection using a planar checkerboard pattern as mounted to a wall. The visual marker in the center is not actually used in the algorithm.

projected points on the image plane using the estimated parameters.

In the above steps, it is important to note the need for a set of *known points* on the image plane. These known points are typically gathered by capturing a number of camera images containing a checkerboard pattern. This pattern is assumed to lie on the image plane at $Z = 0$, and a feature detection algorithm such as in [HS88] is used to deduce the 2D points that correspond to the internal corners of the checkerboard. A visual example of this algorithm can be seen in Figure 25, in which the colored points represent the known 2D points on the image plane.

The known points are then used in the following refinement algorithm, which uses n checkerboard images and m known points on the image plane as input:

$$\sum_{i=1}^{n}\sum_{j=1}^{m} ||\mathbf{x}_{ij} - \tilde{\mathbf{x}}(K, k_1, k_2, R_i, \mathbf{t}_i, \mathbf{X}_j)||^2 \tag{18}$$

Where $\tilde{\mathbf{x}}(K, k_1, k_2, R, \mathbf{t}, \mathbf{X})$ is the earlier defined projection function, $\mathbf{x}_{ij}$ is a known point on the image plane for each internal chessboard corner on each image and $\mathbf{X}_j$ corresponds to the 3D point for each of these corners. Note that $\mathbf{X}_j$ is independent of the image i, since for each image different extrinsics R_i and $\mathbf{t}_i$ are assumed, and all $\mathbf{X}_j$ points are thus reduced to a general precalculated set of 3D points corresponding to the checkerboard pattern at $Z = 0$.

Equation 18 effectively projects a point $\mathbf{X}_j$ using a set of camera parameters and determines the difference, or *reprojection error*, between this projected point and an already known point. Ideally, the two points are identical and the reprojection error is simply zero, representing an ideal projection function. In practice, the error is not zero but is minimized by adjusting camera parameters K, k_1, k_2, R_i and $\mathbf{t_i}$. This minimization is generally done using the well established Levenberg-Marquardt algorithm as described in [Mor78]. The result is a set of camera parameters that correspond as closely as possible to the observable distortion in the checkerboard images captured by the camera, and thus approximate the actual parameters of that camera as closely as possible.

6 DATAFLOW ANALYSIS

In Chapter 1, the evaluation criteria for this thesis were introduced. Recall that the first criterion states that the system must be capable of interfacing with at least 16 image sensors with a capture rate of 60 Hz. This very criterion imposes a very real constraint on the minimal throughput that the system must sustain in order to interface with all of these sensors and to capture and store all their respective data properly in memory. In this chapter, we will discuss the theoretical analysis that serves to give an evaluation indication for this particular criterion, and investigate the required buffer size in case 16 image sensors are interfaced with our system.

Chapter 3 first described the two essential dataflow stages in the hardware domain: *capture* and *readout*, respectively capturing the raw data from the connected image sensors followed by subsequent storage and the transmission of this captured data to the software domain for further off-line processing. Remember that timing constraints are imposed on the capture stage, as it is a stream of data that is to be captured without interruptions that would otherwise lead to data corruption. It is therefore this critical capture stage that must be analyzed in order to make any claims about the throughput of our system, and in turn, the viability of our platform.

The previous chapters have made it clear how the dataflow within this capture stage is set up, as also illustrated before in Figure 9: data starts flowing from the image sensors, into the sensor receiver interfaces, followed by the stream interleaver, and finally ending up in the DRAM controller. The throughput of all components before the DRAM controller is essentially fixed and predictable, as these components simply process data at the rate at which the image sensors are sending without any external interruptions or influences whenever they are enabled. The DRAM controller is however dependent on external DRAM devices which exhibit their own timing and throughput constraints, implying that the DRAM controller throughput is in fact restricted. We therefore identify the primary bottleneck in the system to be the the streaming throughput of the DRAM controller, or the maximum rate at which data can be stored in DRAM. Moreover, since the DRAM controller is not always available to write, a queue mechanism must be put into place to temporarily buffer the requests while the DRAM is not available.

Correct functioning of the system therefore requires a guarantee that none of the FIFOs involved in communicating the raw data from the image sensors into the DRAM controller experience any buffer overflows due to limited consumption of data by the DRAM controller, as such an event would lead to a diminishing integrity of all data and data corruption. In this sense, we observe that our system is *hard real-time* as described in [But11] and [Goo+16] since missing any deadlines involved in the sensor dataflow will lead to total loss of system usefulness in the worst case.

Real-time analysis of the system allows us to verify whether the maximum sustained throughput of the system, or specifically the DRAM controller as is, is high enough to interface with at least 16 image sensors streaming at 60 Hz without causing overflows in the system, and get an indication of the FIFO size required to sustain such a dataflow.

In Section 4.2, we have described our specialized DRAM controller design. With regard to the maximum sustained throughput of this DRAM memory controller, DRAM writes can only occur when the DRAM controller state machine is in a state where a write request can be handled. There are various possible scenarios in which this is not the case and write commands can only be handled after the controller returns to such a state: immediately after DRAM initialisation, during a DRAM read request, a periodic DRAM refresh, or when precharging and activating a new DRAM row. The only scenarios that are really relevant during the sustained DRAM burst writes of our capture stage are the DRAM refresh and precharge and activation of rows, since they can simply not be avoided, even during sustained writes. Moreover, these two scenarios will cause a significant delay of DRAM writes whenever they occur, diminishing the maximum sustained throughput of the DRAM controller as a whole, and thus are to be modelled accordingly in our dataflow analysis.

6.1 THROUGHPUT ANALYSIS

By performing timing analysis of the system's critical components, one gains information on the overall throughput of the system, which in turn allows verification of the given evaluation criteria. As such, the relevant components involved are to be modelled so that a suitable analysis method can be applied, and there are various ways to do so as we will see below.

Previous work in [Goo+16], [AG11] shows that worst-case timing analysis of DRAM controllers integrated in real-time embedded systems is an active field of research. Here, an effort is made to model these controllers in terms of *memory patterns*, or small predictable sequences of DRAM commands, that accurately model different scenarios such as read-after-write, write-after-read and refresh that are common in conventional DRAM controllers. The analysis is tackled by formulating the behaviour as a scheduling problem, choosing a scheduling algorithm and evaluating the quality of this algorithm in terms of worst case performance. In [Kim+15], a similar approach is presented that also prioritizes different levels of criticality in access. Though this prior research proves to be very useful for typical DRAM use scenarios, our case is atypical in the sense that there exists only a single write access pattern for our critical stage, reducing the analysis to a much simpler and predictable problem without the need for scheduling algorithms or prioritization.

The aforementioned research mostly depends on the use of programmatic numerical algorithms for the calculation of worst-case latency and throughput. In [Li+16] however, a *Timed Automata* (TA) model is presented that is used to perform analysis using a set of graph-based models. These models structure the DRAM controller behaviour in terms of a fairly complete set of *templates* that represent the read and write commands as well as most of the internal states of the controller that influence the worst-case throughput. The templates include behaviour imposed by read-after-write and write-after-read scenarios, as well as low-level DRAM timing such as row precharging and activation. The use of the TA model not only provides a more compelling visual overview of the system, but it also permits the use of existing analysis and verification tools such as the Uppaal model checker [Beh+06].

A different but similar type of DRAM controllers model for real-time embedded systems is discussed in [LBW09]. Here, a *Synchronous Dataflow Graph* (SDF) is used to model the non-uniform access latency of contemporary DRAM controllers. The SDF graph, first described in [LM87], has proven useful for rapid prototyping and implementation of relatively simple stream-based embedded systems such as DSPs [SB00], further aided by the availability of analysis tools such as SDF^3 as described in [SGB06].

The two aforementioned tools greatly benefit the practical use of timing analysis, as they avoid the need to resort to purely programmatic numerical methods. The capabilities of the tools play an important role in deciding a suitable type of model for analysis of our own system. While both tools have been briefly evaluated, limitations in the execution time analysis capabilities in Uppaal have led us to believe that the SDF^3 tool, together with a suitable SDF graph, would be most suitable for the analysis in this thesis.

6.1.1 *Scenario-aware dataflow graph*

Our first model serves as an approximation for throughput analysis and uses the FSM-based Scenario-Aware Dataflow (FSM-SADF) graph as described in [Siy+11]. The FSM-SADF graph allows us to properly express the different scenarios of the DRAM controller, where the controller can either be in a state where it can accept writes (e.g. the idle state), or in a state where it is refreshing and unresponsive.

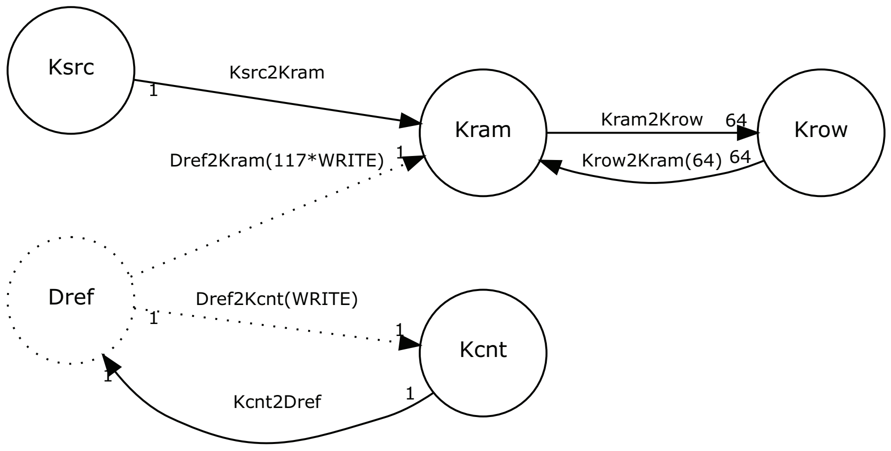


Figure 26: SADF graph representing the video streaming behaviour in our system and used as a basis for dataflow analysis.

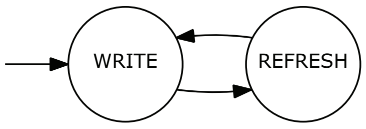


Figure 27: Corresponding FSM for the *Dref* node in our dataflow model, representing write and refresh states with write as initial state.

In order to perform a straightforward dataflow analysis, it is essential to make the dataflow model as simple as possible. We therefore follow these rules that approximate the critical behaviour of our system dataflow:

- The DRAM controller is the main subject of the dataflow model, as it is the primary bottleneck in the system.
- Nodes in the model exhibit different execution times.
- Cycles in the model are executed at a frequency equal to that of the DRAM controller, specifically 150 MHz or 6.67 ns according to DDR3-600 operation.
- A single DRAM write command is assumed to take 2 cycles, equal to the execution time of Kram. A token produced by Kram therefore represents one issue of a DRAM write command.
- Only the DRAM burst write behaviour is modelled, as it represents the critical flow of data in the system.
- The input to the DRAM controller is modelled as a single consolidated node Ksrc. In reality, Ksrc represents all image sensors, sensor receiver interfaces and the stream interleaver components as they stream into a single input from the DRAM controller's perspective.

- Tokens from Ksrc represent DRAM write commands, e.g. writing a single row of data.
- DRAM row precharging and activation is modelled as an event that occurs every 64 cycles or writes by means of a separate Krow node.
- Krow produces and consumes 64 tokens at a time with an execution time of 14 cycles, effectively delaying the execution of Kram every 64 tokens.
- The DRAM controller operates in two scenarios: WRITE, in which it accepts write commands, and REFRESH, in which it is unresponsive to commands.
- The DRAM scenarios are included in Kram such that commands (tokens) from Ksrc are stalled in the graph whenever the REFRESH scenario is active.
- DRAM controller scenarios in Kram are changed by control tokens sent from Dref.
- DRAM refresh is modelled as an event that occurs every 117 cycles, and takes 4 cycles of time. Kcnt models a counter that counts to 117, initiates the REFRESH scenario in Dref, counts to 4, and initiates the WRITE scenario in Dref. Together, these two nodes trigger the scenario transitions in Kram by means of control tokens.
- The throughput of Ksrc represents the maximum sustained throughput of the system.

The dataflow analysis occurs somewhat backwards: if we can derive the throughput of Ksrc using SDF^3, we will know the throughput at which DRAM write commands can be handled by the system as modelled using the SADF graph. Here, SDF^3 can provide us with the throughput of any node in the graph by performing numerical analysis.

In the model, the refresh functionality is implemented using scenarios. This is necessary due to the fact that the refresh must occur completely independent and decoupled from any other functionality in the system, e.g. its interval is purely time-based and not dependent on writes. Both the refresh interval and duration could therefore not be properly modelled as a single node connected to Kram (e.g. similar to the action of Krow). Note that refresh occurs no matter what and regardless of the state the DRAM controller is in. This decoupling action is achieved by the use of SADF scenarios.

One thing that does not follow from the model in Figure 26 are the execution times for each of the nodes (kernels), which are as follows:

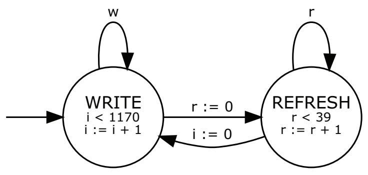


Figure 28: Imaginary FSM for the *Dref* node in our dataflow model, using counters to switch between states and scenarios.

- Ksrc: 1 cycle.
- Kram: 2 cycles in WRITE scenario, none in REFRESH scenario.
- Krow: 14 cycles.
- Kcnt: 117 cycles in WRITE scenario, 4 in REFRESH scenario.

In the above rules, adjustments have been made to work around limitations of the SDF³ toolkit. In reality, the DRAM refresh t_{REFI} occurs every 1170 cycles with a duration t_{RFC} of 39 cycles, but this number has been divided by 10 to avoid state explosion such that the model could be calculated.

Also, due to limitations in the expressiveness of SADF graphs, the behaviour of node *Dref*, and by extension *Kcnt*, is somewhat convoluted. Ideally, we would like to model the FSM in *Dref* using counters, as can be seen in Figure 28, avoiding the need for a separate node *Kcnt*. Here, *i* and *r* respectively represent the counters for t_{REFI} and t_{RFC}. However, expressing such counters in a FSM and any subsequent state unfolding is currently not supported in the SDF³ toolkit. An even more fitting model might have been possible by using Timed Automata, however the Uppaal tool in its current state lacks the execution time analysis functionality that we require for proper throughput analysis.

6.1.2 *Throughput analysis*

The actual XML data corresponding to the SADF in Figure 26 can be found in Appendix A. Throughput analysis of this graph using the SDF³ toolkit yields a throughput of 0.450704 tokens per time unit of 6.67 ns, or effectively a maximum DRAM write frequency f_{WRITE} of $0.450704 \times 150\,\text{MHz} \approx 67.6\,\text{MHz}$. Considering the fact that 512 bits are written for every DRAM write command, we can then estimate the theoretical maximum sustained throughput of the DRAM controller as modelled using this SADF graph:

$$\begin{aligned} f_{WRITE} \times N_{memory\_data\_width} &= 67.6\,\text{MHz} \times 512\,\text{bits} \\ &\approx 33.8\,\text{Gbit/s} \\ &\approx 4.23\,\text{GiB/s} \end{aligned}$$

We can now compare these analyzed values to the theoretical peak transfer rate for DDR3-600 as earlier calculated in Equation 2 in Chapter 4:

$$\begin{aligned} \text{peak transfer rate} \times \text{DRAM width} &= 600\,\text{MT/s} \times 64\,\text{bits} \\ &= 37.5\,\text{Gbit/s} \\ &= 4.6875\,\text{GiB/s} \end{aligned}$$

As one would expect, we can see that the overhead caused by DRAM refresh, row precharge and row activation times indeed have a significant effect on the theoretical maximum throughput of the DRAM, reducing the throughput to approximately 90% of the theoretical peak transfer rate as modelled by our SADF graph.

6.1.3 *Effects on array size*

Knowing the theoretical maximum throughput or f_{WRITE} as derived from our previous dataflow analysis, we can plug this throughput into a rewrited form of our earlier Equation 6 to retrieve an estimation of the maximum sensor array size that we can support:

$$\begin{aligned} N &= \left\lfloor \frac{N_{memory\_data\_width} \times f_{WRITE}}{N_{lanes} \times f_{SERIAL}} \right\rfloor \\ &= \left\lfloor \frac{512\,\text{bits} \times 67.6\,\text{MHz}}{2 \times 445.5\,\text{MHz}} \right\rfloor \\ &= 38\,\text{sensors} \end{aligned}$$

Here, we have assumed a DRAM controller operating at DDR3-600, as well as a 445.5 MHz serial bit rate and 2 lane configuration for all sensors identical to our real-world prototype described in the next chapter, yielding a maximum amount of 38 sensors purely based on the theoretical maximum throughput of the DRAM controller and not taking into account other factors such as FPGA resource or physical hardware constraints and interleaver limitations. Given the above figures, our first evaluation criterion in fact looks to have been satisfied: using the given DRAM controller, our model indicates that our system is capable of interfacing with at least 16 (38) image sensors at a capture rate of 60 Hz.

6.2 BUFFER SIZE ANALYSIS

In Section 4.1, we have outlined the timing characteristics of a single sensor: a sensor simply streams at the fixed rate at which it is configured to stream. The data of two or more of these sensors are then fed

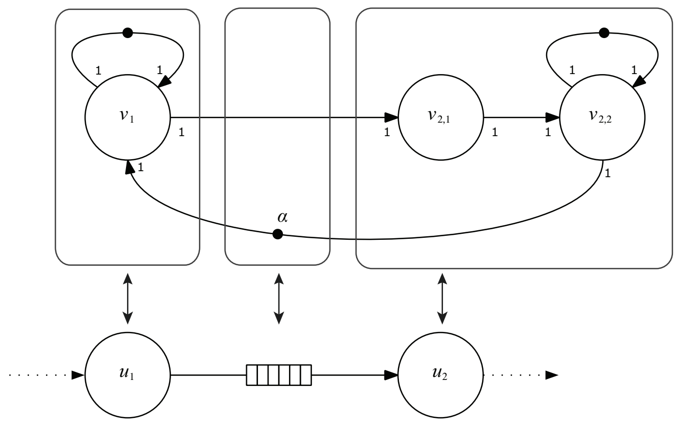


Figure 29: Simplified latency-rate model of our system, with the SRDF graph at the top and the corresponding task graph at the bottom. Dotted lines represent the imaginary flow of data in the system, and are not part of the actual graph.

into the interleaver, as explained in Section 4.3, simply combining the data such that the DRAM controller data bus is maximally saturated. Next to throughput, we are also especially interested in estimating the size of any FIFO buffers involved in this critical flow.

Recall that during the capture phase, data is flowing from the sensors into the sensor receiver interfaces, the stream interleaver and finally into the DRAM controller. While sensors will always be transmitting data, the DRAM controller may not always be immediately available to write data. This means that a certain degree of buffering is required before the DRAM controller. While the sensor receiver interfaces already contain a FIFO to provide safe clock domain crossing of data, for this model, we will assume that our stream interleaver contains a FIFO that acts as a buffering mechanism for DRAM writes whenever the DRAM controller is not ready to accept data. It is critical that the FIFO is large enough such that an overflow never occurs, as this could lead to corruption of data.

6.2.1 *Latency-rate SRDF graph*

In order to make an estimation of the FIFO size, we have chosen to make use of latency-rate analysis, introduced in [WBS07]. This analysis makes use of Single-Rate Data-Flow (SRDF) graphs, also known as Homogeneous Synchronous Data-Flow (HSDF) graphs, as described in [LM87], in which the latency and rate are modelled as separate nodes. This type of analysis is especially suited for streaming systems such as the one presented in this thesis.

Figure 29 shows the simplified model used for the latency-rate analysis of our system. Task u_1, also known as the *source*, represents all the components in the system before the DRAM controller, while task u_2 represents the DRAM controller. The FIFO in between represents the FIFO in the stream interleaver for which we are analyzing the size.

From the figure, it follows that the actors v_1, $v_{2,1}$ and $v_{2,2}$ correspond to their task graph equivalents u_1 and u_2. In particular, actors $v_{2,1}$ and $v_{2,2}$ respectively represent the average *latency* and *rate* of the DRAM controller for our latency-rate analysis. Furthermore, α holds the amount of tokens and corresponds to the FIFO's size.

The above general model can be used to calculate α once the firing durations for all SRDF actors v_1, $v_{2,1}$ and $v_{2,2}$ have been determined. For simplicity, we will assume that the time units of these firing durations are in fact cycles corresponding to system frequency f_{SYSTEM}, e.g. at 150 Mhz. These firing durations, from hereon referred to as ρ_{v_1}, $\rho_{v_{2,1}}$ and $\rho_{v_{2,2}}$ can then be derived directly using our earlier equations as they directly correspond to component rates in our system.

First of all, ρ_{v_1} simply corresponds to the average rate at which the stream interleaver will be issuing write commands to the DRAM controller

$$\rho_{v_1} = \left\lfloor \frac{f_{\text{SYSTEM}}}{f_{\text{WRITE}}} \right\rfloor$$

Next, the *rate* actor $v_{2,2}$ represents the average rate at which the DRAM controller can perform DRAM write commands. Recall that DRAM writes are occasionally stalled by either DRAM refresh or a row precharge and activation events, which respectively occur at an interval of t_{REFI} and every 64 DRAM writes, the duration of which are specified by variables t_{RFC}, t_{ROW}, t_{CWL}, t_{WR}, t_{RP} and t_{RCD} as first introduced in Section 4.2. We calculate the overall average duration of the DRAM refresh events:

$$r_{\text{REFRESH}} = \frac{t_{\text{RFC}}}{t_{\text{REFI}}}$$

Then, in order to calculate the overall average duration of the row precharge and activation events, we first calculate the exact duration t_{PREACT} of a row precharge and activation event after a DRAM write:

$$t_{\text{PREACT}} = t_{\text{ROW}} + t_{\text{CWL}} + t_{\text{WR}} + t_{\text{RP}} + t_{\text{RCD}}$$

Here, t_{ROW}, t_{CWL} t_{WR} equal the amount of time it takes for the previous DRAM write command to finish, due to any potential bursting action, while t_{RP} and t_{RCD} equal the actual precharge and activation times.

Next to the event duration, we require the event interval. Since we assume that the event occurs after every 64 writes, the interval is as follows:

$$t_{PREACTI} = 64 \times \rho_{v_1}$$

Note that here, ρ_{v_1} is used to indicate the actual average duration of a write. Now that we know the interval and duration, we can determine the overall average duration:

$$r_{PREACT} = \frac{t_{PREACT}}{t_{PREACTI}}$$

Knowing the overall average durations of the events that will influence the DRAM write duration, we can calculate the average duration of a DRAM write, or the average DRAM write *rate* corresponding to actor $v_{2,2}$, using the normal DRAM write duration t_{ROW} and taking all these factors in account:

$$\begin{aligned}\rho_{v_{2,2}} &= t_{ROW} + r_{PREACT} + r_{REFRESH} \\ &= t_{ROW} + \frac{t_{ROW} + t_{CWL} + t_{WR} + t_{RP} + t_{RCD}}{64 \times \rho_{v_1}} + \frac{t_{RFC}}{t_{REFI}}\end{aligned}$$

We are now left with calculating the maximum latency as modelled by actor $v_{2,1}$. For this latency, we assume the maximum possible amount of time spent waiting for the DRAM controller to finish a write when it is maximally stalled, which is the case whenever a write is stalled by both a refresh event and a row precharge and activation event:

$$\theta = t_{ROW} + t_{RFC} + t_{PREACT}$$

The firing duration of actor $v_{2,1}$, which in fact excludes any time spent waiting in the modelled FIFO, can then be calculated by subtracting the firing duration $\rho_{v_{2,2}}$ which is accounted for by actor $v_{2,2}$:

$$\begin{aligned}\rho_{v_{2,1}} &= \theta - \rho_{v_{2,2}} \\ &= t_{ROW} + t_{RFC} + t_{PREACT} - \rho_{v_{2,2}}\end{aligned}$$

Now that the firing durations ρ for all actors are known, it is possible to determine the amount of tokens α on the edge between $v_{2,2}$ and v_1 representing the FIFO size. In order to do this, we use the Maximum Cycle Mean (MCM) equation as in [Bac+01]:

$$\mu = \max_{c \in C} \frac{\sum_{v \in V(c)} \rho_v}{\sum_{e \in E(c)} \delta_e}$$

Where C is the set of directed simple cycles in our SRDF graph, $V(c)$ is the set of actors on the cycle c, $E(c)$ is the set of edges on the cycle c, and δ_e represents the number of tokens on the edge e.

Taking all three cycles in our SRDF graph into account, the MCM for our particular model is then calculated as follows using three terms:

$$\mu = \max\left(\frac{\rho_{v_1}}{1}, \frac{\rho_{v_{2,2}}}{1}, \frac{\rho_{v_1} + \rho_{v_{2,1}} + \rho_{v_{2,2}}}{\alpha}\right)$$

The MCM is equal to the inverse of the throughput of the graph. Since our throughput must be at least enough for the sensors to write, we must constrain the upper bound of the MCM. We require a guarantee that the *source* actor v_1 must always write at its own firing rate, and thus must not be delayed by anything else in the graph. The MCM may therefore not exceed ρ_{v_1}, to avoid loss or corruption of data:

$$\mu < \rho_{v_1}$$

Using this constraint, we can simply solve for α by rewriting the last dominant term of the MCM:

$$\frac{\rho_{v_1} + \rho_{v_{2,1}} + \rho_{v_{2,2}}}{\alpha} < \rho_{v_1} \implies \alpha = \left\lceil \frac{\rho_{v_1} + \rho_{v_{2,1}} + \rho_{v_{2,2}}}{\rho_{v_1}} \right\rceil \tag{19}$$

The above equation naturally assumes that DRAM writes can actually be handled at the rate they are issued in the first case, e.g. the average duration of a DRAM write or $\rho_{v_{2,2}}$ is in fact lower than the write rate or ρ_{v_1}. The value for α is then assumed as the required FIFO size.

6.2.2 *Real-world case analysis*

To get an indication of the required FIFO size or α for a real-world case, we will consider an implementation of our system that uses our earlier established minimum of 16 sensors, but is otherwise identical to our real-world prototype described in Chapter 7. By using actual timing values from our prototype, and using 16 sensors, we in fact calculate a realistic FIFO size requirement for a scaled up prototype.

The known parameters and variables for this case are as follows:

$$\begin{aligned}
N &= 16\,\text{sensors} \\
S &= 8\,\text{bits} \\
N_{memory\_data\_width} &= 512\,\text{bits} \\
N_{lanes} &= 2\,\text{lanes} \\
f_{SERIAL} &= 445.5\,\text{MHz} \\
\\
f_{SYSTEM} &= 150\,\text{MHz} \\
t_{WR} &= 3\,\text{cycles}\ \textit{(write recovery)} \\
t_{RP} &= 3\,\text{cycles}\ \textit{(row precharge)} \\
t_{RCD} &= 3\,\text{cycles}\ \textit{(row activate)} \\
t_{ROW} &= 2\,\text{cycles}\ \textit{(row write)} \\
t_{CWL} &= 2\,\text{cycles}\ \textit{(CAS write latency)} \\
t_{REFI} &= 1170\,\text{cycles}\ \textit{(refresh interval)} \\
t_{RFC} &= 39\,\text{cycles}\ \textit{(refresh duration)}
\end{aligned}$$

Note that any cycle-based values are clocked at the system frequency f_{SYSTEM}. The write rate is then calculated using Equation 6:

$$\begin{aligned}
f_{WRITE} &= \frac{f_{SERIAL} \times N \times N_{lanes}}{N_{memory\_data\_width}} \\
&= \frac{445.5\,\text{MHz} \times 16 \times 2}{512} \\
&\approx 27.84\,\text{MHz}
\end{aligned}$$

We then calculate the row precharge and activation duration using our known values:

$$\begin{aligned}
t_{PREACT} &= t_{ROW} + t_{CWL} + t_{WR} + t_{RP} + t_{RCD} \\
&= 2 + 3 + 3 + 3 + 3 \\
&= 14\,\text{cycles}
\end{aligned}$$

Using all of the above values, we determine the firing durations for all our SRDF actors:

$$\begin{aligned}\rho_{v_1} &= \left\lfloor \frac{f_{\mathrm{SYSTEM}}}{f_{\mathrm{WRITE}}} \right\rfloor = \left\lfloor \frac{150\,\mathrm{MHz}}{27.84\,\mathrm{MHz}} \right\rfloor = \lfloor 5.39 \rfloor \\ &= 5\,\text{cycles}\end{aligned}$$

$$\begin{aligned}\rho_{v_{2,2}} &= t_{\mathrm{ROW}} + \frac{t_{\mathrm{PREACT}}}{64 \times \rho_{v_1}} + \frac{t_{\mathrm{RFC}}}{t_{\mathrm{REFI}}} \\ &= 2\,\text{cycles} + \frac{14\,\text{cycles}}{320\,\text{cycles}} + \frac{39\,\text{cycles}}{1170\,\text{cycles}} \\ &\approx 2.077\,\text{cycles}\end{aligned}$$

$$\begin{aligned}\rho_{v_{2,1}} &= t_{\mathrm{ROW}} + t_{\mathrm{RFC}} + t_{\mathrm{PREACT}} - \rho_{v_{2,2}} \\ &= 2 + 39 + 14 - 2.077 \\ &\approx 52.923\,\text{cycles}\end{aligned}$$

Using the rewritten form of the last term in the MCM in Equation 19, we calculate α:

$$\begin{aligned}\alpha &= \left\lceil \frac{\rho_{v_1} + \rho_{v_{2,1}} + \rho_{v_{2,2}}}{\rho_{v_1}} \right\rceil = \left\lceil \frac{5 + 52.923 + 2.077}{5} \right\rceil = \lceil 12 \rceil \\ &= 12\end{aligned}$$

Our model therefore indicates that a FIFO size of $\alpha = 12$, accommodating 12 writes, is required for 16 sensors, when considering average latency-rate characteristics. As we will see in the next chapter, this order of FIFO size is quite small in terms of FPGA resources and shows that the FIFO is not a significant factor in terms of FPGA resources required in order to implement our system using an identical or similar configuration in the real-world.

7 REALIZATION AND RESULTS

Up until this point, we have presented our embedded system platform from a mostly theoretical point of view. This theory would just remain theory if it was not for an actual practical realization of the platform. In this chapter, we will introduce the very first prototype that has been produced according to the design outlined in this thesis. As we will see, this very prototype allows us to perform all-important measurements that will verify the evaluation criteria that are essential in demonstrating the viability of the platform as a high-speed imaging solution.

This first version of the prototype has been designed to support only a small amount of image sensors to make up the sensor array, such that the platform could be evaluated in the real world at a relatively low cost. By producing a small scale array, measuring its performance and combining this with the theoretical analysis that we have done in the previous chapters, enough proof is gathered to come to a conclusion regarding the viability of the platform even at bigger scale.

Note that we have not been able to implement the image rectification steps in the software domain due to time restrictions. The image rectification was not deemed necessary for the performance evaluation of the prototype, and any evaluation of its effectiveness was therefore omitted from this chapter.

7.1 SENSOR ARRAY CONFIGURATION

Our first prototype has been designed to interface with up to 5 image sensors. For these image sensors, the following features were deemed necessary:

- Conventional 720p (1280 x 720) resolution.
- 60 Hz capture rate.
- MIPI CSI-2 or HiSPi standard compatibility.
- Relative low cost per unit, below $20 USD.

Today, many low cost image sensors implementing these features are produced by vendors such as Omnivision, Sony, Samsung and ON Semiconductor and are widely used in mobile phones, tablets and other embedded systems. Unfortunately, these image sensors nearly always contain a rolling shutter. The downsides of the rolling shutter have been discussed before in Chapter 2, and led us to strongly prefer the use of global shutters in this first prototype to avoid any distortion caused by the rolling shutter.

Currently, only a few image sensors exist that are low cost, implement the above features and contain a global shutter. For our prototype, we have chosen to use the Aptina MT9M021, a HiSPI compatible BGA sensor from Aptina Imaging Corporation.

Although our choice of sensor would logically require the use of the HiSPi protocol in our system, we would like to note that a realization using MIPI CSI-2 would be very similar, despite the fact that the MIPI CSI-2 specification would initially seem to require support of both the High Speed (HS) and Low Power (LP) modes on the physical lanes.

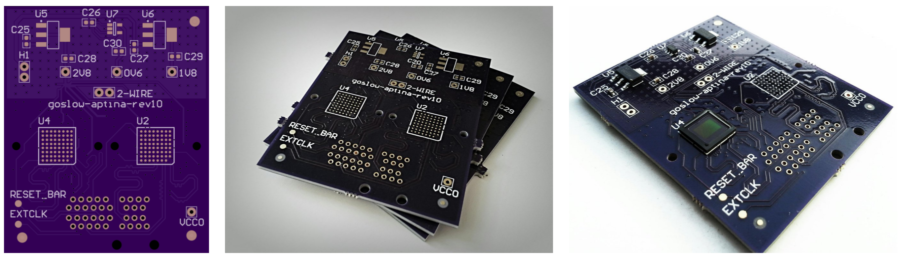
As noted before, fully supporting MIPI CSI-2's LP mode assumes that the receiver is capable of receiving both single-ended and LVDS signals on the same wires or otherwise using split I/O pins, which would lead to considerable extra (physical) complexity in our system. Whereas HiSPi omits single-ended signaling entirely, and is thus fully supported on our system, CSI-2 can actually be supported by implementing a receiver that is not fully compatible but is otherwise capable of receiving all necessary protocol packets. This is done by purposely omitting the LP mode signals and only connecting to the LVDS signals in HS mode. Additionally, some sensors may also provide the option to permanently stay in HS mode as to accommodate this behaviour on the transmitter side [Xil14].

7.2 CHOICE OF HARDWARE

We have chosen for a hardware board containing a Xilinx Spartan 6 FPGA, based on a number of reasons. First of all, our choice of sensor dictates the use of the HiSPi standard, requiring a FPGA that can provide differential I/O signaling according to this standard. HiSPi uses the SubLVDS signaling standard, limiting the choice of FPGA to those that support SubLVDS. These include the Xilinx 7-series FPGAs such as the Artix and Zynq families, as well as some older series including the Spartan 6.

Secondly, the differential I/O and SERDES functionality supported by the FPGA must have enough available bandwidth to support at least 5 sensors streaming at 720p with a rate of 60 Hz.

Thirdly, the inclusion of the DRAM controller in our design requires the presence of a physical DDR3 DIMM interface. Modern FPGA boards typically contain one or more DDR3 devices and FPGA vendors often supply proprietary DRAM IP primitives to interface with these devices. It is fairly uncommon to implement a custom DRAM controller, especially since the supplied IP typically provides enough support for conventional memory needs. It is therefore equally



(a) CAD design. (b) Post production. (c) Post pick and place.

Figure 30: Stages of development for the camera module PCB, from CAD design to production to pick and placing of SMD and BGA components.

uncommon to find a FPGA board that contains a DDR3 DIMM interface considering that this interface requires up to 240 I/O pins to be routed to the FPGA.

Our hardware board was provided by Silicon On Inspiration and contains a XC6SLX25 Xilinx Spartan 6 FPGA in FGG484 package and -2 speed grade along with a fully routed DDR3 DIMM socket and a number of expansion slots supporting differential signaling. The DDR3 DIMM socket allows us to physically interface the DRAM controller with the external DRAM devices, and the extension slots enabled a modular approach to interface the image sensors to the FPGA.

For this purpose, three custom PCBs were specifically designed and assembled to contain up to 2 of our BGA image sensors per board along with the appropriate SMD power and decoupling components. The assembled PCBs were then connected to the main FPGA board, realizing the interface between the image sensors and the hardware domain. Figure 30 shows the different phases of PCB production along with the final product.

For our embedded control processor, we have chosen to use the latest Microblaze IP soft-core from Xilinx. In the configuration of the Microblaze, we have enabled the use of Local Memory Bus (LMB) for internal core memory to enforce single-cycle latency for any instruction and stack accesses in our embedded control program. Furthermore, the core was configured without instruction and data caches, and with a smaller 3-stage pipeline by enabling area optimization to ensure a better latency consistency across instructions.

The final prototype can be seen in Figure 31 and shows the assembled PCBs containing the 5 image sensors and optical lenses along with the FPGA board fitted with a 2 GiB Corsair DDR3-1033 UDIMM module, making up the complete prototype product.

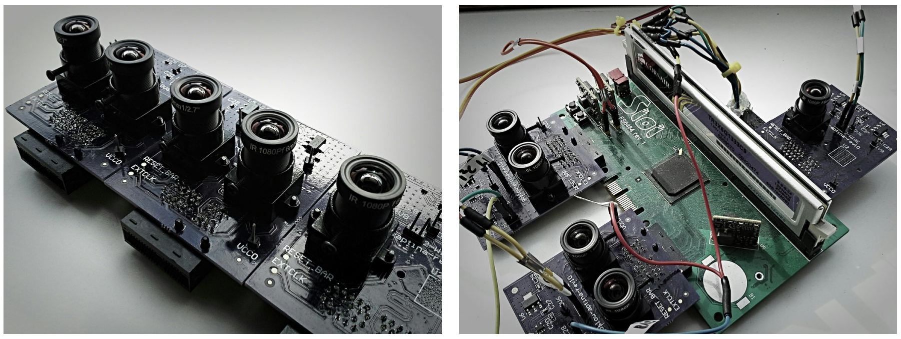


Figure 31: Final prototype hardware, showing the five individual sensor modules (left) and the final setup as evaluated with the modules and FPGA hardware connected (right).

7.3 synthesis results

Synthesis of all hardware domain subsystems has been accomplished by using Xilinx ISE 14.7 configured for the Xilinx Spartan 6 (XC6SLX25) FPGA target device present in the prototype. For these devices, FPGA device resource utilization is quantified in terms of slices, slice registers, and vendor-specific primitives including LUTs, LUTRAMs, BRAMs and such.

Table 1 shows the map synthesis results for our VHDL/Verilog implementation of the entire hardware domain. Note that in this implementation, two i2c_master entities are present instead of one due to a synthesis error, and the arbiter has been implemented as a separate entity from the actual DRAM controller logic in drac, though none of this affects the actual performance or requirements of our system.

In this table, it can be seen that the Microblaze entity takes up the most resources in terms of slices, LUTs, as well as LUTRAM and BRAM memory primitives. This is makes sense, as the softcore processor is generally a quite large general purpose core as has been discussed before. The DRAM controller takes up significant slice register resources, which is only logical due to the wide data bus of 512 bits, and the high amount of required I/O signaling to and from the DIMM device and any associated pipelining logic necessary to meet Xilinx ISE timing constraints. The stream interleaver also uses a considerable amount of resources due to the buffering action. The map report shows that the FIFO here has been modelled as a set of primitives consisting of LUTs and/or registers instead of any memory primitives such as LUTRAM or BRAM. This is by design, as the stream interleaver was modelled as a large shift register. Finally, the sensor receiver, control and readout interfaces take up only small to moderate amounts of FPGA resources, and these resources also vary somewhat depending on their exact location in the FPGA. Here, the rx_data entity that contains the cross-clock FIFO has been fully im-

Entity	Slices	Slice Reg	LUTs	LUTRAM	BRAM	BUFG	PLL
Sensor receiver interface							
rx_clk	0	0	0	0	0	1	0
rx_data*	13-24	18-28	31-43	0	0	0	0
Sensor control interface							
i2c_master*	36-44	61-94	81-119	0	0	0	0
i2c_muxer	3	0	9	0	0	0	0
Streaming DRAM controller							
arbiter	366	1134	430	0	0	0	0
drac	427	2078	774	129	0	4	1
Stream interleaver							
interleaver	353	1548	726	0	0	0	0
Embedded control processor							
microblaze	713	851	830	119	32	1	0
Readout interface							
spi_readout	180	561	377	0	0	0	0
spi_slave	15	30	28	1	0	0	0

Table 1: Map results generated using Xilinx ISE 14.7 for each of the entities in the final VHDL/Verilog implementation. Entities denoted with * represent multiple instances, for which the minimum and maximum results are specified.

plemented using LUTs and registers instead of LUTRAM or BRAM memory primitives, as the former do not provide support for multiple clocks.

Table 2 shows the place and route results in Xilinx ISE 14.7 for the total implementation, and includes all previously described entities as well as a few non-essential auxiliary entities that have been used to aid development. The results show that the overall utilization for this specific target device is generally low, e.g. virtually half of the device is left unused. Exceptions are the use of RAM primitives, as used by the Microblaze core, but these resources do not change whenever the amount of sensors in our system is changed.

Note that these results show that, first of all, there is a clear limitation on the use of primitives related to IOBs and (differential) I/O signaling, most notably ISERDES2 and IODELAY2 which are used for the sensor receiver interfaces, despite the fact that even this package has several hundred of these available. Secondly, the number of BUFG primitives is limited: in our package, only 16 of these are available. Considering a single rx_clk instance uses a single BUFG, this would imply a maximum of 16 sensors for this particular Spartan 6 device. However, a single rx_clk can in fact be used to interface with multiple sensors, if one takes care to ensure that all of these sensors are driven by same input clock such that their internal PLLs are in sync, and that all of the differential sensor data lines are length matched

Resource	Total usage	Total available	Utilization
Slice Registers	7186	30064	23%
Slice LUT logic	3276	15032	27%
Slice LUT memory	249	3664	6%
Bonded IOB	155	266	58%
PLL_ADV	1	2	50%
BUFPLL	4	8	50%
OLOGIC2/OSERDES2	125	272	45%
IODELAY2	108	272	29%
ILOGIC2/ISERDES2	81	272	29%
DCM/DCM_CLKGEN	1	4	25%
BUFG	7	16	43%
BUFIO2/BUFIO2_2CLK	4	32	12%
RAMB16BWER	32	52	61%
RAMB8BWER	4	104	3%

Table 2: Place and Route results generated using Xilinx ISE 14.7 for the total FPGA implementation, including all entities in our design and non-essential auxiliary entities for development purposes.

on the PCB, such that all resulting data lines are in fact in sync and effectively clocked by the same single differential clock. For our prototype, this strategy was used and only two rx_clk entities were used to successfully clock four sensors.

The exact amount of these primitives depends on the package of the target device: larger Spartan 6 packages or similar devices will have many more of these primitives available.

Finally, Table 3 shows all relevant clock nets in the FPGA implementation, providing an insight in the actual present clock domains as earlier described in this thesis. Here, ext_clk is the only clock that does not directly relate to any clock domains, as it is the external crystal oscillator input that drives the internal PLL.

Recall that in Section 4.7, all primary and secondary clock domains have been described. Using Table 3, we can compare the results of the actual implementation to the earlier discussed clock domains:

- Primary clock domains
  - Sensor receiver interface
    - rx_ioclk differential I/O clocks, external frequency.
    - rx_gclk internal clocks, external frequency.
  - System logic: sensor controller, interconnect, memory controller and embedded processor
    - ck150 clock, 150 MHz frequency.
    - ck50 clock, 50 MHz frequency, Microblaze core.

Clock net	Resource	Locked	Fanout
ext_clk	BUFGMUX	No	7
ck150	BUFGMUX	No	1603
ck50	BUFGMUX	No	364
ck27	BUFGMUX	No	4
rx_gclk_br	BUFGMUX	No	106
rx_gclk_bl	BUFGMUX	No	102
drac_ck600_1	Local	-	100
drac_ck600_0	Local	-	100
drac_ck600_180_1	Local	-	22
drac_ck600_180_0	Local	-	26
spi_sck	Local	-	9
rx_ioclk_p_bl	Local	-	16
rx_ioclk_n_bl	Local	-	16
rx_ioclk_p_br	Local	-	16
rx_ioclk_n_br	Local	-	16

Table 3: Place and Route results generated using Xilinx ISE 14.7 for the total FPGA implementation, showing all relevant clock nets.

    * drac clocks, 600 MHz frequency, DRAM controller.
  - Readout interface
    * spi_sck clock, external frequency.
- Secondary clock domains
  - Sensor controller and I²C interface
    * (Unlisted), 250 KHz frequency.
  - Sensor clock and reset logic
    * ck27 clock, 25 MHz frequency.

For the primary system logic clock domain, an additional 50 MHz clock is present for the Microblaze core, as this core does not synthesize properly for our Spartan 6 target device at the system clock of 150 MHz. In order to solve this issue, a dedicated Microblaze clock was derived by simply dividing the system clock down to 50 MHz, retaining the same phase relationship and clock domain. A number of 600 MHz clocks are also available in this domain to facilitate proper signaling with the external DRAM devices on the DIMM.

Furthermore, the sensor controller and I²C interface clock domain is not visible in the results, likely due to the fact that the clock generated from these components is only in the KHz range and is therefore invisible in clock report generated by Xilinx ISE.

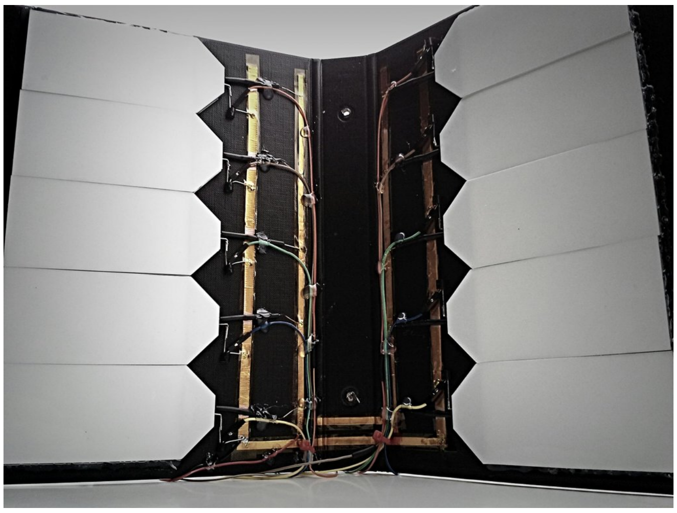


Figure 32: Stroboscope hardware, showing an array of 10 area LED lights driven by 10 MOSFETs and connected to a real-time embedded system.

7.4 MEASUREMENT SETUP

In order to provide a proper evaluation of our prototype, we require the use of additional hardware that can measure the performance aspects. First and foremost, we are interested in knowing whether the prototype can produce video at all using its image sensors at the given resolution and capture rate. Since we know from analysis that the DRAM controller is theoretically capable of interfacing with at least 38 sensors in terms of bandwidth, scalability is technically not an issue, and the use of 5 sensors should not pose any problems. However, correct and compliant functioning of the DRAM controller is especially important, and the production of a valid video stream would demonstrate a correct design and implementation of not only the DRAM controller but also all other subsystems included in our platform.

Secondly, we would like to show that the produced video in fact contains images that are to be expected from a high-speed video. Recall that a high-speed video produced with a sensor array requires a strict timing regime that imposes the exact point in time at which the sensors must start their exposure, and requires a maximum bound on shutter or exposure duration. If this timing is not accurate, a video produced using these sensors will not exhibit linear and consistent temporal effects that we expect from a video, e.g. every frame must exhibit the same lapse of time.

In order to demonstrate this, we have designed an additional piece of hardware which we will refer to as a *stroboscope*. This hardware can be seen in Figure 32 and consists of an array of 10 InGaN-based LEDs driven by MOSFETs connected to a BeagleBone Black real-time

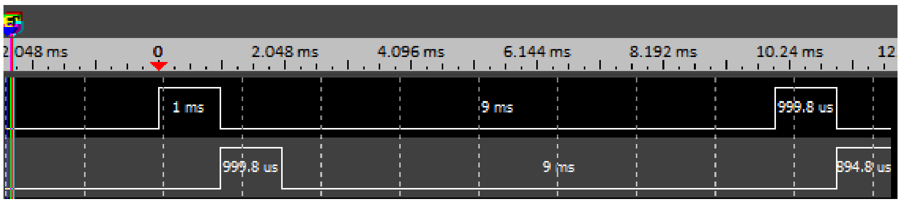


Figure 33: Timing characteristics of the stroboscope, strobing at a 1000 Hz rate, showing two LED-driving MOSFETs being subsequently triggered at 1 ms with an error of only 0.02%. Measured using a 4 GHz logic analyzer.

embedded system. This embedded system contains a Texas Instruments AM3358 Sitara ARM processor and includes a special Programmable Real-time Unit SubSystem (PRUSS) that executes single-cycle, uncached, unpipelined instructions with fully predictable and fixed instruction timing (5 ns for most instructions). The relevant real-time assembly code toggling the LEDs can be found in Appendix B.

In the stroboscope, every subsequent LED lights up for a predefined amount of time, and by pointing our sensor array prototype to this stroboscope, LEDs are then captured by the sensors. In the resulting video, temporal information can be derived from every frame in the produced video simply by looking at which LEDs are illuminated. Since the exposure duration is longer than the LED light duration, multiple LEDs are in fact captured by a single sensor frame. For the purpose of this thesis, we will fix the stroboscope LED light duration to 1 ms, so that we can easily quantify the exposure time of a particular frame by simply counting the LEDs that have lighted up. Finally, as succeeding video frames are actually captured by different image sensors, they must show a consistent lighting pattern of subsequent stroboscope LEDs, proving that the phased start functionality is accurate, as we will show in the next section.

7.5 EXPERIMENTAL RESULTS

In this section, we will discuss the experimental measurements and results related to the final prototype and its setup. We have used an Acute TL-2018E logic analyzer with a 4 GHz sample rate for any digital signal measurements.

Before proceeding with any measurements related to the actual functionality of the prototype, it is important to ensure that the stroboscope setup itself is in fact accurate. In this sense, a number of factors contribute to the accuracy of the stroboscope beyond the real-time program itself: the *slew rate* of the embedded system I/O pins, the *switching speed* of the MOSFETs driving the LEDs, and the *rise time* of the actual LEDs. Fortunately, all of these except the LED rise time can be evaluated by simply measuring the MOSFET outputs with a logic

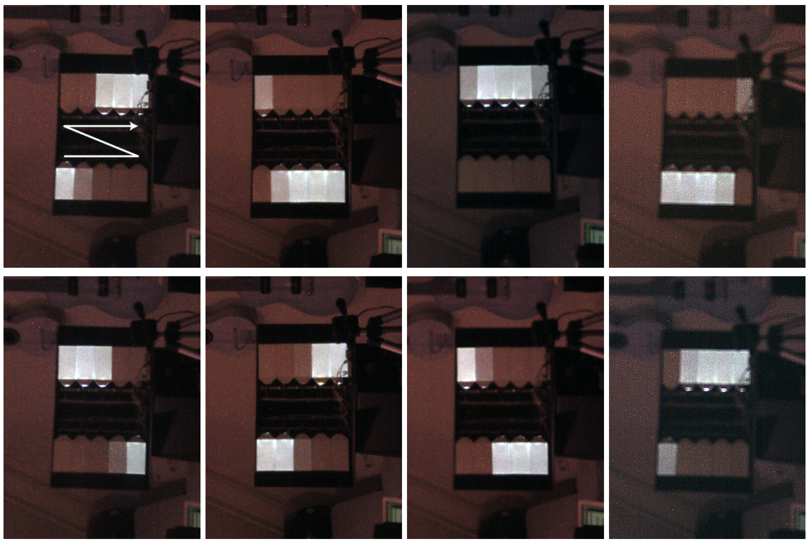


Figure 34: Final frames as captured by four sensors of the prototype at 240 Hz after being processed in the software domain, ordered from top-left to bottom-right and showing a seamlessly overlapping capture of the timed stroboscope lights from sensor to sensor. Each subsequent frame is captured by a different sensor, at different subsequent points in time. Stroboscope lights run from bottom left to top right in time, as indicated by the white arrow in the first image.

analyzer. In [Uch+00] though, it is shown that common white InGaN-based LEDs exhibit a rise time in the order of several hundreds of ns. Since this is much lower than the stroboscope's frequency order of magnitude, we consider the effect of the LED rise time to be insignificant and we can thus evaluate the accuracy of the stroboscope purely by means of the analyzer measurements. These logic analyzer measurements can be seen in Figure 33, showing two subsequent LEDs in the array being driven at virtually exact intervals and durations of 1 ms as we expected.

Knowing that the stroboscope does indeed flash at 1 ms intervals, we proceeded with producing an actual video of the stroboscope using our prototype. Unfortunately, one of the five sensors failed during this process and was no longer responsive. The prototype was then reconfigured to capture with four sensors at 60 Hz each, or a total capture rate of $4 \times 60 = 240$ Hz and a matching exposure duration of $1/240 \approx 4.2$ ms.

To produce the video, the prototype (hardware domain) was powered on and the capture phase was immediately initiated, capturing and storing 32 MiB worth of video data in DRAM, after which the readout phase was started and all video data was transferred to the software domain, in this case hosted by an embedded Raspberry Pi device, using SPI. In the software domain, the earlier described steps of deinterleaving, bitslip correction and protocol decoding were per-

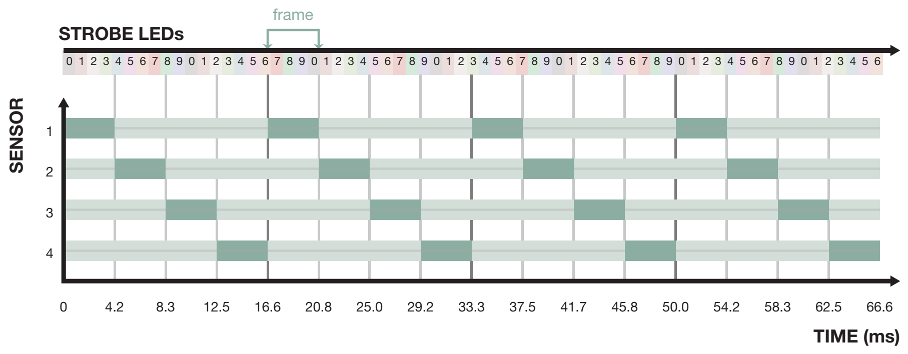


Figure 35: Sensor timing diagram corresponding to our four sensor prototype at 240 Hz and the frames in Figure 34. Our stroboscope LED timing is shown on top, with each number and color representing one of the 10 LEDs that is turned on at that time for 1 ms. Dark green represents a single frame, or the light integrated by a single sensor in the time domain for a single frame exposure, i.e. the first frame captures light from 4 different strobe LEDs. Light green represents the unavoidable time in which a sensor's shutter is closed and image data is read out to our system. When all frames are combined, they in fact form a continuous integration of physical light with an exposure time and capture rate equal to 4.2 ms or 240 Hz as witnessed in Figure 34.

formed using GStreamer, and a raw video file was ultimately produced containing all captured image frames from all subsequent sensors in order.

The individual frames of the produced video at 240 Hz can be seen in Figure 34. Note that each subsequent frame here is captured by a different sensor, at subsequent points in time, as explained before in Section 4.5, and illustrated here again in Figure 35. This figure also contains the timing of the stroboscope LED lights.

By effectively summing up each LED's physical light contribution in each single frame, we can see that the exposure duration of 4.2 ms is in fact correct: we can see approximately 4.2 LEDs being lit in each frame, when accounting for the percentage of light of each individual LED. This means that the physical light of each of these LEDs has been integrated for a duration of 4.2 ms for each individual sensor. Secondly, it can be seen that the stroboscope lights in fact follow each other precisely in each successive frame, and there are no inconsistencies or gaps in between the frames when looking at the lit LEDs. If at least one sensor would not start its exposure 4.2 ms after the previous sensor ended its exposure, e.g. due to timing inaccuracy, the LEDs in subsequent frames would not show consistent light contribution, indicating that physical light was either lost in between frames, or was captured twice by two successive sensors. Since this is not the case, we can conclude that the exposure duration as well as the phased start functionality does indeed function correctly at a rate of 240 Hz.

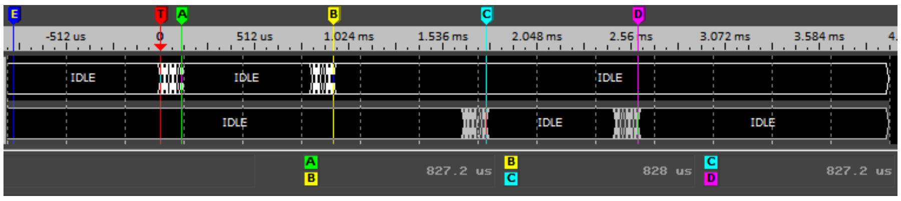


Figure 36: Timing characteristics of the I^{2}C exposure start commands as sent to different sensors in case of a hypothetical 1200 Hz (833 us) capture rate. Measured using a 4 GHz logic analyzer.

Though, recall that the second evaluation criterion requires that the timing of the phased start functionality must be at least 99%. While this criterion has been verified in theory before, measurements of the prototype should show that the criterion also holds in the real world. For this purpose, we have done a separate run in which we reconfigured our embedded control processor to send out I^{2}C phased start commands at an interval of 1200 Hz. Despite lacking the sensors to create a video at this rate, we could nevertheless measure the exact moment of time at which the I^{2}C commands appeared at the image sensors. These measurements can be seen in Figure 36 and show a maximum error ϵ of $0.8\,\text{ns}$ or less than 0.1% (of $1/1200 \approx 833\,\text{ns}$) and thus far below the 1% error tolerance imposed by evaluation criterion. Even so, the measured ϵ was higher than expected and is likely attributed to an improper implementation of the I^{2}C master driver not resetting the bus clock for every issued command.

8 CONCLUSION

The first objective of this thesis was to investigate the concept of using an image sensor array for high-speed imaging by means of an embedded system. The viability of such a sensor array setup was already described in [Wil+01], [Wil+04] and [Wil05], though at a much larger non-embedded scale. Here, various trade-offs and side effects of using an array of sensors for high-speed imaging have also been investigated.

Building upon this knowledge and explicitly targeting an embedded size factor, an embedded system platform has been designed to realize an image sensor array at a much smaller integrated form factor. Designing such a platform required a multi-domain approach, covering extensive design and implementation of subsystems in hardware as well as software. As such, the presented hardware domain includes a sensor receiver interface, dedicated DDR3 DRAM controller, interleaver, embedded processor, sensor control interface and readout interface, and functions as a standalone embedded system. Here, the presence of multiple unsynchronized clock domains required the use of cross domain clocking techniques to ensure signal integrity. The DRAM controller has been specifically designed with linear video streaming in mind, and has made use of the burst writing capabilities of DDR3 in order to maximize throughput. The embedded system has been designed to communicate individual sensor commands using the sensor control interface and to implement auxiliary functions to control the flow of the entire system. Finally, the readout interface has been included to provide a means of streaming any captured raw sensor data into the software domain.

The software domain, on the other hand, exists as a means of transforming any raw data from the hardware domain into usable information such as a video stream, and includes a sensor protocol decoder, image rectification and camera calibration algorithms as part of a video streaming pipeline. The sensor protocol decoder has been designed to decode raw HiSPI protocol data and perform debayer color reconstruction, such that a useable video data can be produced. The image rectification and camera calibration algorithms are then used to mitigate the distortion effects as introduced by the physical distance and characteristic differences of the sensors in the array.

To determine the scalability of the platform in terms of number of image sensors in the array, primary bottlenecks of the system have been identified, modelled as part of a SADF graph and analyzed using real-time dataflow analysis. This resulted in a worst-case estimation of the maximum sustained throughput of the primary bottleneck in the system, being the DRAM controller. The estimation indicated that up to 38 sensors could theoretically be interfaced with the DRAM controller, though not taking into account physical and resource limitations.

The highlight of this thesis is the actual real-world realization of the presented platform. A custom made embedded system prototype has been produced, fully implementing the hardware domain and consisting of a custom FPGA board containing a Xilinx Spartan 6 and a 2 GiB DDR3 DIMM module. Three custom made camera boards have been produced, containing a total of five Aptina MT9M021 image sensors that plug into the main FPGA board, representing a small-scale version of the sensor array and implementing the presented platform. Sensors with global shutters have been chosen to prevent the distortion effects of a rolling shutter and to reduce the required complexity of the rectification algorithms in the software domain.

To verify the high-speed capabilities of the prototype, a custom stroboscope setup has been produced consisting of 10 area LEDs driven by an independent real-time embedded system. Signal timing measurements have been performed using a high frequency logic analyzer to verify that the stroboscope setup was accurate.

Signal timing measurements have also been done on the prototype to verify accurate timing of the I^2C sensor control interface commands. Finally, a video capture of the stroboscope using four sensors at 60 Hz has been performed, resulting in a video with a combined capture rate of 240 Hz. This capture rate in this video has been verified by careful examination of the video data.

We have set out two evaluation criteria to verify whether the presented embedded system platform would be a viable solution for high-speed imaging using an array of image sensors. Both of these evaluation criteria have been satisfied, not only from a theoretical stance as proven by means of dataflow analysis, but also from a practical perspective, as has been demonstrated by our prototype albeit with a small amount of sensors. Theoretical dataflow analysis of the system was made possible by first modelling our system as a FSM-SADF graph, approximating the dataflow of our system in terms of throughput. Problems have been encountered during dataflow analysis related to the existing FSM-SADF and similar graph models and software tools in particular. Limitations of model expressiveness led to a model that did resemble the behaviour of our DRAM controller, but not in an exact manner. Despite these problems however, it was possible to work around these issues and generate results to show the viability of our platform. This model indicates that our DRAM controller is capable of providing a maximum sustained throughput of 4.23 GiB/s, corresponding to 90% of the theoretical peak transfer rate of our DDR3 devices. Our system is therefore capable of interfacing with at least 16, and possibly up to 38 image sensors at a capture rate of 60 Hz using its current DRAM configuration. Further analysis was done by constructing a simplified latency-rate SRDF graph modelling the FIFO buffering mechanism as part of the system's dataflow. Using variables associated with our real-world prototype, and by using the earlier criterion target of 16 image sensors, it was possible to work out a required FIFO size of 12. This relatively small FIFO size shows that FIFOs are not a significant factor in terms of FPGA resources required in order to implement our system. We can therefore conclude that our platform is indeed viable.

8.1 FUTURE WORK

Although the DRAM controller presented in this thesis has been shown to be capable of relatively high sustained throughput, we would also like to investigate the use of dual- or possibly quad-channel DRAM configurations that could effectively double or quadruple the throughput. This would potentially allow for even larger sensor arrays, if suitable embedded systems hardware could be found to interface with these components. Furthermore, the use of more modern DRAM technology such as DDR4 or possibly different storage media such as SATA or PCIe SSDs could open up other potential benefits for our platform.

As far as the dataflow analysis is concerned, the expressiveness of the FSM-SADF model and associated tools unfortunately prevented one-to-one modelling of our DRAM controller functionality, leading to an unnecessarily convoluted model. For one, issues related to state-space explosion due to the large number of cycles involved in modelling the DRAM controller behaviour prevent reasonable and straightforward analysis. In the future, we would like to see a type of model and software tools that would allow a one-to-one mapping of our implementation to allow for a more exact analysis. This not only holds for FSM-SADF, as current TA model tools also lack proper dataflow analysis functionality. These tools could be extended to include such functionality, such that TA models could in fact be used to model critical parts of our system. Either would benefit the practical use of theoretical dataflow analysis for real-world hardware.

We would also like to improve our current design prototype by reducing its form factor even further while employing a larger array of up to 16 sensors, using a fully standalone single-board computer design containing a more modern FPGA such as the Xilinx Artix family. This would allow for a capture rate of 960 Hz, comparable to other commercially available high-speed cameras in the lower end spectrum, and without the need of any external hardware, such that further evaluations of the produced high-speed video could be investigated.

Despite the fact that we have described the image distortion effects involved with using a sensor array and provided insight in how to mitigate these in our software domain, we have not been able to implement any of the OpenCV algorithms into the video streaming pipeline of the software domain due to timing constraints. As these algorithms are widely used in the industry and are known to perform quite well, the choice was made to skip the implementation as it would not have contributed to the actual verification of the embedded system platform as outlined in this thesis. In a future iteration of the embedded system platform we would like to integrate these algorithms into the video streaming pipeline.

Though we have presented a platform to make use of image sensor arrays and have covered the viability of using image sensor arrays for high-speed photography, we believe that the use of sensor arrays versus single sensor systems reaches far beyond the results or any benefits presented in this thesis so far. First of all, we have shown the inherently high amount of scalability and flexibility of a sensor array, as implementations easily vary in terms of types and amounts of sensors. This system does not only support a varying amount of sensors, but also different types of sensors, which may prove to be useful in yet to be found cases. An image sensor array could therefore provide new possibilities not only in high-speed video capture, but also for very high-resolution video capture (e.g. microscopy), plenoptic video capture (e.g. digital refocusing), full-spectrum video capture (e.g. using an array with a combination of IR, UV and no filters), low noise video capture (e.g. stochastic analysis using multiple sensors, same scenery) or other cases where a single-sensor approach would be cost-prohibitive or simply not possible, all while building upon the platform and components presented in this thesis. We would like to investigate these possibilities in the future.

Part I

APPENDIX

A SADF GRAPH LISTING

In Chapter 6, a SADF graph was introduced to perform dataflow throughput analysis using the SDF^3 toolkit.

Here, the full XML data representing this SADF graph is provided according to SDF^3's SADF XML specification.

Due to limitations of the toolkit, some token amounts had to be scaled down by a factor of 10 to avoid state space explosion. To compensate for this, executions times have also been scaled appropriately to mitigate any effects on the dataflow analysis results.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf3 type="sadf" version="1.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="uri:sadf" xsi:schemaLocation="uri:sadf
http://www.es.ele.tue.nl/sadf/sdf3-sadf.xsd">
  <sadf name="goslow">
    <structure>
      <!-- Source -->
      <kernel name="Ksrc"/>
      <!-- DRAM write / refresh logic -->
      <kernel name="Kram"/>
      <!-- DRAM row precharge logic -->
      <kernel name="Krow"/>

      <channel name="Ksrc2Kram" source="Ksrc" destination="Kram"
      type="data"/>
      <channel name="Kram2Krow" source="Kram" destination="Krow"
      type="data"/>
      <channel name="Krow2Kram" source="Krow" destination="Kram"
      type="data"/>

      <!-- DRAM refresh counter logic -->
      <kernel name="Kcnt"/>
      <!-- DRAM simplified state machine (detector) -->
      <detector name="Dref"/>
      <channel name="Kcnt2Dref" source="Kcnt" destination="Dref"
      type="data"/>
      <channel name="Dref2Kram" source="Dref" destination="Kram"
      type="control"/>
      <channel name="Dref2Kcnt" source="Dref" destination="Kcnt"
      type="control"/>
    </structure>
    <properties>
```

```
<!-- Output throughput for the source kernel after analysis
will indicate the maximum throughput that can be sustained
here -->
<kernel_properties kernel="Ksrc">
  <scenario name="SRC">
    <produce channel="Ksrc2Kram" tokens="1"/>
    <!-- Assume maximum throughput for the source -->
  </scenario>
</kernel_properties>

<kernel_properties kernel="Kram">
  <scenario name="WRITE">
    <!-- Source data is consumed, colums are produced -->
    <consume channel="Ksrc2Kram" tokens="1"/>
    <consume channel="Krow2Kram" tokens="1"/>
    <produce channel="Kram2Krow" tokens="1"/>
    <profile execution_time="2"/> <!-- DRAM write time: 2
    clocks @ 150 MHz -->
  </scenario>
  <scenario name="REFRESH">
    <!-- Nothing is done during a REFRESH scenario -->
  </scenario>
</kernel_properties>

<kernel_properties kernel="Krow">
  <scenario name="PRECHARGE">
    <consume channel="Kram2Krow" tokens="64"/>
    <produce channel="Krow2Kram" tokens="64"/>
    <!-- DRAM write recovery + precharge + active time: 14
    clocks @ 150 MHz -->
    <profile execution_time="14"/>
  </scenario>
</kernel_properties>

<!-- Buffer size should be unbounded, but SDF3 suffers from
state space explosion, so set it "big enough" (e.g. such
that the throughput does not change when increasing) -->
<channel_properties channel="Ksrc2Kram" buffer_size="100"/>

<!-- 64 columns are written before a PRECHARGE is done -->
<channel_properties channel="Krow2Kram"
number_of_initial_tokens="64"/>

<!-- REFRESH counter logic -->
<kernel_properties kernel="Kcnt">
  <scenario name="WRITE">
    <produce channel="Kcnt2Dref" tokens="1"/>
    <!-- DRAM refresh interval: 1170 / 10 clocks interval, so
    it takes 117 clocks to produce a data token -->
    <profile execution_time="117"/>
  </scenario>
  <scenario name="REFRESH">
```

```
      <produce channel="Kcnt2Dref" tokens="1"/>
      <!-- DRAM refresh time: 39 clocks / 10 = 260.13 ns / 10
      (division by 10 due to SDF3 state space explosion) -->
      <profile execution_time="4"/>
    </scenario>
  </kernel_properties>

  <detector_properties detector="Dref">
    <subscenario name="WRITE">
      <!-- After 117 clocks, produce 117 WRITE control tokens
      for Kram -->
      <!-- Move Kcnt to REFRESH scenario -->
      <consume channel="Kcnt2Dref" tokens="1"/>
      <produce channel="Dref2Kram" tokens="117" value="WRITE"/>
      <produce channel="Dref2Kcnt" tokens="1" value="REFRESH"/>
    </subscenario>
    <subscenario name="REFRESH">
      <!-- After 4 clocks, move Kram to REFRESH scenario -->
      <!-- Move Kcnt to WRITE scenario -->
      <consume channel="Kcnt2Dref" tokens="1"/>
      <produce channel="Dref2Kram" tokens="1" value="REFRESH"/>
      <produce channel="Dref2Kcnt" tokens="1" value="WRITE"/>
    </subscenario>

    <!-- Deterministic FSM: WRITE -> REFRESH, and REFRESH ->
    WRITE (transition is done when all subscenario conditions
    are met) -->
    <markov_chain initial_state="WRITE">
      <state name="WRITE" subscenario="WRITE">
        <transition destination="REFRESH"/>
      </state>
      <state name="REFRESH" subscenario="REFRESH">
        <transition destination="WRITE"/>
      </state>
    </markov_chain>
  </detector_properties>

  <!-- refresh interval time divded by 10 (normally 1170) to
  correct above refresh time division by 10 -->
  <channel_properties channel="Dref2Kram" buffer_size="117"
  initial_tokens="117 * WRITE"/>
  <channel_properties channel="Dref2Kcnt" buffer_size="1"
  initial_tokens="1 * WRITE"/>
  <channel_properties channel="Kcnt2Dref" buffer_size="117"/>
 </properties>
 </sadf>
</sdf3>
```

B STROBOSCOPE PROGRAM LISTING

In Chapter 6, a real-time stroboscope hardware setup was described which was used to evaluate the timing characteristics of the camera array prototype.

The stroboscope consists of an array of 10 area LEDs, driven by 10 MOSFETs and connected to a BeagleBone Black real-time embedded system. This embedded system contains a Texas Instruments AM3358 Sitara ARM processor and includes a special Programmable Real-time Unit SubSystem (PRUSS) that executes single-cycle, uncached, unpipelined instructions with fully predictable and fixed instruction timing (5 ns for most instructions), also described in [Kim+15].

To ensure accurate LED trigger times, a real-time PRU program was created in the TI PRUSS assembly language, exhibiting strict timing and triggering the MOSFETs through dedicated I/O pins. The assembly code listing can be seen below.

```
// *
// * PRU_strobo.p
// *
#define LEDS   10
#define SLEEP_HZ  2

#define R30_LED_CARRY  (LEDS)
#define R30_LED_BITS  ((1 << (LEDS)) - 1)

#define LED_PREPARE_CYCLES 5
#define SLEEP_LOOP_CYCLES 50
#define CLOCK   200000000 // 200 MHz
#define SLEEP_CYCLES  (((CLOCK / SLEEP_LOOP_CYCLES) / SLEEP_HZ)
- LED_PREPARE_CYCLES)

.origin 0
.entrypoint START

START:
      // Reset R30 (output)
      MOV r30, 0

      // Initialize registers
      MOV r2, 1
      MOV r4, R30_LED_BITS

LED_CYCLE:
      // Move shift register to R30 (output)
      MOV r30, r2

      // Busy-loop sleep routine
      MOV r1, SLEEP_CYCLES
SLEEP_LOOP:
      SUB r1, r1, 1
      QBNE SLEEP_LOOP, r1, 0

LED_PREPARE:
      // Left-shift shift register by 1
      lsl r2, r2, 1

      // Carry (N+1)th bit back to LSB
      lsr r3, r2, R30_LED_CARRY
      and r2, r2, r4
      or r2, r2, r3

      // Loop endlessly
      JMP LED_CYCLE

DONE:
      // Reset R30 (output)
      MOV r30, 0

      // Halt
      HALT
```

BIBLIOGRAPHY

[AG11] Benny Akesson and Kees Goossens. *Memory Controllers for Real-Time Embedded Systems: Predictable and Composable Real-Time Systems*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 144198206X, 9781441982063.

[Alt15a] Altera Corporation. *External Memory Interface Handbook Volume 1: Altera Memory Solution Overview and Design Flow*. 2015.

[Alt15b] Altera Corporation. *Nios II Classic Processor Reference Guide*. 2015.

[Apt12a] Aptina Imaging Corporation. *Global Shutter Pixel Technologies and CMOS Image Sensors - A Powerful Combination*. 2012.

[Apt12b] Aptina Imaging Corporation. *MT9M021/MT9M031 Developer Guide*. 2012.

[Axe+14] Philip Axer et al. "Building Timing Predictable Embedded Systems." In: *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014), 82:1–82:37. ISSN: 1539-9087. DOI: 10.1145/2560033. URL: http://doi.acm.org/10.1145/2560033.

[Bac+01] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity: an algebra for discrete event systems*. 2001.

[Beh+06] G. Behrmann, A. David, K. G. Larsen, J. Hakansson, P. Petterson, Wang Yi, and M. Hendriks. "UPPAAL 4.0." In: *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06)*. 2006, pp. 125–126. DOI: 10.1109/QEST.2006.59.

[Big+06] M. Bigas, E. Cabruja, J. Forest, and J. Salvi. "Review of {CMOS} image sensors." In: *Microelectronics Journal* 37.5 (2006), pp. 433 –451. ISSN: 0026-2692.

[BS14] Gregor Bloch and Thomas Sattelmayer. "Effects of turbulence and secondary flows on subcooled flow boiling." In: *Heat and Mass Transfer* 50.3 (2014), pp. 427–435. ISSN: 1432-1181. DOI: 10.1007/s00231-014-1301-9. URL: http://dx.doi.org/10.1007/s00231-014-1301-9.

[Bou08] J. Y. Bouguet. *Camera calibration toolbox for Matlab*. 2008. URL: http://www.vision.caltech.edu/bouguetj/calib_doc/.

[But11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. 3rd. Springer Publishing Company, Incorporated, 2011. ISBN: 1461406757, 9781461406754.

[Dor+09] Taho Dorta, Jaime Jiménez, José Luis Martín, Unai Bidarte, and Armando Astarloa. "Overview of FPGA-Based Multiprocessor Systems." In: *Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs*. RECONFIG '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 273–278. ISBN: 978-0-7695-3917-1. DOI: 10.1109/ReConFig.2009.15. URL: http://dx.doi.org/10.1109/ReConFig.2009.15.

[EGE05] A. El Gamal and H. Eltoukhy. "CMOS image sensors." In: *Circuits and Devices Magazine, IEEE* 21.3 (2005), pp. 6–20. ISSN: 8755-3996. DOI: 10.1109/MCD.2005.1438751.

[Fol+94] James D. Foley, Richard L. Phillips, John F. Hughes, Andries van Dam, and Steven K. Feiner. *Introduction to Computer Graphics*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN: 0201609215.

[Fos97] E.R. Fossum. "CMOS image sensors: electronic camera-on-a-chip." In: *Electron Devices, IEEE Transactions on* 44.10 (1997), pp. 1689–1698. ISSN: 0018-9383. DOI: 10.1109/16.628824.

[Fur+07] M. Furuta, Y. Nishikawa, T. Inoue, and Shoji Kawahito. "A High-Speed, High-Sensitivity Digital CMOS Image Sensor With a Global Shutter and 12-bit Column-Parallel Cyclic A/D Converters." In: *Solid-State Circuits, IEEE Journal of* 42.4 (2007), pp. 766–774. ISSN: 0018-9200. DOI: 10.1109/JSSC.2007.891655.

[Geo07] Maria George. *Memory Interface Application Notes Overview*. Xilinx, Inc., 2007.

[Gin11] Ran Ginosar. "Metastability and Synchronizers: A Tutorial." In: *IEEE Design and Test of Computers* 28.5 (2011), pp. 23–35. ISSN: 0740-7475.

[Gol11] John Goldie. *The Many Flavors of LVDS*. Texas Instruments Inc., 2011.

[Goo+05] Kees Goossens, Om Prakash Gangwal, Jens Röover, and A.P. Niranjan. "Interconnect and Memory Organization in SOCs for Advanced Set-Top Boxes and TV." In: *Interconnect-Centric Design for Advanced SoC and NoC*. Ed. by Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch. Boston, MA: Springer US, 2005, pp. 399–423. ISBN: 978-1-4020-7836-1. DOI: 10.1007/1-4020-7836-6_15. URL: http://dx.doi.org/10.1007/1-4020-7836-6_15.

[Goo+16] S. Goossens, K. Chandrasekar, B. Akesson, and K. Goossens. "Power/Performance Trade-Offs in Real-Time SDRAM Command Scheduling." In: *IEEE Transactions on Computers* 65.6 (2016), pp. 1882–1895. ISSN: 0018-9340. DOI: 10.1109/TC.2015.2458859.

[Gra04] T. Granberg. *Handbook of Digital Techniques for High-speed Design: Design Examples, Signaling and Memory Technologies, Fiber Optics, Modeling and Simulation to Ensure Signal Integrity*. Prentice Hall Signal Integrity Library. Prentice Hall PTR, 2004. ISBN: 9780131422919.

[Gül+12] U. Gülan, B. Lüthi, M. Holzner, A. Liberzon, and W. Kinzelbach. "5th European Conference of the International Federation for Medical and Biological Engineering: 14–18 September 2011, Budapest, Hungary." In: ed. by Ákos Jobbágy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. Chap. Experimental Analysis of the Lagrangian Flow Field in an Ascending Aorta by Particle Tracking Velocimetry, pp. 595–598. ISBN: 978-3-642-23508-5. DOI: 10.1007/978-3-642-23508-5_154. URL: http://dx.doi.org/10.1007/978-3-642-23508-5_154.

[HS88] Chris Harris and Mike Stephens. "A combined corner and edge detector." In: *In Proc. of Fourth Alvey Vision Conference*. 1988, pp. 147–151.

[HZ04] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, ISBN: 0521540518, 2004.

[JED08] JEDEC Solid State Technology Association. *JEDEC Standard, DDR3 SDRAM, JESD79-3C*. 2008.

[JNW07] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007. ISBN: 0123797519, 9780123797513.

[Kim+15] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh. "A predictable and command-level priority-based DRAM controller for mixed-criticality systems." In: *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 2015, pp. 317–326. DOI: 10.1109/RTAS.2015.7108455.

[Kle07] Dean A. Klein. "The Future of Memory and Storage: Closing the Gaps." In: *Microsoft WinHEC 2007* (2007).

[KL10] Tamar Kranenburg and Rene van Leuken. "MB-LITE: A Robust, Light-weight Soft-core Implementation of the MicroBlaze Architecture." In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '10. Dresden, Germany: European Design and Automation Association, 2010, pp. 997–1000. ISBN: 978-3-9810801-6-2. URL: http://dl.acm.org/citation.cfm?id=1870926.1871169.

[Lat15] Lattice Semiconductor Corporation. *DDR & DDR2 SDRAM Controller for MachXO2 PLD Family IP Cores User Guide*. 2015.

[LBW09] D. Lee, S. S. Bhattacharyya, and W. Wolf. "High-Performance Buffer Mapping to Exploit DRAM Concurrency in Multiprocessor DSP Systems." In: *2009 IEEE/IFIP International Symposium on Rapid System Prototyping*. 2009, pp. 137–144. DOI: 10.1109/RSP.2009.34.

[LM87] E. A. Lee and D. G. Messerschmitt. "Synchronous data flow." In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876.

[Lee09] F. Leens. "An introduction to I2C and SPI protocols." In: *Instrumentation Measurement Magazine, IEEE* 12.1 (2009), pp. 8–13. ISSN: 1094-6969. DOI: 10.1109/MIM.2009.4762946.

[LGZ08] *Image demosaicing: a systematic survey*. Vol. 6822. 2008, 68221J–68221J–15. DOI: 10.1117/12.766768.

[Li+16] Y. Li, B. Akesson, K. Lampka, and K. Goossens. "Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers." In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2016, pp. 1–12. DOI: 10.1109/RTAS.2016.7461341.

[Lim+10] Kyusam Lim, Gye Su Kim, Suki Kim, and Kwang-Hyun Baek. "A multi-lane MIPI CSI receiver for mobile camera applications." In: *Consumer Electronics, IEEE Transactions on* 56.3 (2010), pp. 1185–1190. ISSN: 0098-3063. DOI: 10.1109/TCE.2010.5606244.

[MIP16a] MIPI Alliance, Inc. *Camera Interface Specifications*. 2016. URL: http://mipi.org/specifications/camera-interface.

[MIP16b] MIPI Alliance, Inc. *Frequently Asked Questions*. 2016. URL: http://mipi.org/about-mipi/frequently-asked-questions.

[Mor78] Jorge Moré. "The Levenberg-Marquardt algorithm: Implementation and theory." In: *Numerical Analysis*. Ed. by G. A. Watson. Vol. 630. Lecture Notes in Mathematics. Springer Berlin Heidelberg, 1978. Chap. 10, pp. 105–116. ISBN: 978-3-540-08538-6. DOI: 10.1007/bfb0067700.

[Per15] T. S. Perry. "Sanstreak lowers the cost of high-speed photography [Resources$_s$tartups]." In: *IEEE Spectrum* 52.3 (2015), pp. 27–27. ISSN: 0018-9235. DOI: 10.1109/MSPEC.2015.7049433.

[Phi03] Philips Semiconductors. *The I2C-bus specification*. 2003.

[Ram08] Chitra Ramalingam. "Stopping time: Henry Fox Talbot and the origins of freeze-frame photography." In: *Endeavour* 32.3 (2008), pp. 86 –93. ISSN: 0160-9327.

[Sem15a] Semiconductor Components Industries, LLC. *AR0141CS Datasheet, Rev. D - 1/4-Inch Digital Image Sensor*. 2015.

[Sem15b] Semiconductor Components Industries, LLC. *AR0330 Datasheet, Rev. U - 1/3-Inch CMOS Digital Image Sensor*. 2015.

[Siy+11] F. Siyoum, M. Geilen, O. Moreira, R. Nas, and H. Corporaal. "Analyzing synchronous dataflow scenarios for dynamic software-defined radio applications." In: *System on Chip (SoC), 2011 International Symposium on*. 2011, pp. 14–21. DOI: 10.1109/ISSOC.2011.6089222.

[Son11] Sony Corporation. *Advantage of the CMOS Sensor*. 2011.

[SB00] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. 1st. New York, NY, USA: Marcel Dekker, Inc., 2000. ISBN: 0824793188.

[SGB06] Sander Stuijk, Marc Geilen, and Twan Basten. "SDF^3 : SDFForFree." In: *2010 10th International Conference on Application of Concurrency to System Design* 0 (2006), pp. 276–278. ISSN: 1550-4808. DOI: http://doi.ieeecomputersociety.org/10.1109/ACSD.2006.23.

[TAK06] J. G. Tong, I. D. L. Anderson, and M. A. S. Khalid. "Soft-Core Processors for Embedded Systems." In: *2006 International Conference on Microelectronics*. 2006, pp. 170–173. DOI: 10.1109/ICM.2006.373294.

[Uch+00] Yuji Uchida, Tatsumi Setomoto, Tsunemasa Taguchi, Yoshinori Nakagawa, and Kazuto Miyazaki. *Characteristics of high-efficiency InGaN-based white LED lighting*. 2000. DOI: 10.1117/12.389397. URL: http://dx.doi.org/10.1117/12.389397.

[VM11] Michael Vollmer and Klaus-Peter Möllmann. "High speed and slow motion: the technology of modern high speed cameras." In: *Physics Education* 46.2 (2011), p. 191.

[WBS07] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. "Modelling Run-time Arbitration by Latency-rate Servers in Dataflow Graphs." In: *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*. SCOPES '07. Nice, France: ACM, 2007, pp. 11–22. DOI: 10.1145/1269843.1269846. URL: http://doi.acm.org/10.1145/1269843.1269846.

[Wil+04] B. Wilburn, N. Joshi, V. Vaish, M. Levoy, and M. Horowitz. "High-speed videography using a dense camera array." In: *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*. Vol. 2. 2004, II–294–II–301 Vol.2. DOI: 10.1109/CVPR.2004.1315176.

[Wil+01] Bennett S. Wilburn, Michal Smulski, Hsiao-Heng K. Lee, and Mark A. Horowitz. *Light field video camera*. 2001. DOI: 10.1117/12.451074. URL: http://dx.doi.org/10.1117/12.451074.

[Wil05] Bennett Wilburn. "High-performance Imaging Using Arrays of Inexpensive Cameras." AAI3153705. PhD thesis. Stanford, CA, USA, 2005. ISBN: 0-496-13789-1.

[Wil+05] Bennett Wilburn, Neel Joshi, Vaibhav Vaish, Eino-Ville Talvala, Emilio Antunez, Adam Barth, Andrew Adams, Mark Horowitz, and Marc Levoy. "High Performance Imaging Using Large Camera Arrays." In: *ACM SIGGRAPH 2005 Papers*. SIGGRAPH '05. Los Angeles, California: ACM, 2005, pp. 765–776. DOI: 10.1145/1186822.1073259. URL: http://doi.acm.org/10.1145/1186822.1073259.

[WM15] Xu Wu and Guy Meynants. *High speed global shutter image sensors for professional applications*. 2015. DOI: 10.1117/12.2179227. URL: http://dx.doi.org/10.1117/12.2179227.

[Xil08] Xilinx, Inc. *UG081 (v9.0) - MicroBlaze Processor Reference Guide*. 2008.

[Xil14] Xilinx, Inc. *XAPP864 - D-PHY Solutions*. 2014.

[Zha99] Zhengyou Zhang. "Flexible camera calibration by viewing a plane from unknown orientations." In: *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*. Vol. 1. 1999, 666–673 vol.1. DOI: 10.1109/ICCV.1999.791289.

[Zha00] Zhengyou Zhang. "A Flexible New Technique for Camera Calibration." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22.11 (Nov. 2000), pp. 1330–1334. ISSN: 0162-8828. DOI: 10.1109/34.888718.