VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY

HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY

FACULTY OF COMPUTER SCIENCE AND ENGINEERING

REPORT
CAPSTONE PROJECT

HOG-BASED REAL TIME HUMAN DETECTION ASIC FOR SMART VIDEO SURVEILLANCE AT THE EDGE

MAJOR: COMPUTER ENGINEERING

COMMITTEE : 06
SUPERVISOR : PHAM QUOC CUONG, Assoc. Prof. Dr.
REVIEWER : HUYNH PHUC NGHI

————o0o————

STUDENTS : PHAN DUC DAT - 2113152
: TO HOANG PHONG - 2112012
: TRAN ANH TAI - 2114700

HO CHI MINH CITY, May 2025

HĐ6 KTMT - 154

ĐẠI HỌC QUỐC GIA TP.HCM
TRƯỜNG ĐẠI HỌC BÁCH KHOA

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

KHOA:KH & KT MÁY TÍNH
BỘ MÔN: KỸ THUẬT MÁY TÍNH

NHIỆM VỤ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

Chú ý: Sinh viên phải dán tờ này vào trang nhất của bản thuyết trình

HỌ VÀ TÊN: Tô Hoàng Phong — MSSV: 2112012
HỌ VÀ TÊN: Phan Đức Đạt — MSSV: 2113152
HỌ VÀ TÊN: Trần Anh Tài — MSSV: 2114700
NGÀNH: Kỹ thuật Máy tính — LỚP: MT21KTM

1. Đầu đề luận văn/ đồ án tốt nghiệp:
Thiết kế mạch ASIC cho điện toán biên phát hiện người thời gian thực dựa trên HOG *(HOG-based real time human detection ASIC for smart video surveillance at the edge.)*

2. Nhiệm vụ (yêu cầu về nội dung và số liệu ban đầu):
- Tìm hiểu, thiết kế và hiện thực các khối chức năng:
  - RV32 processor: trích xuất vùng nhớ và cấu hình các thành phần khác
  - VDMA (Video Direct Memory Access): bộ truy xuất vùng nhớ ảnh trực tiếp.
  - Frame Buffer: bộ nhớ đệm cho để lưu trữ khung hình có người
  - Accelerator: bộ gia tốc xử lý ảnh trong việc phát hiện khung hình có người
  - Camera Interface: dùng để giao tiếp với camera OV7670
- Tìm hiểu ứng dụng và tập dữ liệu đánh giá hệ thống
- Hoàn thiện lõi tính toán và hệ thống
- Thử nghiệm với tập dữ liệu và so sánh đánh giá

3. Ngày giao nhiệm vụ: 06/01/2025

4. Ngày hoàn thành nhiệm vụ: 22/5/2025

5. Họ tên giảng viên hướng dẫn: **Phần hướng dẫn:**

1) Phạm Quốc Cường

Nội dung và yêu cầu LVTN/ ĐATN đã được thông qua Bộ môn.

Ngày 06 tháng 01 năm 2025

CHỦ NHIỆM BỘ MÔN
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

GIẢNG VIÊN HƯỚNG DẪN CHÍNH
(Ký và ghi rõ họ tên)

Phạm Quốc Cường

PHẦN DÀNH CHO KHOA, BỘ MÔN:
Người duyệt (chấm sơ bộ):
Đơn vị:
Ngày bảo vệ:
Điểm tổng kết:
Nơi lưu trữ LVTN/ĐATN:

HD 6 KTMT – 154

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày 15 tháng 5 năm 2025

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người hướng dẫn)

1. HỌ VÀ TÊN: Tô Hoàng Phong MSSV: 2112012
Phan Đức Đạt MSSV: 2113152
Trần Anh Tài MSSV: 2114700
NGÀNH: Kỹ thuật Máy tính LỚP: MT21KTM

2. Đề tài:
Thiết kế mạch ASIC cho điện toán biên phát hiện người thời gian thực dựa trên HOG *(HOG-based real time human detection ASIC for smart video surveillance at the edge.)*

3. Họ tên người hướng dẫn: Phạm Quốc Cường

4. Tổng quát về bản thuyết minh:
Số trang: Số chương:
Số bảng số liệu Số hình vẽ:
Số tài liệu tham khảo: Phần mềm tính toán:
Hiện vật (sản phẩm)

5. Những ưu điểm chính của LV/ ĐATN:
- Sinh viên hoàn thành tốt nhiệm vụ đặt ra
- Hệ thống hoạt động ổn định, chính xác, đáp ứng được tiêu chí đề ra
- Sinh viên làm việc nghiêm túc, báo cáo đạt yêu cầu
- Nhóm sinh viên đã mang sản phẩm dự thi cuộc thi thiết kế vi mạch cho thành phố thông minh lần 2 do Khu Công nghệ cao TPHCM tổ chức và được vào vòng chung kết

6. Những thiếu sót chính của LV/ĐATN:
- Hệ thống thử nghiệm còn gặp một số vấn đề liên quan đến hiển thị do chất lượng camera và màn hình thử nghiệm có giá thành thấp
- Phần ứng dụng và tập dữ liệu test còn đơn giản

7. Đề nghị: Được bảo vệ ☒ Bổ sung thêm để bảo vệ ☐ Không được bảo vệ ☐
8. Các câu hỏi SV phải trả lời trước Hội đồng:
a. Đâu là rào cản cho việc gia tăng độ phân giải ảnh?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Điểm: 9.8/10

Ký tên (ghi rõ họ tên)

Phạm Quốc Cường

HĐ 6 KTMT - 154

TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KH & KT MÁY TÍNH

CỘNG HÒA XÃ HỘI CHỦ NGHĨA VIỆT NAM
Độc lập - Tự do - Hạnh phúc

Ngày tháng năm

PHIẾU ĐÁNH GIÁ LUẬN VĂN/ ĐỒ ÁN TỐT NGHIỆP

(Dành cho người hướng dẫn/phản biện)

1. Họ và tên SV: PHAN DUC DAT - 2113152
TO HOANG PHONG - 2112012
TRAN ANH TAI - 2114700

Ngành (chuyên ngành): Kĩ thuật máy tính

2. Đề tài: HOG-BASED REAL TIME HUMAN DETECTION ASIC FOR SMART VIDEO SURVEILLANCE AT THE EDGE

3. Họ tên người hướng dẫn/phản biện: Huỳnh Phúc Nghị

4. Tổng quát về bản thuyết minh:
Số trang: 86 Số chương: 7
Số bảng số liệu: 9 Số hình vẽ: 94
Số tài liệu tham khảo: 11 Phần mềm tính toán:
Hiện vật (sản phẩm)

5. Những ưu điểm chính của LV/ ĐATN:
- Nhóm đã hiện thực thiết kế bộ xử lí phát hiện người trong hình áp dụng giải thuật HOG và SVM.
- Nhóm đã đưa thiết kế từ RTL ra GDSII thông qua dự án mã nguồn mở Openlane với kết quả tổng hợp đạt Fmax=200MHz và area=30,424.1792 nm^2

6. Những thiếu sót chính của LV/ĐATN:
- Nhóm chỉ tổng hợp ra GDSII được một số giới hạn khối chức năng. Một số khối chức năng liên quan đến memory không tổng hợp được vì Openlane không hỗ trợ.
- Quá trình triển khai trên FPGA không nhất quán giữa Arty z7-20 và De10-nano. Thông số đánh giá không thống nhất (lúc có đơn vị %, lúc thì không)
- Kết quả báo cáo không đề cập tới hiệu quả về năng lượng, một yếu tố quan trọng trong ứng dụng của thiết bị biên.

7. Đề nghị: Được bảo vệ ⊗ Bổ sung thêm để bảo vệ ○ Không được bảo vệ ○

8. Các câu hỏi SV phải trả lời trước Hội đồng:
a. Đâu là yếu tố quan trọng về thiết kế dẫn đến khác biệt về kết quả Resolution của nhóm (320x240 trên De10-nano Cyclone IV) và bài báo số 6 (1920x1080 trên Cyclone V)?

9. Đánh giá chung (bằng chữ: Xuất sắc, Giỏi, Khá, TB): Xuất sắc Điểm : 9.2 /10

Ký tên (ghi rõ họ tên)

Huỳnh Phúc Nghị

This thesis is dedicated for our parents and our instructors at HCMUT.

Declaration Of Authenticity

We declare that we conducted this specialized project under the supervision of Assoc. Prof. Pham Quoc Cuong at the Faculty of Computer Science and Engineering, Vietnam National University - Ho Chi Minh City University of Technology.

We have taken care to properly acknowledge and document all external sources and references used in the project.

If there is any instance of plagiarism, we are ready to accept the consequences. Ho Chi Minh City University of Technology - Vietnam National University HCMC will not be held responsible for any copyright violations that may have occurred during my research.

Ho Chi Minh City, May 2025

Authors,

Phan Duc Dat

To Hoang Phong

Tran Anh Tai

Acknowledgment

We would like to express our appreciation to Assoc. Prof. Pham Quoc Cuong for his advices and invaluable guidance. My research has greatly benefited from his deep knowledge, perceptive criticism, and constant support.

In addition, we also would like to express our gratitude to our family and friends. Their unwavering faith in our abilities and constant encouragement have been our pillars of strength. Their belief in our potential has been a constant source of motivation and resilience. We are eternally grateful for their love and support.

Abstract

This work describes a real-time human detection application-specific integrated circuit (ASIC) with an image processor with Histogram of oriented (HOG) and Support vector machine (SVM) integration. It features a simplified HOG algorithm with cell-based scanning and simultaneous Support Vector Machine (SVM) calculation, cell-based pipeline architecture, and parallelized modules. To evaluate the effectiveness of our approach, the proposed architecture is implemented onto a FPGA prototyping board and Openlane tool. Results show that the proposed architecture can generate HOG features and detect objects with 200 MHz for resolution video of 320 × 240 pixels at 2,604 frames per second (fps).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In chapter 1, provides a brief overview of the project, including its motivation, project objectives, project scope, and outline of this project.

1.1 Motivation

Nowadays, edge devices are becoming more common in many applications. However, the amount of data they collect and send to servers is often too large for the servers to handle or store, especially as the number of edge devices increases. Edge computing is an important solution to this problem because it processes data locally, reducing the workload on servers and saving bandwidth.

Modern cameras are also considered edge devices. They can process data locally to detect motion or detect human(s) in images. However, building such cameras to work in real-time requires expensive components or high-performance embedded systems, which pose significant barriers to widespread implementation.

Thus, to have a device that can effectively serve a specific purpose, be cost-efficient, and compact, this thesis aims to design an application-specific integrated circuit (ASIC) for digital cameras used as edge devices. The ASIC can classify image frames to detect people using the Histogram of Oriented Gradients (HOG) and Support Vector Machine (SVM) algorithms. It is designed to meet real-time requirements while being programmable to adjust algorithm

settings and support compatibility with other system components.

1.2 Goals

The goals of this project are as follows:

- First is to design the system architecture and the architecture of an image processor using the HOG algorithm for feature extraction and SVM for image classification to detect human(s).
- Second is to design control components for the system to manage the data flow from the camera to the system and output to external displays. Additionally, design components that support programming and system configuration.
- Third is to implement the full system on FPGA and test it in real-world scenarios to verify its functionality.

1.3 Scope

The scope of this project includes designing, verifying the system for digital cameras using the HOG-based algorithm for human detection on FPGA boards and implementing the system down to the GDSII file.

1.4 Thesis structure

There are five chapters in this project:

- **Chapter 1 - Introduction**: briefly describes the project about the problem's motivation, as well as the project's goals and scope.
- **Chapter 2 - Theoretical background**: Provides basic information about the knowledge used in this project, such as the standardized algorithm, and standardized methodology.

- **Chapter 3 - Related works**: Proposed a brief idea for the system architecture via block diagrams and detail circuits.

- **Chapter 4 - Proposed architecture**: Proposed a brief idea for the system architecture via block diagrams and detail circuits.

- **Chapter 5 - Functional verification**: Verify a hardware module of the HOG algorithm. A Python reference model ensures correctness across all processing steps. Verify a hardware module of direct memory access module used to move mem in system. Additionally, verify system to ensure it work accurately.

- **Chapter 6 - Experimental results**: Show the experimental results of the proposed image processing system. The hardware modules were evaluated through synthesis and layout processes, where synthesis results provided data on resource usage, timing, and power estimates, while GDSII outcomes confirmed the design's physical implementation readiness. The system was also deployed on an FPGA board to validate functionality under real operating conditions, confirming that data flow and processing steps such as HOG and DMA worked correctly.

- **Chapter 7 - Conclusion**: Summarizes the design and implementation of a real-time image processing IP, highlights key performance metrics, and confirms the system's efficiency and flexibility through ASIC synthesis and FPGA prototyping.

Chapter 2

Theoretical Background

In Chapter 2, it presents preliminary knowledge in this project.

2.1 Histogram of oriented (HOG)

Histogram of oriented gradients (HOG) is feature descriptor which is primarily used in computer vision and image processing. The technique of HOG is capturing the shape or structure of an object by analyzing the distribution of gradient direction and intensity in an image. It is particularly effective in object detection such as pedestrian detection.

To understand how the HOG algorithm works, we'll look at the block diagram below: Before calculating the HOG, the input image needs to be carefully prepared to increase the processing efficiency. Firstly, if input image is color image, we need to change it to gray scale to decrease calculating complexity. Because HOG relies only on intensity, not color information. Next, image may be resized to fixed size, for example 64x128 pixels for human detection to synchronize the input image with the information trained for classification. This step prepare suitable image for the following steps.

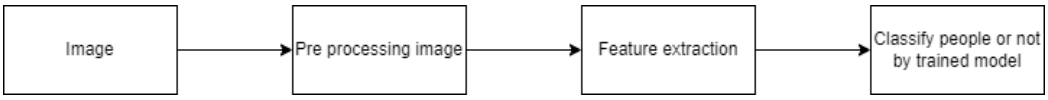


Figure 2.1: Block diagram of histogram of oriented gradients

After preprocessing image, image will come to feature extraction step. At that step, we will extract the hog feature through several steps such as gradient calculation, histogram calculation, normalization, and combine it into a feature vector. The figure 2.2 show the flow of the hog in detail.

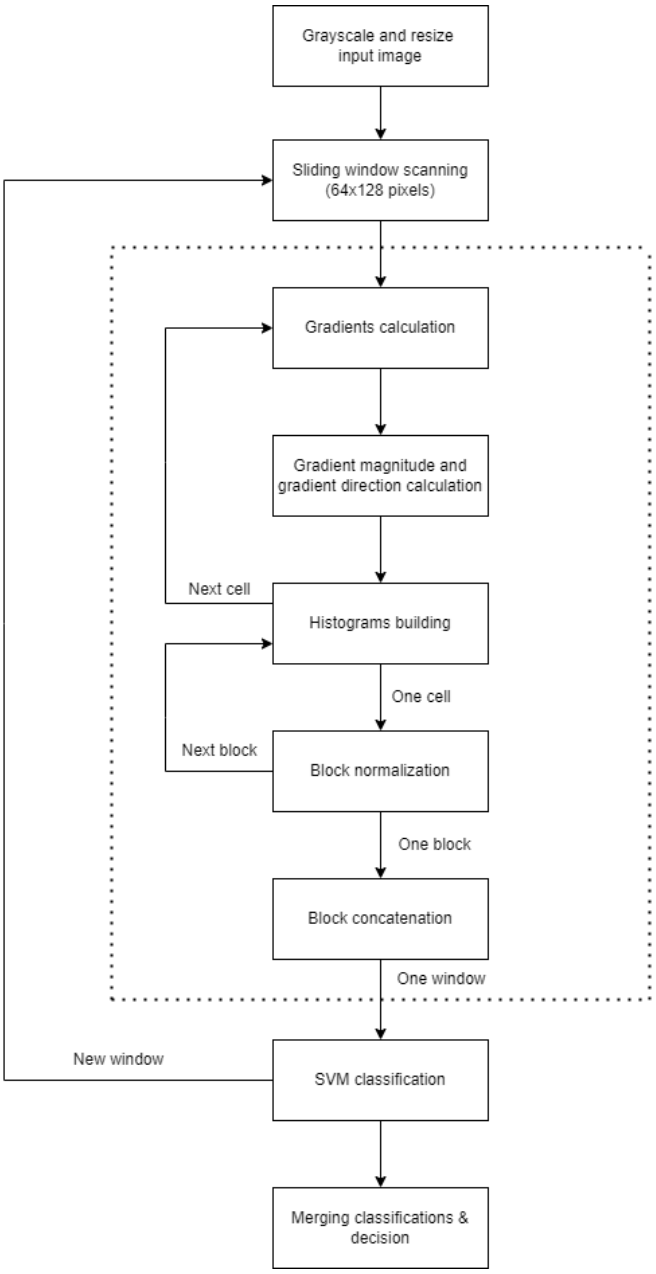


Figure 2.2: Flow histogram of oriented gradients

Firstly, in this gradient calculation step, for example image is 64x128 pixels. We divide the image into 8x8 pixels cells and perform gradient calculations on them. On 8x8 pixels, we will calculate the gradient follow x axis (horizontal

change) and y axis (vertical change). Note that:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{2.1}$$

But the smallest h of the image is 1 (1 pixel), so the formula above become:

$$f'(x) = f(x+1) - f(x) \tag{2.2}$$

In practical, formula 2.2 above called forward difference, the backward difference 2.3 and central difference 2.4 is following by formula below:

$$f'(x) = f(x) - f(x-1) \tag{2.3}$$

$$f'(x) = \frac{f(x+1) - f(x-1)}{2} \tag{2.4}$$

But we use central difference without divide 2 because it does not affects the direction of the vector, it only affect the magnitude, but we will normalize after weight bin vote, so it does not matter, and it will reduce complexity when implementing in hardware.

Let L(x, y) is intensity at coordinate (x, y), the horizontal and vertical differences of intensity $\Delta L_x(x,y)$, $\Delta L_y(x,y)$ are calculated according to the formula below:

$$\Delta L_x(x,y) = L(x+1,y) - L(x-1,y) \tag{2.5}$$

$$\Delta L_y(x,y) = L(x,y+1) - L(x,y-1) \tag{2.6}$$

After gradient calculation, We will perform gradient direction calculation, and gradient magnitude as formula below:

$$magnitude(x,y) = \sqrt{\Delta L_x(x,y)^2 + \Delta L_y(x,y)^2} \tag{2.7}$$

$$\theta(x,y) = \tan^{-1} \frac{\Delta L_y(x,y)}{\Delta L_x(x,y)} \tag{2.8}$$

The value of θ is only calculated in the range [0, 180] degrees, meaning if the θ value is greater than 180 degrees, we will add 180 degrees to make it within the allowed range (such as -90 degrees or 270 degrees it will add 180 to become 90 degrees,)

Next to, when we get gradient direction and gradient magnitude, we will do weight voting. Weight voting is done by first dividing into 9 bins each bin corresponds to 20 degrees (bin[0] corresponds to [0, 20) degrees, bin [1] corresponds to [20, 40) degrees, ..., bin[8] corresponds to [160, 180) degrees and 180 degrees will be voted into bin [0].

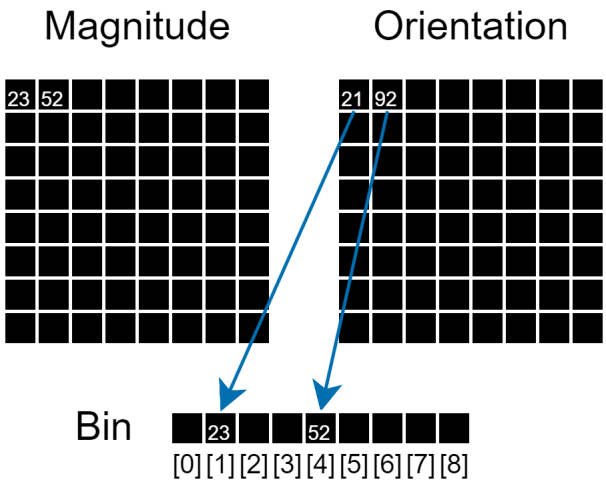


Figure 2.3: Weight voting

As image above, the first pixel of cell has magnitude is 23 and orientation is 21 (corresponds to bin[1], so bin [1] = 23), the second pixel has magnitude is 52 and orientation is 92 (corresponds to bin [4], so bin [4] = 52). Doing the same for all the pixels in the cell.

Before we get to the vector histogram and normalization step, we will discuss the following ideas. 64x128 image has 8 cells horizontally and 16 cells vertically. Block (figure 2.4) is defined as 4 adjacent cells so that it forms a square. We will calculate the bin for each cell of a block and then concatenate them together. For each bin of each cell we will crate a 1x9 vector, 4 concatenated cells will create a 1x36 vector. Then we will normalize this vector.

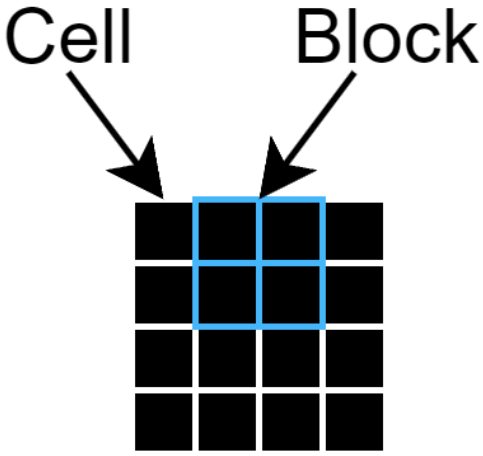


Figure 2.4: Block of image

We will calculate the bin for each cell of a block and then concatenate them together. For each bin of each cell we will have a 1x9 vector, 4 concatenated cells will have a 1x36 vector. Then we will normalize this vector

The main goal of normalizing is to make the feature extraction from the block more stable to external environmental factors such as brightness and contrast. It helps machine learning models (such as SVM) easily recognize the features of the object. Thereby, the machine learning models will give accurate predictions.

There are some popular normalization methods such as L1-norm, L2-norm. However, in this article, the group will use L1-sqrt because it still ensures accuracy while the complexity will be significantly reduced because there is no need to square. Formula of L1-sqrt is as below:

$$L1-sqrt = \sqrt{|v_1|+|v_2|+...+|v_n|} \tag{2.9}$$

Then, we calculate the gradient magnitude, gradient orientation, bin of each cell in turn and when enough to form a block, we will normalize according to L1-sqrt as formula below:

$$f_i = \sqrt{\frac{v_i}{|v_1|+|v_2|+...+|v_n|+\varepsilon}} \tag{2.10}$$

With denominator is L1-sqrt (ε avoid 0 in denominator) and numerator is value of histogram at index i.

A 64x128 pixel image will have 7 blocks horizontally and 15 blocks vertically so it will have a total of 105 blocks. Doing this in turn, we will get a 1x3780 feature vector. Then we will give this to machine learning models for classification.

2.2 Support vector machine (SVM)

Support Vector Machine (SVM) is powerful supervised learning models which is primarily used for classification and regression problems (support vector regression - SVR). For example, we want to distinguish a set of red points and a set of blue points in 2D space, SVM will help us do that - it will find a straight line to distinguish the two data sets above. In addition, SVM can also do more complex problems such as classifying data sets in multi-dimensional space.

The basic concept of SVM is finding the line in 2D space (or hyperplane in higher multi - dimensional) that separate different data sets. But there are many line or hyperplane can separate that data classes, so what is the best line or hyperplane SVM can find?

The aim of SVM is finding the line or hyperplane that separate data classes correctly and maximize the distance of nearest data point from each data classes to this line or hyper plane.

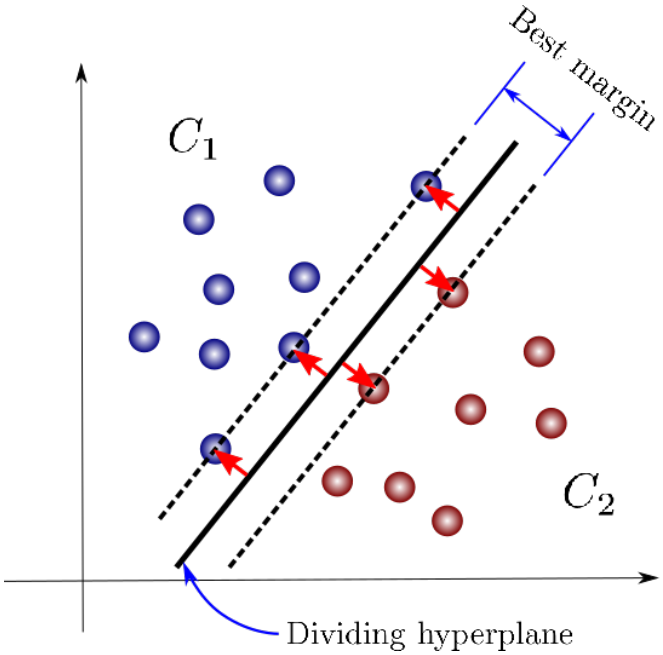


Figure 2.5: Optimal line or hyperplane quote from link

This distance is called margin, the margins of each data class must be equal

and as large as possible. The idea of maximize margin is to avoid overfitting the training dataset. This leads to the fact that when applied to real-world datasets, it will not perform well and will result in incorrect classifications.

Kernel is mathematical function in SVM, the function of kernel in transform data input to the desired form. Some popular kernel is linear kernel, non - linear kernel (polynomial, radial basis function - rbf, sigmoid). But we use linear kernel to implement in hardware base on this complexity.

In our scenario, we will train the hog model with the sk-learn library to detect people, our output will be a vector of weights ($\vec{w}$) and bias (b). To detect people we will extract the hog feature ($\vec{x}$) which is also a vector with the same size as $\vec{w}$ then calculate the following formula to detect whether there is a person or not:

$$w^T \cdot x + b \tag{2.11}$$

If the result is greater than 0 then it is above the hyperplane meaning there is a person, if it is less than 0 then it is below the hyperplane meaning there is no person.

2.3 HOG-SVM integrated system

The HOG-SVM Integrated System combines two techniques, Histogram of Oriented Gradients (HOG) and Support Vector Machine (SVM), to detect objects, especially people, in images. HOG is used to extract important features from the image, and SVM is used to classify these features into categories, such as "human" or "not human.". There are 2 steps to integrate it.

- Firstly, the system is to use HOG to extract features from the image. HOG looks for changes in brightness (gradients) in different parts of the image, which helps in identifying edges and shapes of objects. The image is divided into small blocks, and for each block, a histogram is created that shows the directions of the gradients. These histograms are then combined into one long feature vector that describes the entire image.

- Secondly, the features are extracted using HOG is used for classification with an SVM. SVM is a machine learning method that finds the best way to separate different categories ("human" or "not human") based on their features.

Totally, HOG and SVM are two powerful techniques in object recognition; when combined, they form a strong and accurate system for feature extraction and object classification in images.

2.4 Universal verification methodology (UVM)

UVM is a method of simulating hardware designs, described in a hardware specification language such as Verilog or VHDL.

UVM was born with the goal of creating a common simulation method for the IC industry. Simulation is a long and very time-consuming process, and the construction and editing of the environment for the simulation is quite time-consuming. Therefore, UVM was born to have a method, a common environment for simulation that increases reusability, and is easily scalable.

Before the birth of UVM, there were also a number of other simulation methods such as OVM (Open Verification Methodology), RVM (Reference Verification Methodology), VMM (Verification Methodology Manual),...it is created by various companies such as Synopsys, Mentor, Cadence,... however, it is still not synchronized between different organizations, so UVM was born to achieve the above goal.

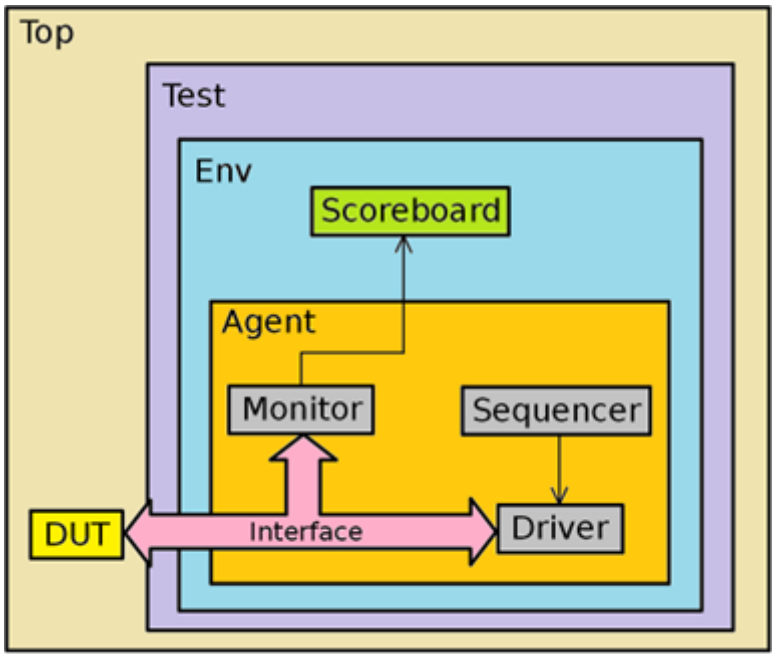


Figure 2.6: UVM enviroment quote from link

Top: This is the top level of the test environment, which includes all the components to test a DUT.

The list of components in UVM:

- Test: This is a specific class to define test scenarios. It inherits from the uvm_test class and will configure the test environment here.
- Env: The environment contains all the components to test and simulate the DUT. It usually consists of one or more agents, scoreboard.
- Agent: Is a part of the environment that is responsible for simulating communication with the DUT, it usually includes a driver, monitor and sequencer.
- Driver: Responsible for receiving sequences from the sequencer and then driving those signals to the DUT through the interface. The driver will simulate the behavior of the hardware in reality.
- Sequencer: It is responsible for providing sequence packets to the driver.
- Monitor: Monitors communication between driver and DUT, collects data and then transfers this data to the scoreboard for correctness checking.
- Scoreboard: Responsible for checking the correctness of the DUT output. It compares the obtained results with the expected results (golden model) to ensure the DUT is operating correctly.

The flow of data in the UVM environment, first the sequencer will send sequence to the driver, after the driver receives the package it will proceed to drive the data on the DUT through the interface. The monitor will simultaneously collect the necessary data and transfer it to the scoreboard. On the scoreboard we can build a golden model and compare it with the data collected from the monitor to check the correctness.

In UVM, phase refers to different stages in the test verification process. Phases in UVM help in managing and coordinating testing activities in an organized manner. Here are some basic phases in UVM:

Type phase	Phase	Type	Description
Build	build_phase	Function	Build testbench components and create instances
	connect_phase	Function	Connect testbench components via TLM port
	end_of_elaboration_phase	Function	Print out UVM topology
	start_of_simulation_phase	Function	Initialize run-time or display topology
Run	run_phase	Task	The stage where major tasks such as sending and processing signals
Clean	extract_phase	Function	Extract data and calculate desired data from scoreboard
	check_phase	Function	Check between actual data and desired data
	report_phase	Function	Generate a report
	final_phase	Function	Freeing resources

Table 2.1: Phase in UVM

2.5 Clock domain crossing (CDC)

In VLSI design CDC refer to transfer data between 2 different domains clock.

Modern VLSI systems often integrate many blocks, IPs. Each of these runs at a different optimized frequency for performance and power. Therefore, CDC design is very important to avoid potential errors and malfunctions. Without CDC can potentially lead to some issues as below:

- **Metastability:** Flip-flops can enter unstable state (unpredictable outcome values).
- **Glitches:** Asynchronous transitions can create short-lived spurious changes in signal values, lead to unwanted logic.
- **Data loss:** Timing mismatch can lead to lose data or incorrect sample data.

Some CDC techniques are common used: Synchronizer circuits: We use multiple flip-flops to resample coming signal to destination block. It reduces probability of metastability. Here 2.7 is synchronizer circuits using 2 flip-flops:

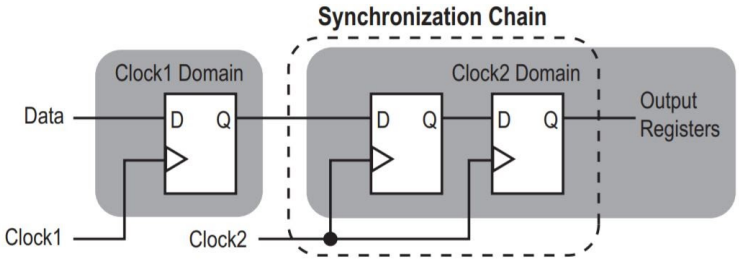


Figure 2.7: 2 Flip-flops synchronizer circuit quote from link

As image above, 2 flip-flops synchronizer circuit using 2 flip flops connected in series. The second flip flop is used as a backup for the first flip flop, because the first flip flop can enter metastable state or meet some issues like violation setup time and hold time. However, using 2 flip flops still have small

chance to enter metastable state. So to improve it, we can use more flip flops (3 or 4 flip flops), or using handshake mechanism as image 2.8

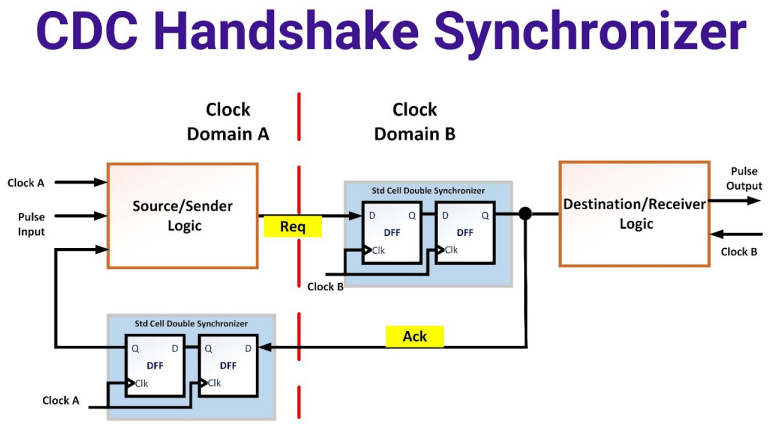


Figure 2.8: 2 Flip-flops synchronizer circuit with handshake quote from link

Sender domain: send data and assert request signal, wait ack signal from receive domain. After that, de-assert request signal.

Receive domain: detect positive edge of request, read data. Send ack signal to sender, wait negative edge of request signal. After that, de-assert ack signal.

Additionally, we can use asynchronous FIFO (dual port sram) for synchronize signal between 2 domain clocks. Sender write data to port, and receiver read data from another port as image 2.9.

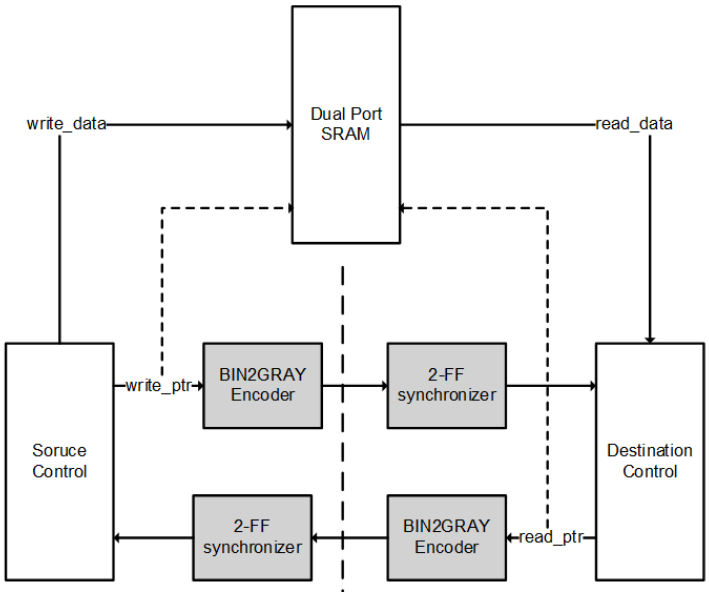


Figure 2.9: Asynchronous FIFO quote from link

2.6 Advanced eXtensible Interface (AXI) bus protocol

AXI (Advanced eXtensible Interface) is a type of bus within the AMBA (Advanced Microcontroller Bus Architecture) family developed by ARM. AXI is an advanced, flexible, and high performance bus protocol designed to meet the needs of modern System-on-Chip (SoC) architectures. AXI supports parallel communication and includes several enhancements to improve performance and scalability. Some key features of AXI include out-of-order transactions, parallel communication, burst transfers, and pipelining.

Below is an illustration showing the signals commonly used in AXI:

Global Signals	Write Address	Write Data	Write Return	Read Address	Read Return
ACLK ARESETN	AWVALID AWREADY	WVALID WREADY	BVALID BREADY	ARVALID ARREADY	RVALID RREADY
	AWADDR AWPROT	WDATA WSTRB	BRESP	ARADDR ARPROT	RRESP RDATA
	AWID AWLEN AWSIZE AWBURST AWLOCK AWCACHE	WLAST	BID	ARID ARLEN ARSIZE ARBURST ARLOCK ARCACHE	RID RLAST
	AWQOS			ARQOS	
	AWUSER	WUSER	BUSER	ARUSER	RUSER

Figure 2.10: Common signal in axi protocol quote from link

AXI operation flow: First, it performs a handshake on the address channel, then processes data on the data channel, and finally operates on the response channel (for write transactions).

An example of a handshake: the handshake is successful when both the valid and ready signals are asserted (i.e., both are high).

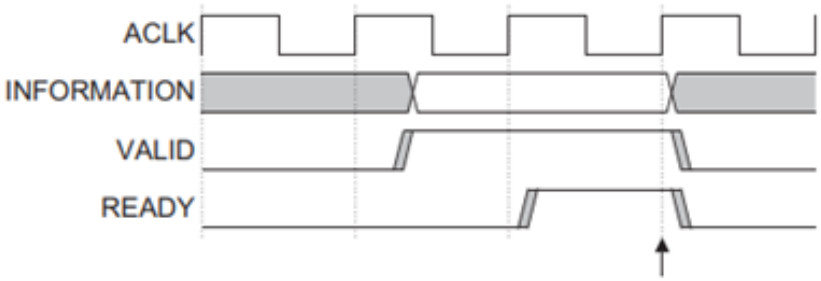


Figure 2.11: Handshake in axi quote from link

Below is a basic example of a read burst transfer:

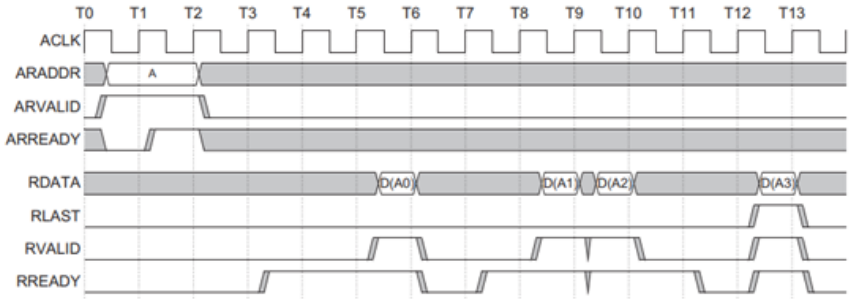


Figure 2.12: Read burst transfer in axi quote from link

As shown in the figure above, from T0 to T2, the operation is in the address channel. A successful handshake occurs when both the ARVALID and ARREADY signals are high, at which point the process moves to the data channel. In the data channel, the output data is considered valid when both RVALID and RREADY are high. Finally, when the RLAST signal is asserted (i.e., equals 1), it indicates that the read transfer has completed.

Below is a basic example of a write burst transfer:

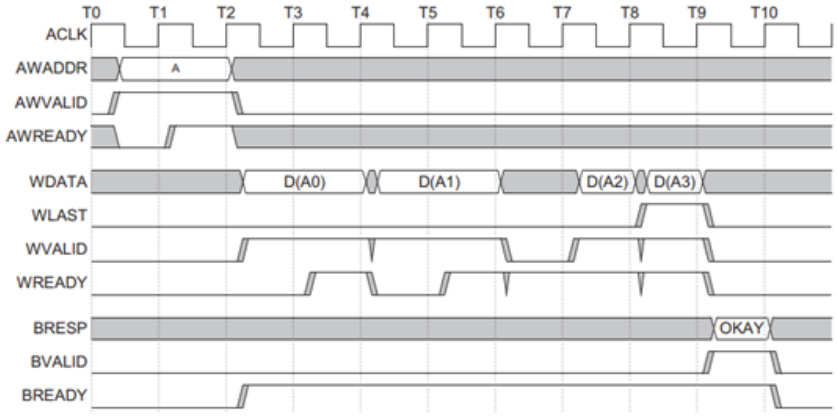


Figure 2.13: Write burst transfer in axi quote from link

First, it also operates on the address channel, with a successful handshake occurring at T2. Then, write data is transferred over the data channel, and the WDATA signal is considered valid when both WVALID and WREADY are high. Finally, when WLAST equals 1, it indicates that this is the last piece of data to be written. After that, the system waits for a response on the response channel. When the BRESP signal is OKAY, it indicates that the write transfer has completed successfully.

2.7 Digital Video Port (DVP) protocol

Digital Video Port (DVP) is a common digital image interface standard, often used to connect camera sensors to processors such as microcontrollers, FPGAs, or SoCs. It is a parallel interface, simple and easy to implement, especially suitable for embedded systems that require moderate image resolution and frame rate.

Here is main signals in DVP interface:

- **PCLK:** Clock signal used to synchronize pixel data transmission.
- **D[7:0]:** 8-bit data bus carrying pixel data. Each pixel may require 1 or 2 clock cycles depending on the image format.
- **HSYNC:** Indicates the start or end of a row in the image.
- **VSYNC:** Indicates the start or end of a frame.
- **XCLK:** External clock signal provided to the camera by the processor.

Operation principle: When receiving the XCLK signal, the camera begins operation and outputs pixel data on each PCLK cycle. The pixel data is placed on the D[7:0] bus, and HSYNC and VSYNC signals are used to indicate the boundaries of rows and frames. The processor reads the data based on these signals to reconstruct the complete image.

Here is common image formats in DVP:

- **RGB565:** 16 bits per pixel $\rightarrow$ each pixel transmitted in 2 PCLK cycles.
- **Grayscale:** 8 bits per pixel $\rightarrow$ each pixel transmitted in 1 PCLK cycle.
- **YUV422:** A compressed color format used in some video applications.

2.8 Serial Camera Control Bus (SCCB) protocol

The Serial Camera Control Bus (SCCB) is a serial communication protocol developed by OmniVision Technologies, mainly used to interface a host processor (such as a microcontroller or FPGA) with camera sensors like the OV7670, OV2640, OV5640,... SCCB is not designed for transferring image data, but rather for configuring internal control registers of the camera, such as resolution settings, output format, brightness, contrast.

While SCCB is similar in concept to the standard I2C (Inter-Integrated Circuit) protocol, it differs slightly in signaling and does not fully implement I2C features like acknowledgment (ACK/NACK) or multi-master support.

Here is 2 main signals in SCCB:

- **SIO_C:** Clock line, controlled by the master device.
- **SIO_D:** Bidirectional data line for serial communication

Operation principle:

- Firstly, it sends start condition by de-assert SIO_D while SIO_C remains HIGH.
- Secondly, the master sends the camera device address (7 bits) followed by a write (0) or a read (1) bit. Then, master sends the register address within sensor. At the end, master send data to be written to selected register.
- Finally, it sends stop condition by assert SIO_D while SIO_C remains HIGH.

2.9 Display Bus Interface (DBI) protocol

DBI (Display Bus Interface) is a communication standard defined by MIPI (Mobile Industry Processor Interface), designed to connect processors (MCUs, MPUs, FPGAs) to display modules (typically TFT LCDs). The DBI standard provides a method to transmit commands and data from the controller to the display controller, enabling pixel rendering on the screen.

MIPI defines three main DBI types:

- **DBI Type A:** Serial communication (commonly SPI).
- **DBI Type B:** Parallel communication in 8080/6800 style.
- **DBI Type C:** Extended version, used in DPI/DSI-like transfers.

Here is main signals in DBI type B (main using in my thesis):

- **D[7:0] / D[15:0]:** 8-bit or 16-bit data bus (for commands/data)
- **CS (Chip Select):** Selects the LCD module (active low)
- **RS (Register Select):** 0 = command, 1 = data
- **WR (Write Enable):** Write strobe; falling edge triggers a write
- **RD (Read Enable):** Read strobe; falling edge triggers a read
- **RESET:** Resets the LCD to default state
- **VCC, GND:** Power supply and ground

Operation principle:

Write command:

- Set CS = 0 (select display).
- Set RS = 0 to indicate a command.

- Place the command code on D[7:0] (or D[15:0]).
- Generate a write pulse: WR = 0, hold some cycles, then WR = 1.
- Set CS = 1 to finish.

Write data:

- Set CS = 0.
- Set RS = 1 to indicate a data.
- Generate a write pulse: WR = 0, hold some cycles, then WR = 1.
- Set CS = 1 to finish.

Read data: (if supported)

- Set CS = 0.
- Set RS = 1 to indicate a data.
- Set RD = 0, wait for the display put data in D[7:0] (or D[15:0]) (negative edge).
- Set RD = 1, and then CS = 1 to finish.

Chapter 3

Related works

Studies of the human-detection algorithms have been carried out in the hardware implementation for a long time. There are extensive papers that illustrate the optimal solution for human detection, especially YOLO and HOG-SVM.

Many FPGA-based YOLO designs use Vivado high-level synthesis (HLS) [1], [2], [3], but these projects are not optimized in terms of both hardware resource and performance. C. Zhang [1] shows a single processing engine (PE) using a theoretical roofline model to design an accelerator to execute each layer. However, the accelerator utilizes a significant portion of the FPGA chip with a modest throughput of 61 giga operations per second (GOPS) for a small network of five levels. M. Alwani [2] proposes a fused-convolutional layer to reduce the off-chip memory access by enabling caching of intermediate data between the evaluation of adjacent CNN layers. Similar to [1], the accelerator of CNN model in [3] decreases the number of weights using a pruning method in the fully connected layers by using Vivado HLS. The proposed accelerator can achieve a peak performance of 498.6 GOP/s and the power efficiency with the value of 21.3 GOP/s/W under 100MHz clock frequency. Although these HLS designs can be implemented in FPGA, they consumes a large number of resources and the programmable logic (PL) contributed in only a small part of models, the other parts were handled by the processing system (PS). It is not a full image

process flow in PL, from getting images to obtaining results. Another design is also a YOLO model but implemented completely by hardware description language (HDL) [4]. Authors illustrated a model having 17 convolution layers and 5 max-pooling layers [removed 2 CONV layers from YOLO-v2] and archived the maximum throughput of 30 fps with an accuracy of 64.16% in the resolution of 416 x 416.

On the other hand, abundant papers [5], [6], [7], [8] propose the HOG and SVM architecture for detecting human with FPGA with the better throughput and larger resolution. After the state-of-art HOG algorithm [9] was published by Dalal in 2005. All of the next FPGA implementations use a slide window of 128 x 64 pixels, a cell of 8 x 8 pixels and a block of 2 x 2 cells, but with different approach. Pang [7] tested 320 x 240 resolution, while the design of Michael Hahnle [5] can run with HD resolution images (1920×1080) at 64 fps. Kazuhiro Negi [8] designed a slide window of 64×80 with the maximum throughput of 112 fps. However, the approach of these design is storing a high resolution image then fetching each slide windows and applying HOG-SVM algorithm in that slide window, but the features of previous windows can be recycled for the next windows, which causes the resource waste. Cell-based Scanning Method illustrated in [6] handles that problem. That architecture can generate HOG features and detect objects with 40 MHz for SVGA resolution video (800×600) at 72 fps.

Chapter 4

Proposed Architecture

As mentioned in previous chapter, we want to design an architecture that is a real-time system (ASIC), and we also have approached the cell-based scanning method to human detection and a pipeline architecture in each steps of HOG-SVM algorithm to reduce the system frequency, but there is a process from getting an image to archiving results implemented totally in HDL.

4.1 System

In this project, we implement the ASIC for smart camera surveillance at the edge. We integrate all the necessary components of a camera into the system. Moreover, the system includes an image processor, which is used to classify captured images at the edge and satisfy real-time processing purposes. Besides, the system is programmable; users can configure internal components or control the image data flow. The architecture of the system is shown in the figure 4.1. The system is separated into 2 subsystems:

- Configuration subsystem
- Image processing subsystem

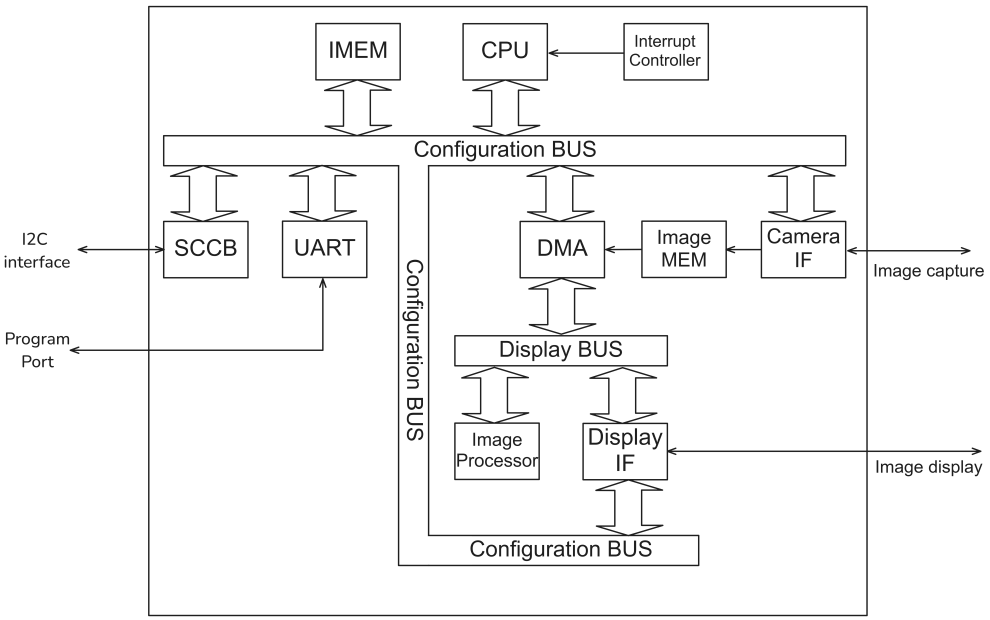


Figure 4.1: The system's architecture

4.1.1 Configuration subsystem

The configuration subsystem enables the system to support programming, configuration of internal components, and setting up the image streaming flow in the image processing subsystem. The figure 4.2 illustrates the configuration subsystem. There are 6 main components in the configuration subsystem:

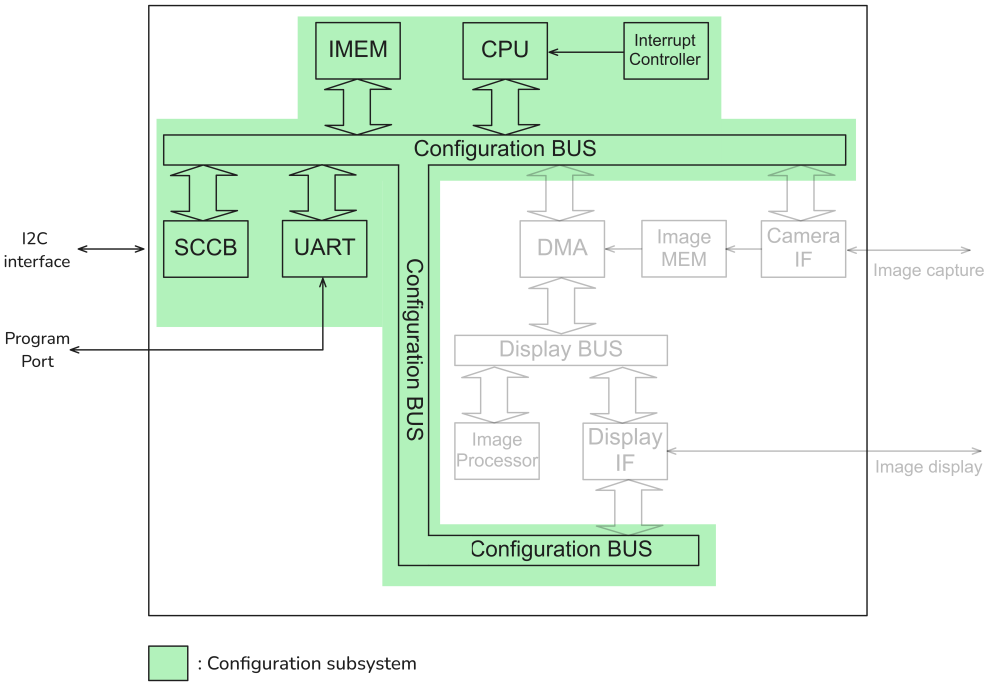


Figure 4.2: The configuration subsystem's diagram

- **Central processing unit (CPU)**: Used to process RV32I instructions stored in the instruction memory. The CPU is a main master of the system's bus.
- **Instruction memory (IMEM)**: Used to store 3 programs of the system, consisting of a bootloader program, a main program, and an interrupt service routine program.
- **Interrupt controller**: Used to collect interrupt signals from multiple sources into a single interrupt signal, and forward the single interrupt signal to the processor
- **UART controller**: Used for the system to communicate with external components via the UART interface. Moreover, the UART controller is used as a programming port for the system.
- **SCCB controller**: Used for the system to configure the external camera's parameters via the SCCB interface.
- **Interconnect**: Used to connect components in the system, and route transactions from a single master to multiple slaves.

4.1.2 Image processing subsystem

The processing subsystem is used to process images, which are captured into the system via the camera interface, by classifying them into two groups: images with humans and images without humans. Moreover, the user can control the image streaming flow by configuring the direct memory access (DMA). The image processing subsystem provides a display interface to stream out the captured images, giving the user flexibility in deciding whether to stream images out. The figure 4.3 shows the diagram of the image processing subsystem.
There are 5 main components in the image processing subsystem:

- **Camera interface (Camera IF)**: Used to capture images from the external camera into the system.

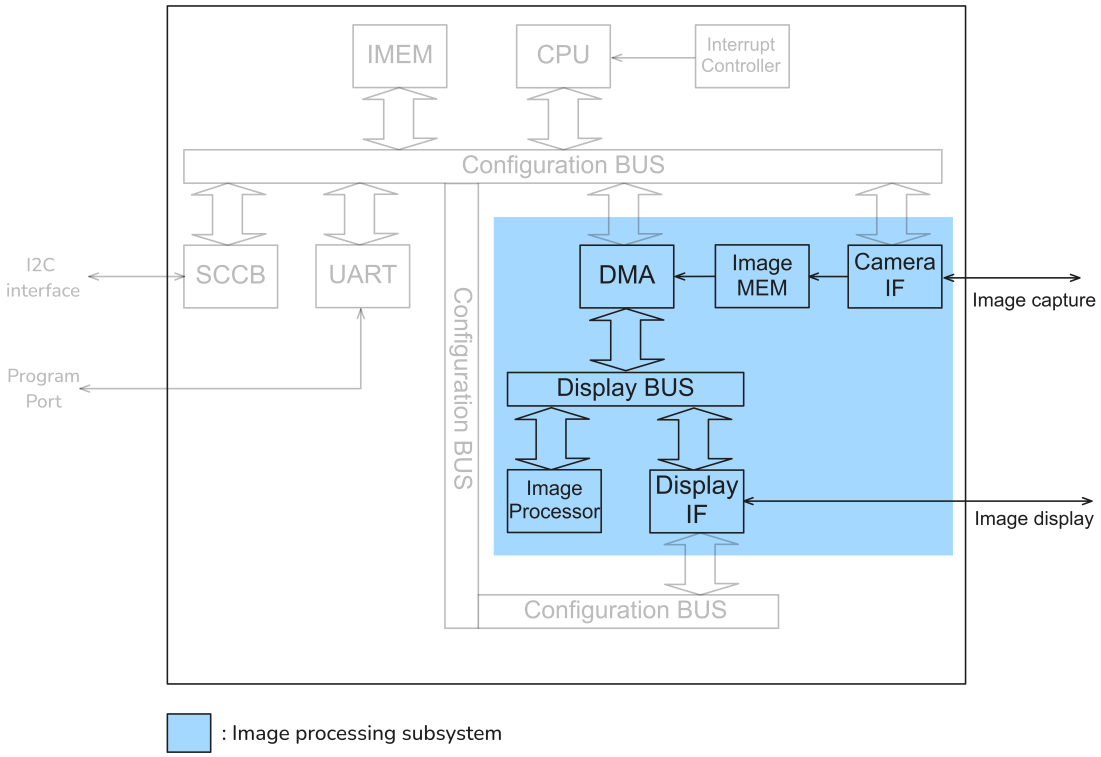


Figure 4.3: The image processing subsystem's diagram

- **Image memory (Image MEM)**: Used to store the image, which is captured from the camera interface.
- **Direct memory access (DMA)**: Used to control the image streaming flow from the image memory to two destinations: the image processor and the display interface.
- **Image Processor**: Used to process and classify the captured images.
- **Display interface (Display IF)**: Used to stream the images to the external display component.

4.2 Image processor

As mentioned in previous chapter, we want to design an architecture that is a real-time system, we also have approached the cell-based scanning method to human detection and a ***pipeline architecture*** *in each steps of HOG-SVM algorithm to reduce the system frequency, but there is a process from getting an*

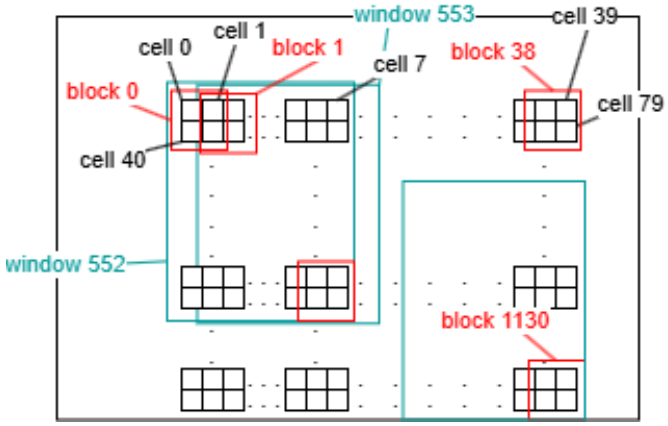
image to archiving results implemented totally in HDL.

Image resolution	640 x 480 pixels
Scale ratio	1 : 2
Cell size	8 x 8 pixels
Block size	2 x 2 cells
Slide window size	7 × 15 blocks or 128 x 64 pixels

Table 4.1: Image parameters

4.2.1 Cell-based scanning method

Proposed by [6], this method will scan cells line-by-line and then stored histogram results until getting a block in the histogram normalization step (figure 4.4). Compared with window-based scanning [9] (Figure 4.5) having a loop and scanning overlap cells, cell-based scanning method shares and reuses of a cell, so having a significant impact on memory bandwidth reduction (Figure 4.6).



Cell histogram generation	Cell 0	Cell 1	Cell 2	...	Cell 40	Cell 41	Cell 42	Cell 43	...	Cell 608	Cell 609	Cell 610	Cell 611
Histogram normalization						Block 0	Block 1	Block 2	...	Block 552	Block 553	Block 554	Block 555
SVM classification							Block 0	Block 1	...	...	Block 552	Block 553	Block 554
Results												Window 552	Window 553

time

Figure 4.4: Cell scanning method

In SVM classification, image features from histogram normalization will be multiplied with SVM coefficients then stored and accumulated with next results until getting a window result.

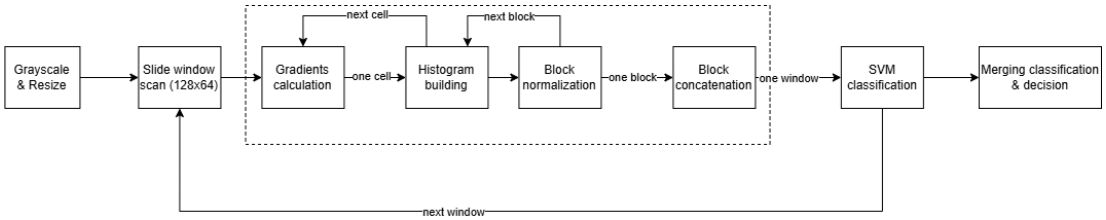


Figure 4.5: Window-based scanning method

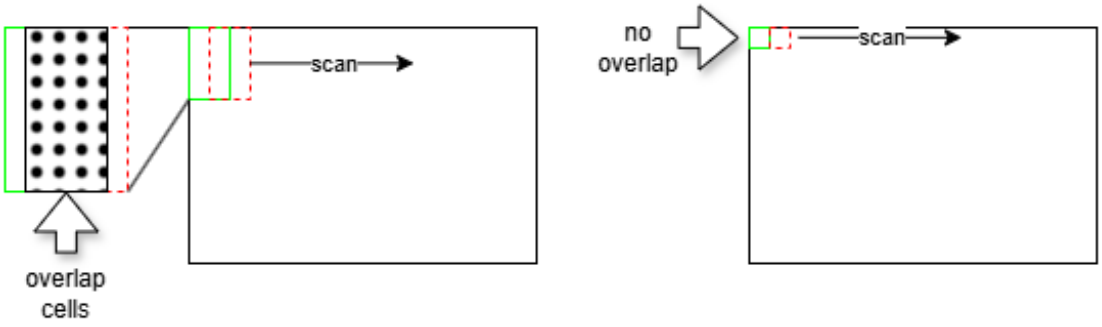


Figure 4.6: Window-based scanning vs cell-based scanning

Note: For hardware optimization, slide window indexes will be the last indexes of blocks in that windows.

In general, a software implementation utilizes floating-point number format to archive high accuracy, but the floating-point unit uses hardware resources to a great degree. Therefore, fixed-point operation is often used for hardware implementation. The accuracy depends on the fractional width, although that affects the memory capacity and the calculation complexity. Table 4.2 presents the results of parameter adjustment.

Number	Signed or not	Integer width	Fraction width
Gradient (G_x, G_y)	signed	8	0
tan value	signed	3	16
Magnitude	unsigned	9	4
Bin	unsigned	16	4
HOG feature	signed	4	16
SVM coefficient	signed	4	16

Table 4.2: Number format in fixed point

4.2.2 Diagram

The diagram of *image processor* is illustrated in figure 4.7. It includes:

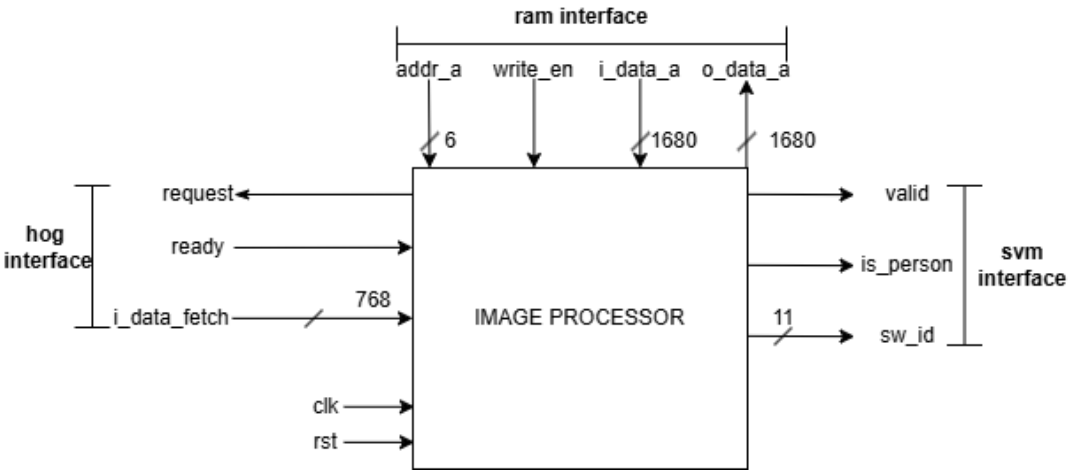


Figure 4.7: Diagram of the image processor

- HOG interface:
  - request (output) & ready (input): handshake signals to fetch input data.
  - i_data_fetch (input): input data, it contains a cell (8x8 pixels) and 4 borders of the cell.
- Ram interface: to access coefficients RAM of SVM system.
  - addr_a (input): (port a) the address of accessed element.
  - write_en (input): (port a only) if assert, there is a write instruction. Otherwise, there is a read instruction.
  - i_data_a (input): (port a) the input data to write into accessed element.
  - o_data_a (output): (port a) the output data to read from accessed element.
- SVM interface:
  - valid (output): the below SVM signals is valid, only if this control signal is asserted.
  - is_person (output): if this image processor detects people in a slide window, this signal will be asserted.
  - sw_id (output): the index of slide windows

The detailed architecture is shown in figure 4.8

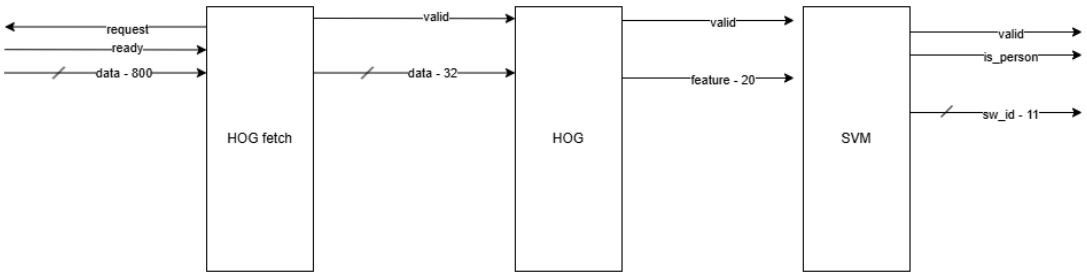


Figure 4.8: Detail of the image processor

4.2.3 HOG fetch module

This module will fetch data from the frame memory, store cell data into cell buffer, then transfer a group of pixel data in turn. A group of pixel data is defined as {top, bot, left, right}

For example: In resolution 320x280, data group of pixel 0 is {0, pixel 320, 0, pixel 1}, data group of pixel 1 is {0, pixel 321, pixel 0, pixel 2}

The diagram of HOG fetch is shown in figure 4.9. Because the input is a cell (8x8 pixel) and the ouput is pixel group, it will take at least 64 clocks for the next asserting request signal.

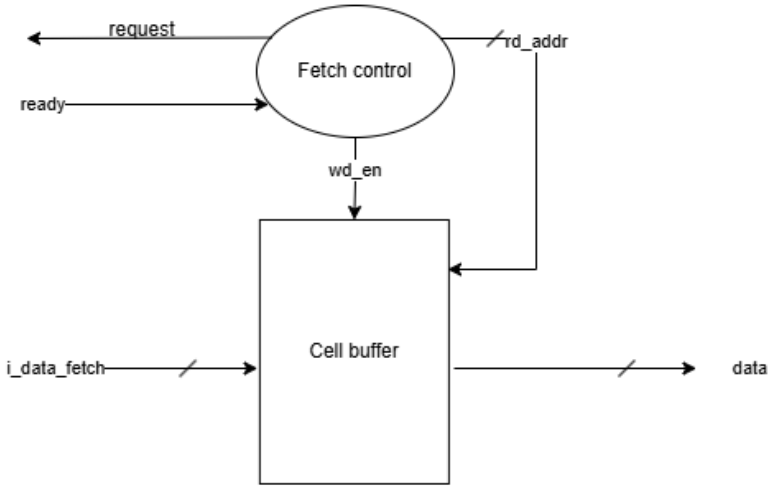


Figure 4.9: HOG fetch module

4.2.4 HOG module

As mentioned in previous part, the 3-stage pipeline architecture is applied in this HOG module (Figure 4.10).

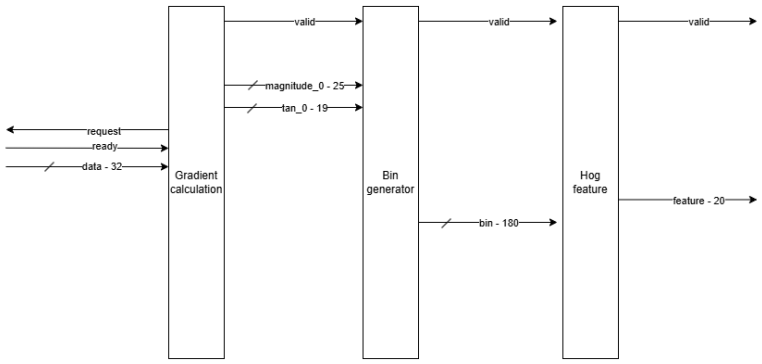


Figure 4.10: HOG module

There is a cell-based FIFO with the handshake mechanism on the outside of the HOG module. The HOG module consist of:

- Gradient calculation (1 cycle): calculating the gradient of a pixel: $bottom - top$, $right - left$; returning a corresponding magnitude, and tan
- Bin generator (1 cycle): arranging all magnitude results into 9 bins, depending on tan and negative.
- HOG feature (2 cycles): storing bins and returning 4 features if getting a block (feature map in figure 4.11; for simplicity, bin_* in the figure is 9-bin groups)

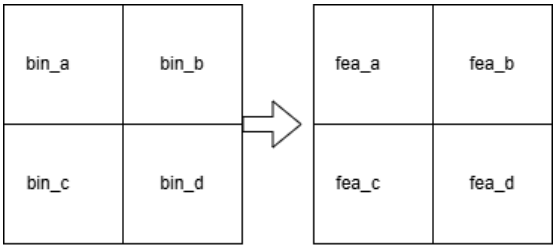


Figure 4.11: Feature map in a block

4.2.5 SVM module

In the SVM classification module, extracted features are multiplied with SVM coefficients and accumulated until the operations reach one window level. Then the accumulation result is compared with an SVM threshold to evaluate whether the window includes people or not. Figure 4.12 shows a block diagram

for simultaneous SVM classification. This architecture consists of 15 classification cores. One classification core manages MAC operations of 7 blocks. As a result, the architecture can handle 105 blocks corresponding to one detection window.

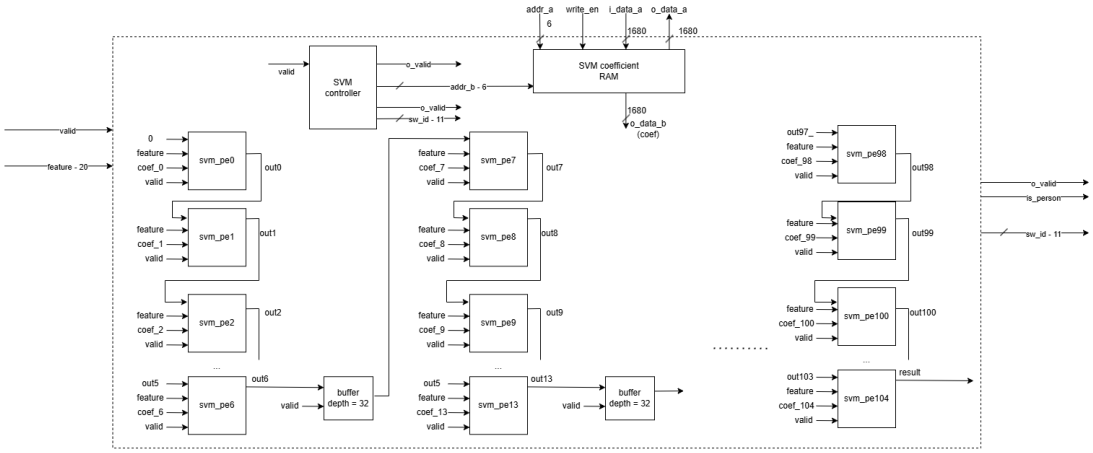


Figure 4.12: SVM architecture

- SVM controller: deciding when the signal *o_valid* asserts, and returning the index of slide windows.
- SVM coefficient RAM: storing 3780 SVM coefficient.
- SVM PE (Process element): handling SVM calculation.
- Buffer for the result storage.

Because of the multiplication and accumulation requirement, the output of SVM PE is connected with the next input of that sub-module. In addition, cell-based scanning method need to save PE results for 32 cycles to add with the product of the next line.

For example in figure 4.4, after SVM_PE 6 returning:

$$S = \sum_{i=0}^{6} block_i * coef_i \tag{4.1}$$

The next operation is $S + block_{39} * coef_7$ for window 552, so the result S

must be saved when scanning block 7, 8, ..., and 38. The depth of buffers is $38 - 6 = 32$.

4.3 Cell fetch

To integrate the Image processor with the system, we provide the cell fetch module to receive the image data from another AXI master component via the AXI-stream interface and fetch 8x8 cells to the Image processor. The figure 4.13 shows the connection between the image processor and the AXI master component.

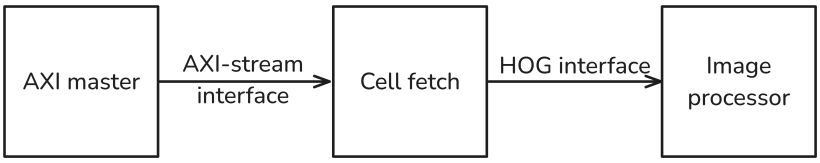


Figure 4.13: The overview of the cell fetch

The cell fetch converts the line-based fetching of AXI component to the cell-based fetching of the HOG operation. The figure 4.14 shows the block diagram of the cell fetch. The AXI master component fetches the data from the image memory into the cell fetch, fetching row by row from left to right and top to bottom. While fetching row by row, the cell fetch module maps the AXI data to the corresponding cell buffers. After one cell is completed, it is stored in the cell cache. After one frame is fetched by AXI master component, the cell fetch starts fetching the cell into the Image processor.

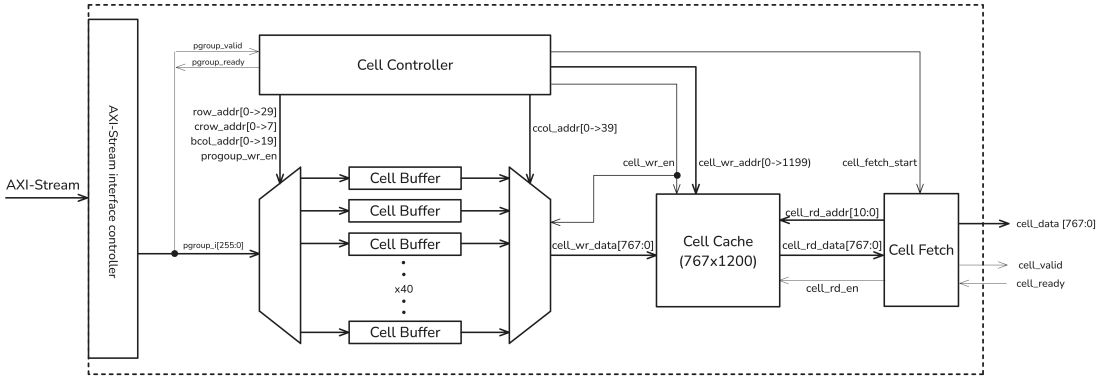


Figure 4.14: The block diagram of the Cell fetch module

The figure 4.15 illustrates the pixel line term and the cell term.

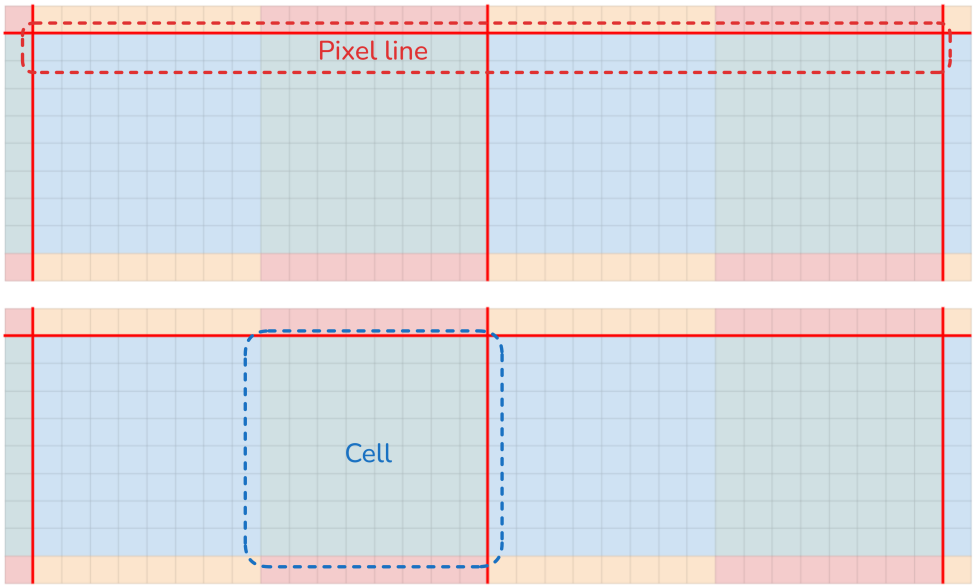


Figure 4.15: The pixel lines and the cells

There are 2 main interfaces of the Cell fetch module:

- AXI-stream slave interface: Used to receive the 256-bit pixels line from the AXI master component.
- HOG interface: Used to fetch the 8x8 cell to the Image processor.

There are 3 main processing stages in the cell fetch module:

- The cell buffer mapping: Used to map pixels in a pixel line, which are received from the AXI-stream interface, to the corresponding cells. The figure 4.16 illustrates the mapping mechanism of the cell buffer module.

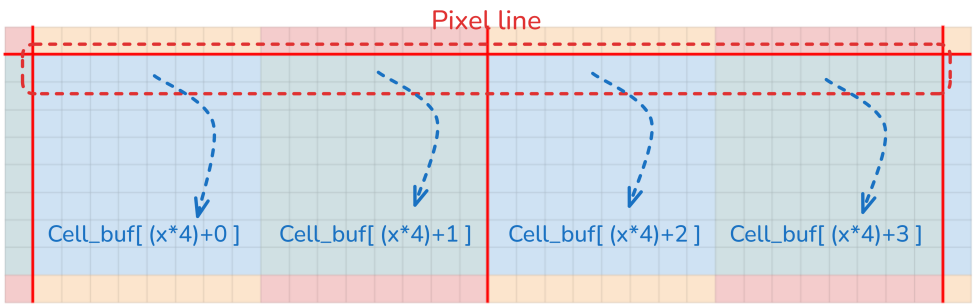


Figure 4.16: The mapping mechanism of the cell buffer module

After 1 cell is completed, it is stored in the corresponding address in the cell cache based on the cell's address in the frame.

- The cell cache storing: Used RAM to store all cells of a frame. The interface of the RAM is shown in the figure 4.17.

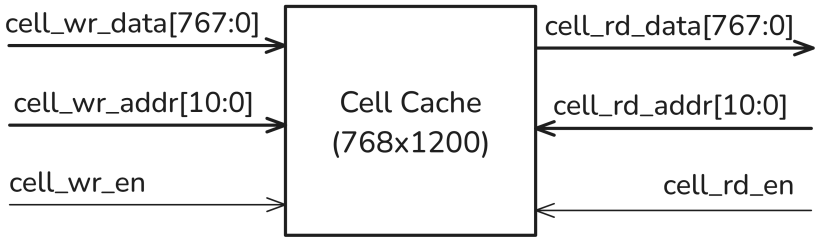


Figure 4.17: The interface of the cell cache

- The cell fetching: After all the cells in a frame are completed, the cell controller starts fetching to the HOG interface until all cells are sent.

The cell controller block controls the addresses of all pixel lines, which are used to map to the cell buffers and the cell cache, and the fetching state

4.4 Camera interface

To capture images into the system, we use an OV7670 camera, which employs the Digital Video Port (DVP) protocol to stream image data to the system. Therefore, we design a DVP RX Controller integrated with an internal AXI DMA to store the images in the image memory.
The figure 4.18 illustrates the design of the DVP RX Controller.
There are 3 main interfaces of the DVP RX Controller:

- DVP interface: Using the Digital Video Port protocol to communicate with the OV7670 camera.

- AXI4 slave interface: The AXI master can configure parameters or monitor the state of the DVP RX controller via this interface.

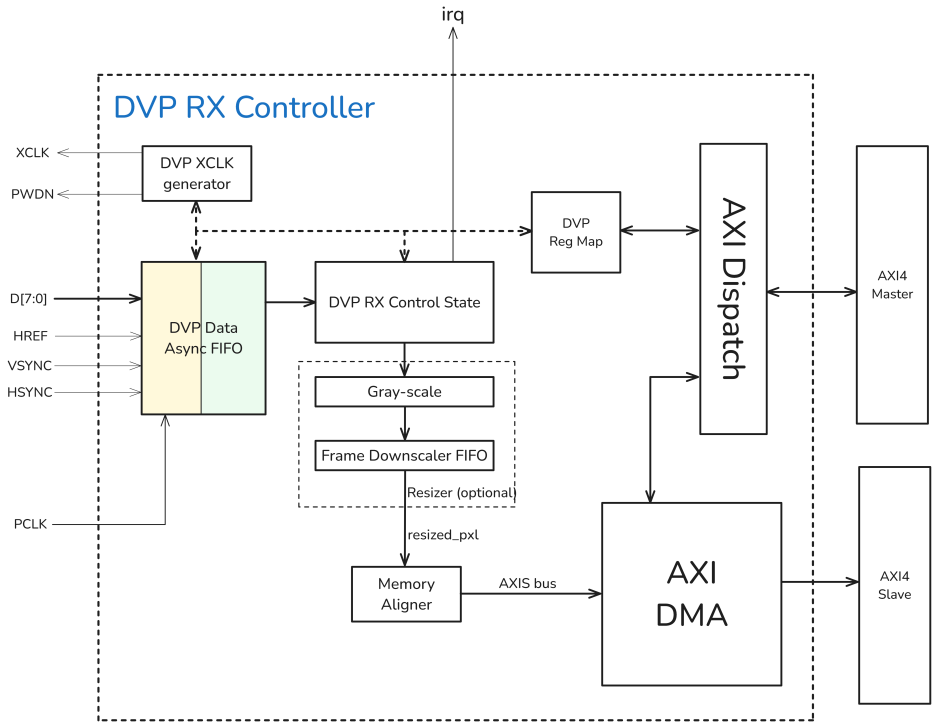


Figure 4.18: Camera Interface architecture

- AXI4 master interface: This interface is used for streaming the images captured from the camera to the destination memory.

There are 6 main components in the controller:

- **DVP Data Asynchronous FIFO** block: The OV7670 uses a separate clock to stream data via the DVP interface, which is out of phase with the system clock (XCLK). To handle this clock domain crossing, we use a synchronizer to synchronize data with the system clock. The synchronization mechanism we use is an Asynchronous FIFO, where the write-clock is driven by the camera's clock and the read-clock is driven by the system clock. The figure 4.19 illustrates the DVP Data Asychronous FIFO

- **Register Map** block: To configure the DVP RX Controller, the processor can access and modify control registers to set some parameters, such as the capturing mode of the camera, the interrupt masks, etc., by using the memory-mapping mechanism. The base address the DVP RX controller is 0x4000_0000. More details about register map table 4.20

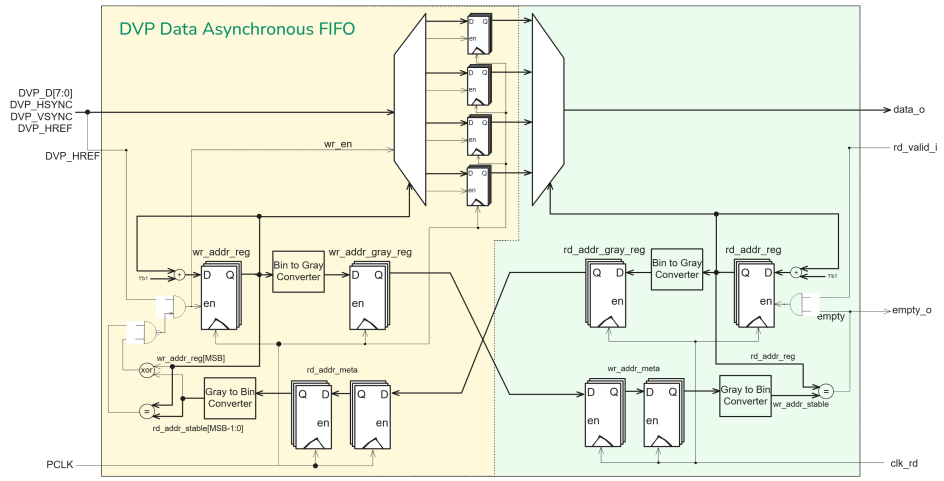


Figure 4.19: The DVP Data Asynchronous FIFO detail block

Register Name	Register Offset	Bit map	Description	Type
CAM_RX_EN	0x0000			RW
		[0]	Enable the RX Controller	
		[31:1]	Reserved	
CAM_PWDN	0x0001		Set dvp_pwdn_o output pin	RW
CAM_RX_MODE	0x0002		Set capturing mode	RW
		[1:0]	2'b00: SLEEP 2'b01: SINGLE-SHOT 2'b10: STREAM	
		[31:2]	Reserved	
CAM_RX_START	0x0010			RW1S
CAM_RX_STATE	0x0020			RO
		[1:0]	3'b000: SLEEP state 3'b001: IDLE state 3'b010: PXL_ALIGN state 3'b011: PXL_CAPTURE state 3'b100: ERR_CORRECT state	
		[31:2]	Reserved	
CAM_RX_LEN	0x0021		Number of received pixels of the current frame	RO
IRQ_MASK	0x0003			RW
		[0]	IRQ_FR_COMP_MASK	
		[1]	IRQ_FR_ERR_MASK	
		[31:2]	Reserved	
		[31:1]	Reserved	
IMG_WIDTH	0x0004	[15:0]	The width of captured images	RW
IMG_HEIGTH	0x0005	[15:0]	The height of captured images	RW

Figure 4.20: Register table of the DVP RX controller

- **DVP RX State Machine** block: To handle the streaming flow, we use a finite state machine to control the entire controller. There are some features handled by the state machine, such as sleep mode, pixel alignment, error correction, etc. In addition, the block not only controls the state machine,

but also merges DVP data to 1 RGB block. The block diagram of the DVP RX control state is illustrated in the figure 4.21. The finite state machine diagram is illustrated in the figure 4.22

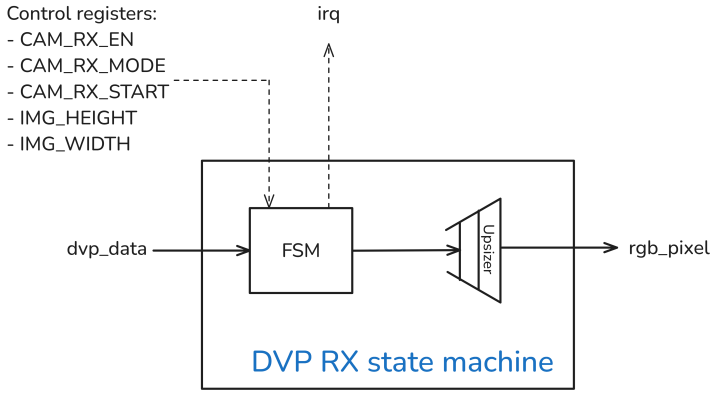


Figure 4.21: The block diagram of the control state block

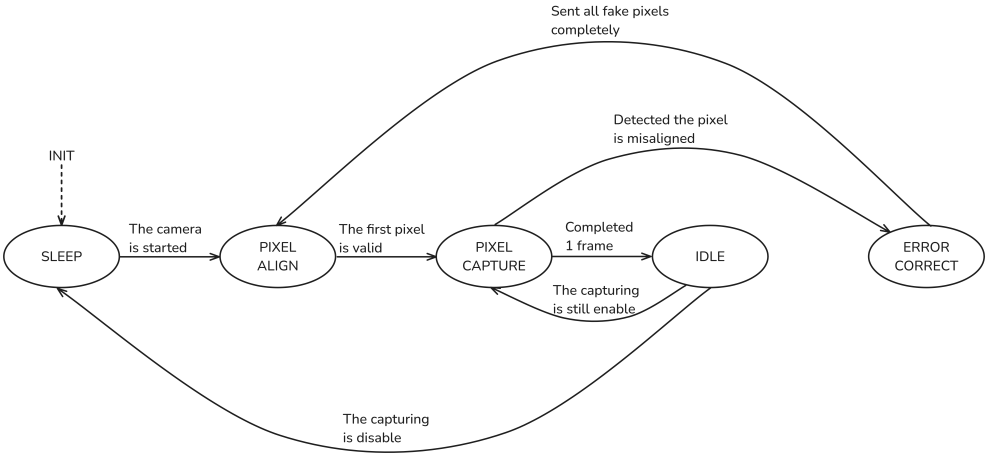


Figure 4.22: The main state machine of the DVP RX Controller

- **Resizer** block: After receiving the RGB pixels from the RX state control block, the module include some scalers to make the frame compatible with the HOG-SVM algorithm and to reduce memory resource usage in the system, there are a frame downscaler and a pixel grayscaler. We use the average-pooling algorithm for the downscaler (illustrated in the figure 4.23)

- **Memory Aligner** block: After receiving resized pixels from the Resizer block, those pixels are grouped into 1 word in the system (256-bit word) by

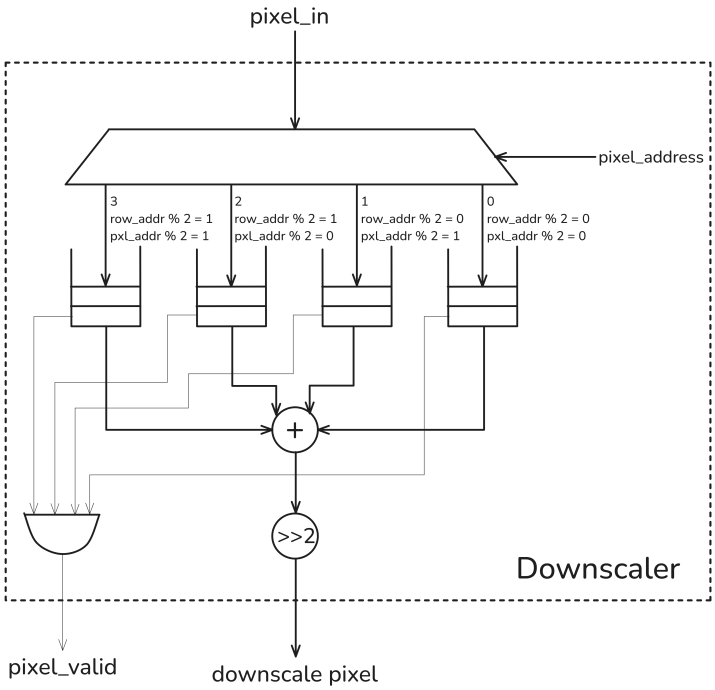


Figure 4.23: The downscaler detail block diagram

the Memory Aligner block. Then, the groups of pixels is pushed into the DMA to move to the image memory. The structure of the group is shown in the figure 4.24.

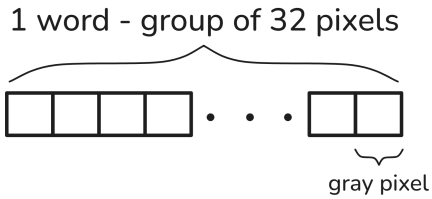


Figure 4.24: The word structure in the image memory

- **Direct-Memory-Access** block: Using AXI4 master to stream the groups of pixels to the memory. The processor can configure some parameters such as the destination address, some characteristics of AXI transactions, etc. The base address of the internal DMA region is 0x5000_0000. More details about register map in figure 4.25

We provide a document for more details about the DVP RX controller here

Register name	Register offset	Bit map	Description	Type
DMA General CSRs				
DMA_CONTROL	0x0000		The control signal of the DMA	R/W
		[0]	ENABLE signal of the DMA	
Descriptor				
DST_ADDR	0x000A		The destination address	RW
TRANSFER_SUBMIT	0x1000		The fifo write enable signal	RW1S
TRANSFER_X_LEN	0x000B		The number of words in a row + 1 (same as transfer length in 1-D transfer mode)	RW
TRANSFER_Y_LEN	0x000C		The number of rows + 1(equal 0 when DMA in 1-D transfer mode)	RW
DST_STRIDE	0x000E		The number of words between the start of one row and the next row for the destination address. Needs to be aligned to the bus width. Note, this field is only valid if the DMA channel has been configured with TRANSFER_2D support and write to memory support	RW
Channel CSR				
CHN_CONTROL	0x0001		The control signal of channel	RW
		[0]	ENABLE	
		[31:1]	Reserved	
CHN_FLAGS	0x0002			
		[0]	TRANSFER_2D	
		[1]	TRANSFER_CYCLIC : A cyclic transfer once completed will restart automatically with the same configuration.	
CHN_IRQ_MASK	0x0003			RW
		[0]	when a TRANSFER_COMPLETED, the interrupt request will be asserted	
		[1]	when a TRANSFER_QUEUED, the interrupt request will be assert	
		[31:2]	Reserved	
CHN_IRQ_SOURCE	0x2000			RO
		[0]	TRANSFER_COMPLETED	
		[1]	TRANSFER_QUEUED	

Figure 4.25: Register table of the internal DMA in DVP RX controller

4.5 Camera controller interface

The OV7670 camera uses the Serial Camera Control Bus (SCCB) protocol to control the parameters and the state of the camera. We provide an SCCB Master controller with the AXI interface to configure the camera via the processor. The AXI master can write/read the parameters of the camera via the controller. The block diagram of the SCCB master controller is shown in the figure 4.26.

There are 2 main interfaces of the SCCB master controller:

- SCCB master interface: Used SIO_C wire to generate the transaction clock,

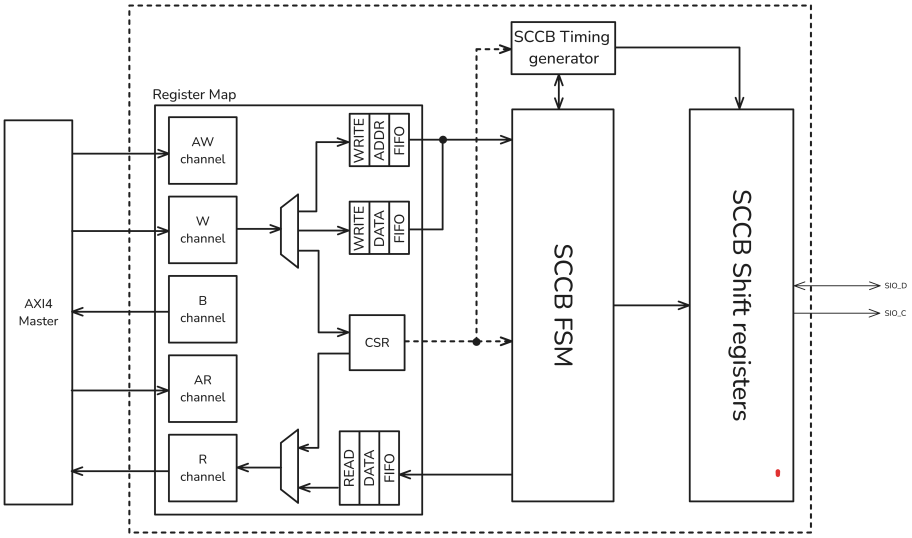


Figure 4.26: SCCB master controller architecture

and use bidirectional SIO_D wire to write/read the parameter or status of the camera

- AXI slave interface: Used to communicate with AXI masters in the system, the masters can configure, transmit SCCB-write data or receive SCCB-read data via this interface.

There are 3 main components in the SCCB master controller:

- **Register map** block: Used by AXI masters to configure, transmit data to, or receive data from the SCCB interface. There are 2 control registers, 3 TX registers and 1 RX register, shown in the figure 4.27
- **SCCB finite state machine** block: Used to control the state of SCCB transmissions. There are 3 types of SCCB transmissions, such as 2-phase-write, 3-phase-write, and 2-phase-read transmission. Each transmission has a Start of Transmission and a Stop of Transmission to align the SCCB frame; the number of phases based on the type of transmission; DATA and ACK in a phase to transfer and validate the data. The figure 4.28 shows the state machine diagram of the controller.
- **SCCB timing generator** block: Used to generate the SCCB clock (SIO_C clock) and generate the sample clock for the state machine.

Register's name	Bit map	Description	Address (default)
`SLV_DVC_ADDR_REG`	[7:0]	Slave device's address registers	`0x6000_0000`
`PRESCALER_REG`	[7:0]	The prescaler (clock divider) of the SCCB clock	`0x6000_0001`
`CONTROL_BUF`	[7:0]	Control signal buffer	`0x6000_0010`
	[1:0]	Represents the number of phases in a transmission	
	[2]	Represents the type of transmission - `1` : Write - `0` : Read	
	[7:3]	Reserved	
`SUB_ADDR_BUF`	[7:0]	The sub-address buffer of a SCCB transmission	`0x6000_0011`
`WRITE_DATA_BUF`	[7:0]	The write data buffer of a SCCB transmission	`0x6000_0012`
`READ_DATA_BUF`	[7:0]	Read data of SCCB transmissions buffer	`0x6000_0020`

Figure 4.27: SCCB master controller registers table

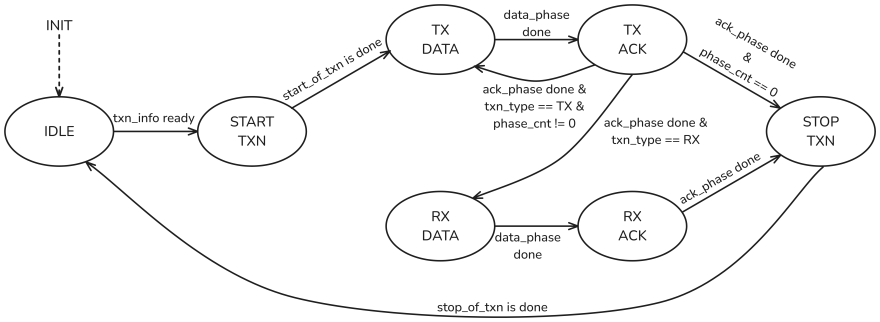


Figure 4.28: The state machine diagram of the SCCB master controller

4.6 Display interface

We using the LCD 3.5 inch to stream out the image from the system memory. The LCD uses the MIPI Display Bus Interface (DBI) Type-B protocol to capture images. Therefore, we provide the DBI TX controller to communicate with the LCD. The DBI TX controller uses the AXI4 interface for configuration via the processor and the AXI-Stream interface for streaming pixels into the system memory. The figure 4.29 illustrates the block diagram of the DBI TX Controller

There are 3 main interfaces in the DBI TX controller:

- MIPI DBI interface: Using MIPI Display Bus Interface Type-B to communicate with the LCD

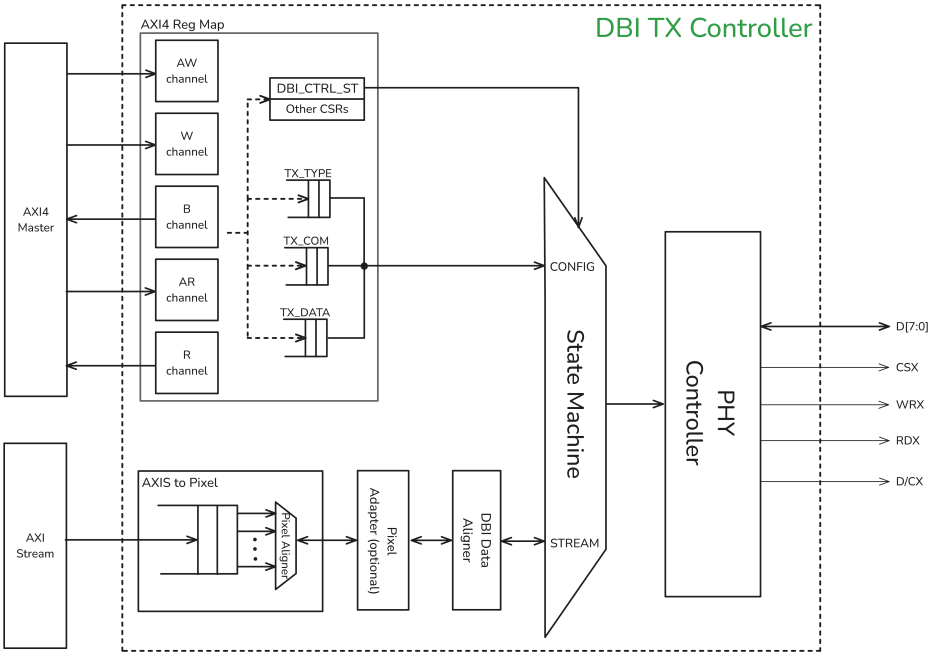


Figure 4.29: Display Interface architecture

- AXI4 slave interface: Used to configure the controller via register map. The AXI master can change the mode or parameters of the DBI TX controller via this interface.
- AXI-Stream slave interface: Used for streaming the image data to the controller.

There are 4 main components in the DBI TX controller:

- **Register map** block: To configure the parameters or the mode of the controller via AXI4 interface. The base address of all registers in the DBI TX controller is 0x2000_0000.
- **AXI-Stream to pixel** block: Using AXI-Stream slave interface to receive pixels and stream them in STREAM mode.
- **DBI adapter** block: the width of the system memory's word and the width of the DBI data bus are different, therefore, we provide the adapter to align the two separate widths.
- **State machine** block: In the MIPI DBI protocol, there are 2 types of transactions, such as LCD configuration command and Image streaming. There-

fore, we design 2 modes for the DBI TX controller: configuration mode and streaming mode. The processor can configure the mode of the controller via the register map block. The finite state machine of the block is shown in the figure 4.30.

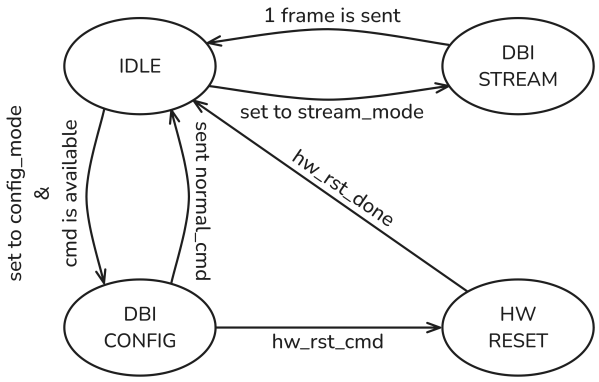


Figure 4.30: The finite state machine diagram of the State Machine block

- **PHY controller** block: The MIPI DBI protocol has some physical timing constraints based on the maximum frequency of the LCD (shown in the figure 4.31), therefore, we provide the PHY controller to satisfy these constraints.

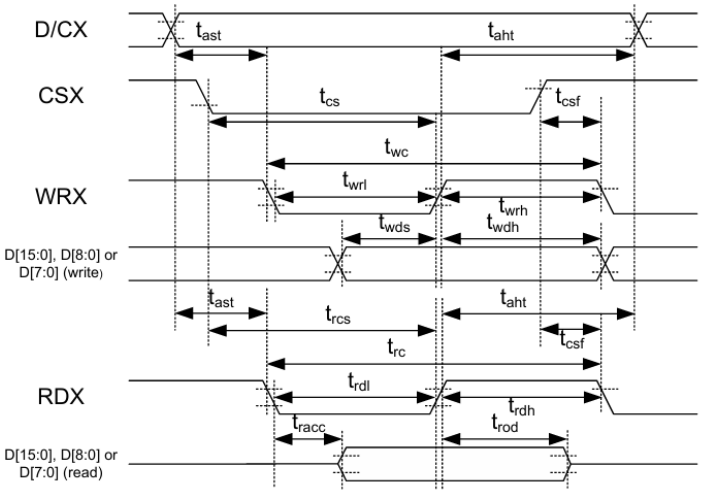


Figure 4.31: The physical timing constraints of MIPI DBI Type-B protocol

4.7 UART interface

The UART controller is provided in the system for programming and debugging purposes. The controller uses the AXI4 interface to communicate with the system. The AXI master can configure, transmit TX data, or receive RX data via the AXI interface. The figure 4.32 shows the block diagram of the UART controller.

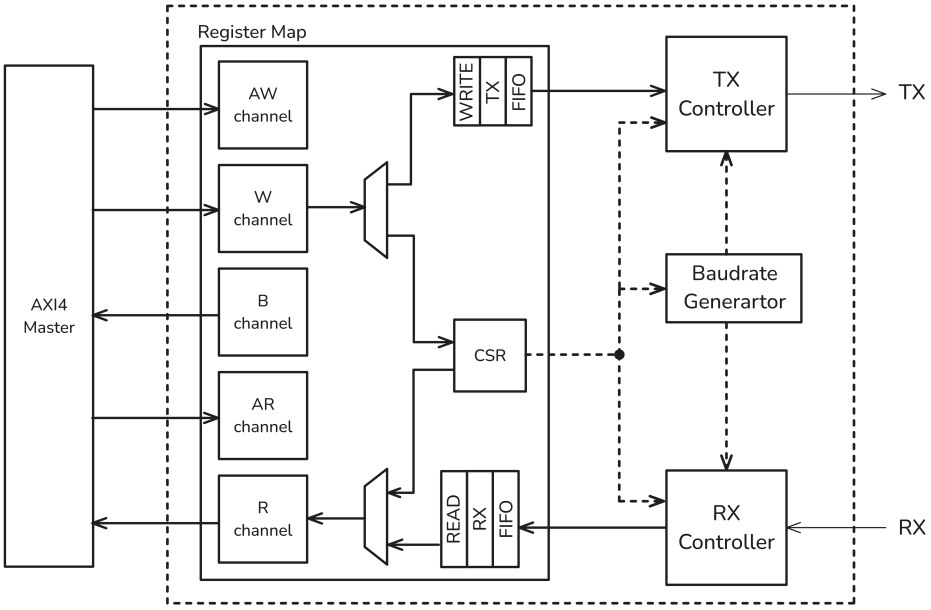


Figure 4.32: The block diagram of the UART controller

There are 2 main interfaces of the UART controller:

- UART interface: Used to communicate with external devices via the UART protocol
- AXI4 slave interface: Used to communicate with the system

There are 4 main components of the UART controller:

- **Register map** block: Used for the AXI master to map into registers for configuration, transmission, or reception. There are 3 configuration registers shown in the figure 4.33.

Register Name	Offset	Bit Map	Description	Type
TX_CONF	0x0000		The TX transmission configuration register	RW
		[7:5]	Baudrate selection of TX transmissions B4800 - B9600 - B19200 - B38400 - B14400 - B28800 - B57600 - B115200	
		[4]	Stop bit selection of TX transmissions - STOP1B: b0 - STOP2B: b1	
		[3:2]	Parity bit selection of TX transmissions - NO_PARITY: b00 - ODD_PARITY: b01 - EVEN_PARITY: b10	
		[1:0]	Data length selection of TX transmissions - DATA5B: b00 - DATA6B: b01 - DATA7B: b10 - DATA8B: b11	
RX_CONF	0x0001		The RX transmission configuration register	RW
		[7:5]	Baudrate selection of RX transmissions B4800 - B9600 - B19200 - B38400 - B14400 - B28800 - B57600 - B115200	
		[4]	Stop bit selection of the RX transmission - STOP1B: b0 - STOP2B: b1	
		[3:2]	Parity bit selection of the RX transmission - NO_PARITY: b00 - ODD_PARITY: b01 - EVEN_PARITY: b10	
		[1:0]	Data length selection of the RX transmission - DATA5B: b00 - DATA6B: b01 - DATA7B: b10 - DATA8B: b11	
TX_DATA	0x0010	[7:0]	TX data phase	RW1S
RX_DATA	0x0020	[7:0]	RX data phase	RW1C

Figure 4.33: The configuration register table of the UART controller

- **Baudrate generator** block: Used to generate sample baudrate clocks for the TX and RX transmission.
- **TX controller** block: Used to control the TX transmission state.
- **RX controller** block: Used to control the RX transmission state.

4.8 Direct Memory Access

The Direct Memory Access (DMA) controller is provided to move the image or sub-window to two modules: the Image processor and the Display Interface. Therefore, the DMA consists of configurable 2 channels. We provide the interleaving weighted round-robin arbitration mechanism between two channels, so we can decide the throughput of each channel (or each module) by configuring. The processor can configure the parameters, such as the mode of the DMA, the

arbitration weight for each channel, the source/destination address, the interrupt masks, etc. The block diagram of the DMA is illustrated in the figure 4.34.

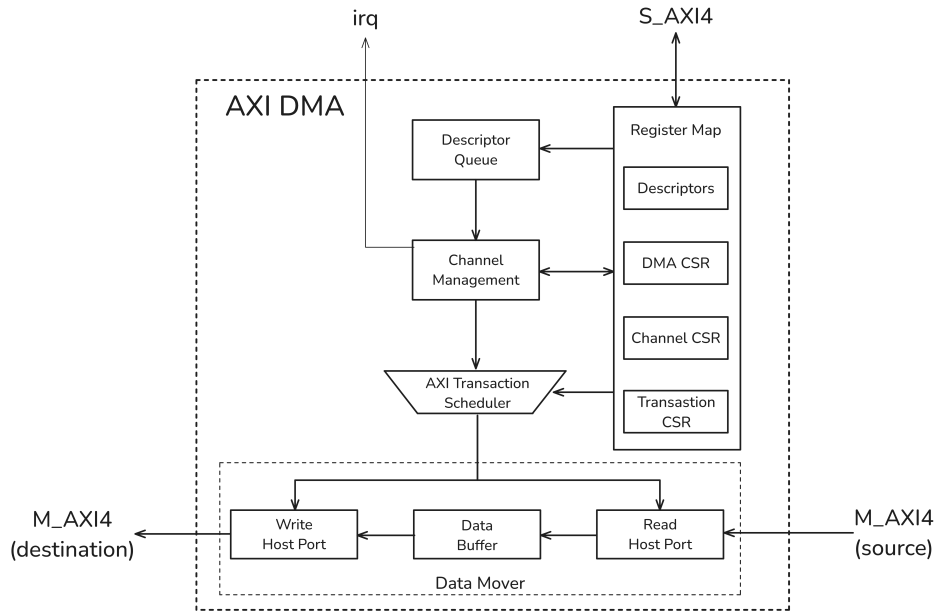


Figure 4.34: Direct Memory Access architecture

There are 3 main interfaces in the DMA:

- AXI4 slave interface: The AXI master can configure parameters or monitor the state of the DMA via this interface
- AXI4 master interface (source): Used this interface to read the data from the source memory (the image memory).
- AXI-Stream master interface (destination): Used this interface to write the buffered data to the destination memory (the image processor and the display interface).

There are 5 main components in the DMA:

- **Register map** block: Consists of control registers, status registers of the DMA. The processor can change the streaming type, the interrupt masks, etc, via the control register or monitor the state of the DMA via the status register. There are two channels corresponding two module (the image processor and the display module), therefore, we provide two address regions

for the two channels. The base address of the DMA is 0x8000_0000. To calculate the address of a register of the corresponding channel, we use the formula below:

$register\_addr = dma\_base\_addr + (channe\_id << 4) + register\_offset$

More details about all registers in figure 4.35.

Register name	Register's offset	Channel's offset	Bits	Description	Type
Channel CSRs					
CHN_CONTROL	0x0001	chn_id << 4		The control signal of channel	RW
			[0]	ENABLE	
			[31:1]	Reserved	
CHN_FLAGS	0x0002	chn_id << 4			
			[0]	TRANSFER_2D	
			[1]	TRANSFER_CYCLIC : A cyclic transfer once completed will restart automatically with the same configuration.	
CHN_IRQ_MASK	0x0003	chn_id << 4			RW
			[0]	when a TRANSFER_COMPLETED , the interrupt request will be asserted	
			[1]	when a TRANSFER_QUEUED , the interrupt request will be assert	
			[31:2]	Reserved	
CHN_IRQ_SOURCE	0x2000	chn_id << 4			RO
			[0]	TRANSFER_COMPLETED	
			[1]	TRANSFER_QUEUED	
			[31:2]	Reserved	
CHN_ARBIT_RATE	0x0004	chn_id << 4		The arbitration rate of the channel (must be different from 0 when the channel is enabled)	RW
Descriptor					
SRC_ADDR	0x0009	chn_id << 4		The source address	RW
DST_TDEST	0x000A	chn_id << 4		The destination of the transfer in the AXI-Stream interface (TDEST bus value)	RW
TRANSFER_SUBMIT	0x1000	chn_id << 4		The fifo write enable signal	RW1S
TRANSFER_X_LEN	0x000B	chn_id << 4		The number of words in a row + 1 (same as transfer length in 1-D transfer mode)	RW
TRANSFER_Y_LEN	0x000C	chn_id << 4		The number of rows + 1(equal 0 when DMA in 1-D transfer mode)	RW
SRC_STRIDE	0x000D	chn_id << 4		The number of words between the start of one row and the next row for the source address. Needs to be aligned to the bus width. Note, this field is only valid if the DMA channel has been configured with TRANSFER_2D and read from memory support.	RW
TRANSFER_ID	0x2001	chn_id << 4		The next transfer's ID	RO

Figure 4.35: Register table of the DMA

- **Descriptor queue** block: The processor creates a transfer (known as a descriptor) and pushes it to the descriptor queue. A transfer (or a descriptor) contains information such as the source address, the destination address, the width or height of the interest image, etc.
- **Channel management** block: Used to convert a descriptor to multiple AXI transactions which have consecutive addresses. Then, the block submits the AXI transaction of the corresponding channel to the AXI transaction

scheduler block. The figure 4.36 illustrates the Channel management block.

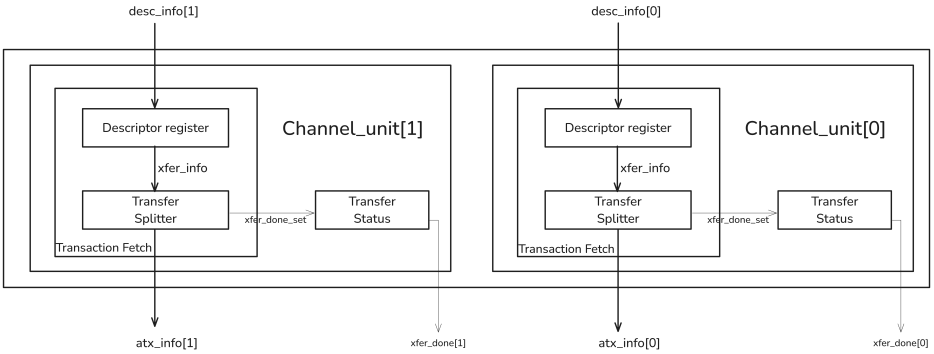


Figure 4.36: The block diagram of the channel management block

- **AXI transaction scheduler**: Receive the AXI transactions of 2 channels from the Channel management block and arbitrate between two channels, and each grant is 1 AXI transaction. Then, send the granted AXI transaction to the Data mover block. We use interleaving weight round-robin arbitration mechanism for the arbiter. The figure 4.37 illustrates the AXI transaction scheduler block.

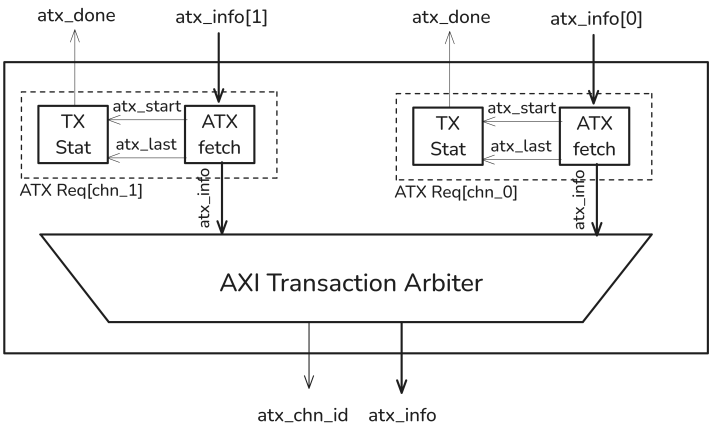


Figure 4.37: The AXI Transaction Scheduler block

- **Data mover** block: After receiving the granted AXI transaction, the Data mover generates multiple AXI transfers based on the transaction to move the data from the source memory to the destination memory.

4.9 Interconnect

The Interconnect is used to connect AXI components via the AXI4 protocol, which supports the AXI master to route the read/write transactions to the corresponding AXI slave based on the memory-mapping mechanism. We provide the AXI4 Interconnect to integrate all components to the processor. In the configuration subsystem, there are 1 AXI master and 6 AXI slaves. The master-slave diagram of the system is shown in the figure 4.38, where the red block represents the master and the blue blocks represent slaves.

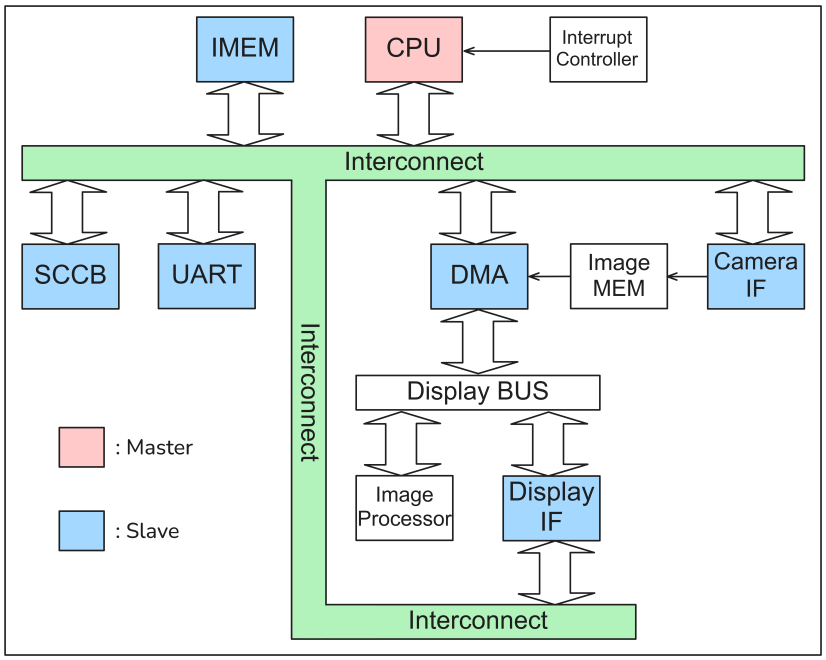


Figure 4.38: The master-slave diagram

The AXI4 interconnect we designed is of the single-master, multiple-slaves type, and the processor (AXI master) supports single-burst transactions. Therefore, the interconnect only needs to route the AXI transaction to the corresponding slave and supports AXI interleaving depth of 1. The block diagram of the AXI interconnect is shown in the figure 4.39.

There are 2 types of interface in the interconnect:

- AXI master interface: used to connect to the master in the system
- AXI slave interface: used to connect to the slaves in the system

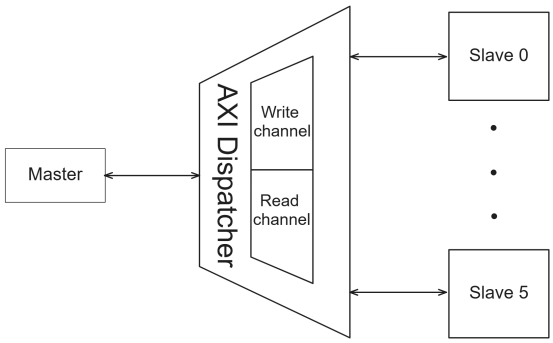


Figure 4.39: The block diagram of the AXI interconnect

There are 2 main components in the interconnect:

- **Write channel** block: Used to handle write transactions, which include three channels: AW channel, W channel, and B channel. In the AW and W channels, only one AXI master issues the transfers, so the interconnect only needs to route the AW and W transfers. However, in the B channel, multiple AXI slaves respond to the interconnect. Therefore, the interconnect needs to reorder the B transfers from the slaves based on the order of the AW transfers. The block diagram of the Write channel is shown in the figure 4.40

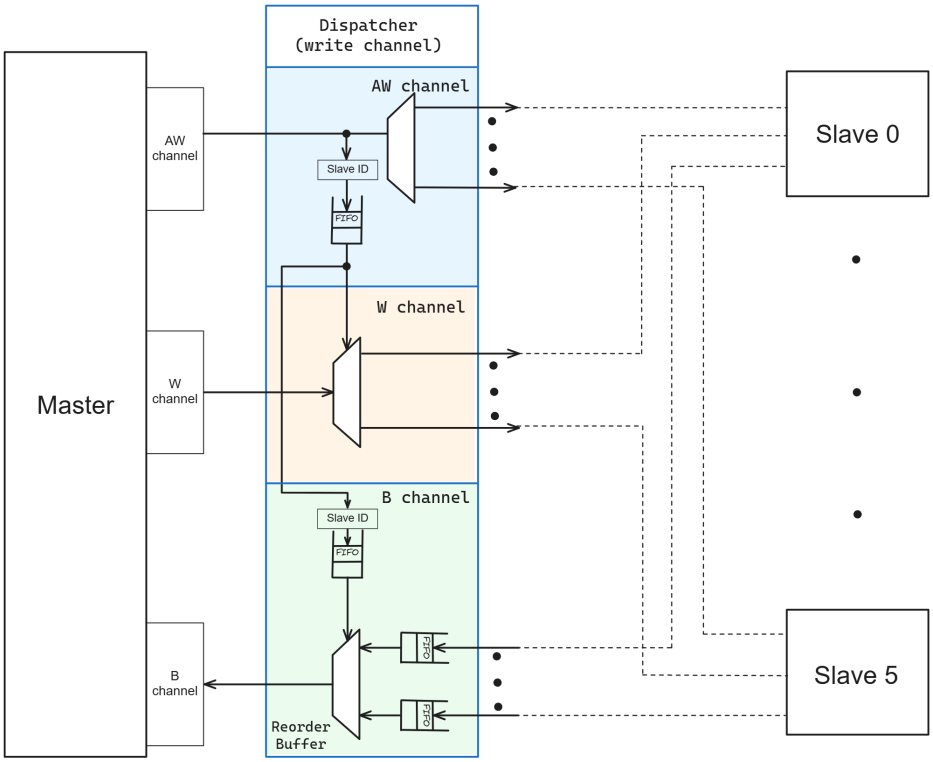


Figure 4.40: The write channel block

- **Read channel** block: Similar to the Write channel, the orders of transactions are based on the AR transfers, and the out-of-order event can occur in the R channel. Therefore, the interconnect must handle out-of-order R transfers from multiple slaves and ensure that all R transfers are reordered to match the order of AR transfers when they are returned to the AXI master. The block diagram of the Read channel is illustrated in the figure 4.41

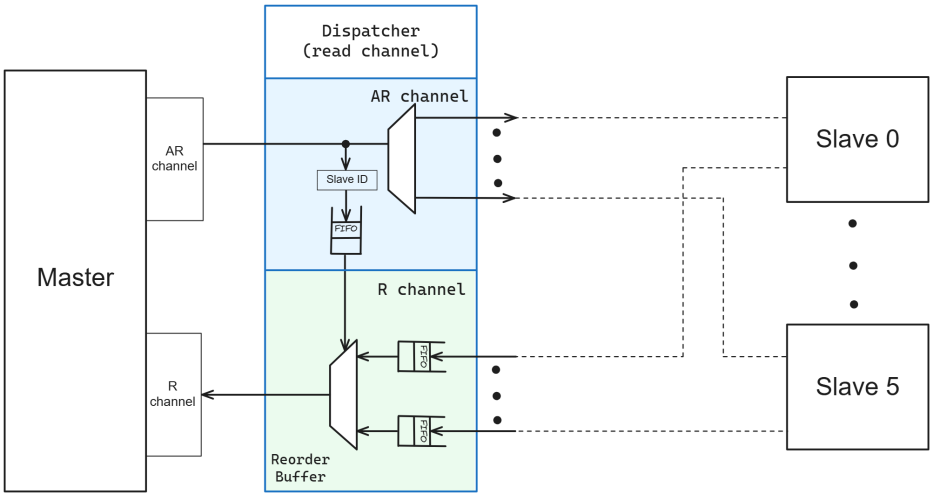


Figure 4.41: The read channel block

We provide the document detailing the I/Os, mapping, and some special mechanisms here.

4.10 Firmware implementation

In addition to programming the system, we provide 3 programs: a bootloader program, an interrupt service routine (ISR) program, and a main program. The bootloader region range is from 0x0000_0000 to 0x0000_FFFF, the ISR region is from 0x0001_0000 to 0x0001_FFFF, and the Main program region is from 0x0002_0000 to 0x0002_FFFF. The figure 4.42 illustates the memory region of each program

The software flow of the system is separated into 2 phases: the loading program and the execution phases (shown in the figure 4.43). Firstly, the system

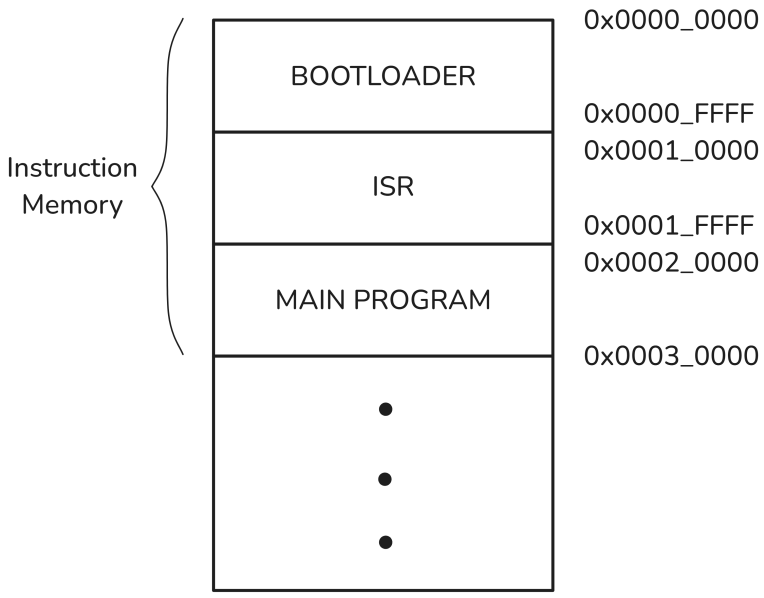


Figure 4.42: The memory regions in the instruction memory

performs the loading program phase to set up the data in the memory and prepare for the execution phase. Then, the system is changed to the execution phase to perform the main application of the system.

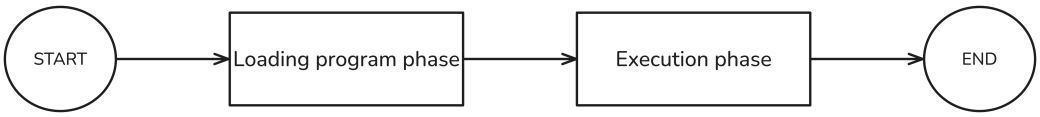


Figure 4.43: The overview flow of the system's firmware

The system runs the bootloader program in the loading program phase, and the main program and the ISR program in the execution phase.

Each program consists of RV32 instructions, and their function will be discussed in the sections below.

4.10.1 The bootloader program

The bootloader program is initialized in the system's ROM. It is used for the user to program the system via the UART interface. Firstly, the program configures the UART controller and receives the command via the UART interface. The program then parses the received command and performs the operation based on the command. the figure 4.44 shows the format of the commands.

Each command frame consists of two data items: a 32-bit address and a

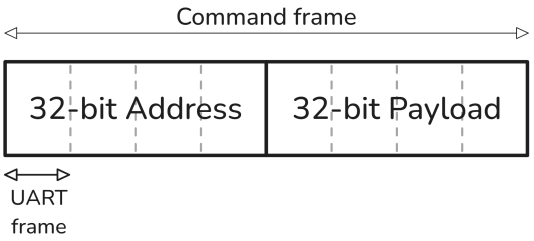


Figure 4.44: The format of the commands

32-bit payload. The program parses the received command and stores the command's payload in the corresponding address, which is the first data item of the command.

To exit the bootloader program, we provide the End-of-Programming command, which has both the address item and the data item equal to 0xFFFF_FFFF. After receiving the End-of-Programming command, the program jumps into the main program region, which is placed at 0x0001_0000.

The figure 4.45 shows the flowchart of the bootloader.

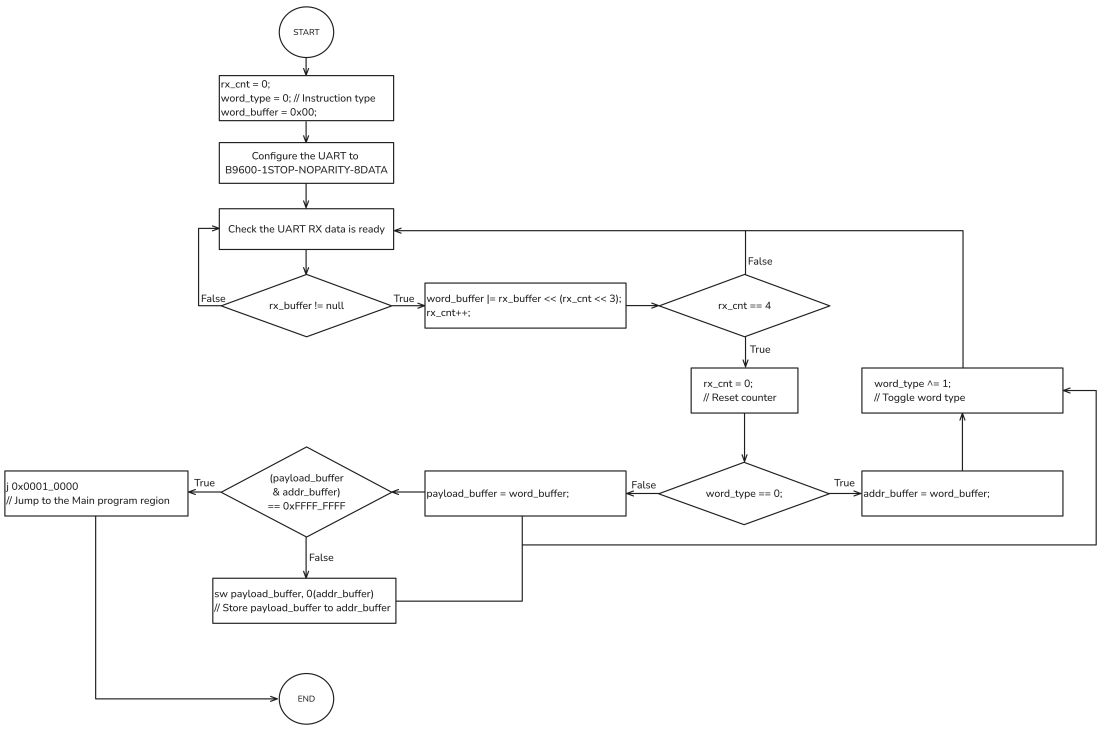


Figure 4.45: The Flowchart of the bootloader program

4.10.2 The ISR program

The interrupt service routine (ISR) program can be loaded in the loading program phase. The program is used to handle some interrupt events, and there are 4 main interrupt sources: the frame captured interrupt, the frame stored interrupt, the frame transferred-to-display interrupt, and the frame transferred-to-image-processor interrupt. However, we only use 1 interrupt source in this application, that is the frame stored interrupt.

When the frame stored interrupt event occurs, the ISR program starts the DMA to transfer the frame, which has been stored, to the 2 destinations: the display interface and the image processor. The figure 4.46 illustrates the flowchart of the ISR program.

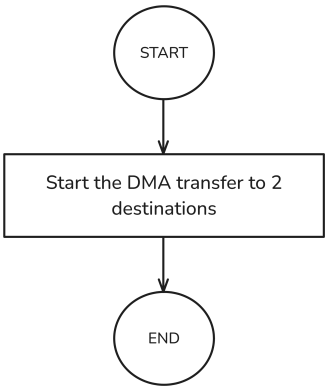


Figure 4.46: The flowchart of the ISR program

4.10.3 The main program

The main program is loaded during the loading program phase. The program is used to set up the interrupt masks, configure internal components, communicate with the external components, and control the data flow of the system. There are 4 internal components, which are configured in the main program. The flowchart of the main program is shown in the figure 4.47.

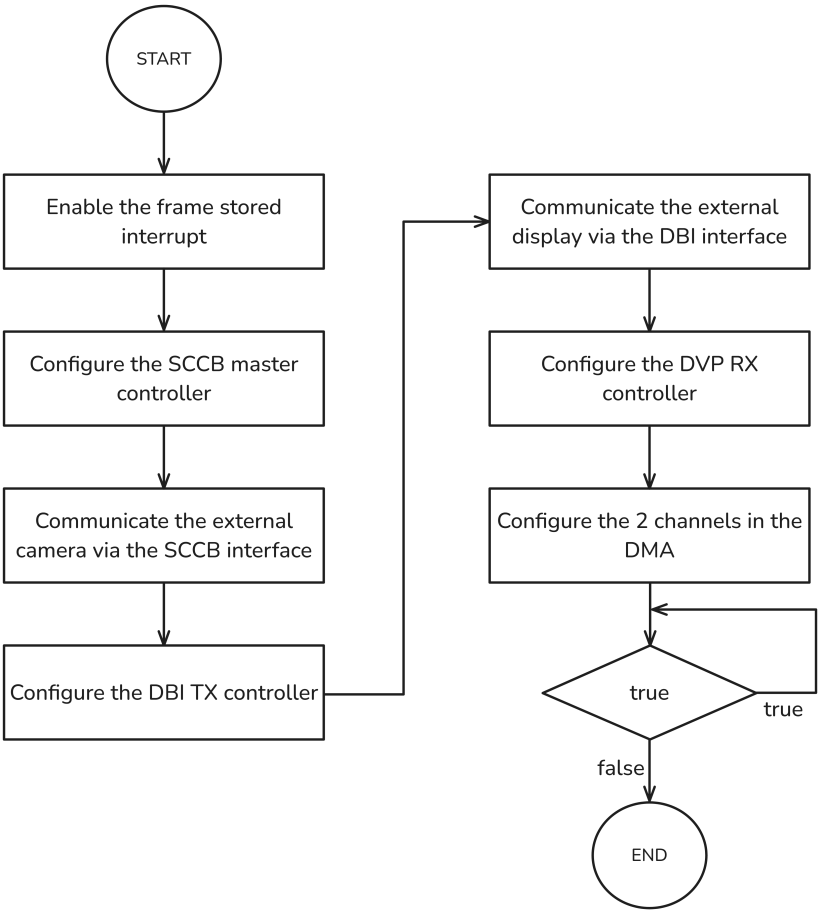


Figure 4.47: The flowchart of the main program

Chapter 5

Functional verification

The Histogram of Oriented Gradients (HOG) is a widely used algorithm in computer vision, especially for applications such as human detection. It extracts image features through steps like gradient magnitude, orientation, bin voting, and normalization. Implementing HOG in hardware requires verifying its correctness against a reference model to ensure accurate operation.

In addition, Direct Memory Access (DMA) plays a key role in efficient data transfer between memory and peripherals without burdening the processor. Verifying the DMA controller ensures that data is transferred accurately.

This chapter presents a functional verification approach using Universal Verification Methodology (UVM) for key hardware blocks such as HOG and DMA. The verification process is based on a Python reference model, and the goal is to ensure that all processing steps (magnitude, orientation, voting, and normalization) are implemented correctly and produce consistent results with the software model.

Furthermore, this chapter also presents the results of full system verification, covering the interaction between multiple hardware components including the HOG module, DMA controller, control processor, and memory. The objective is to confirm that the entire system functions as intended under different scenarios, meets design specifications, and operates reliably without errors.

5.1 HOG & SVM module verification

5.1.1 Functional verification result

Throughout the design and implementation process, we verified testing at the following levels:

Test	Level	Description	Result
Normalize	Unit test	Check result after normalize input feature raw, output feature after normalize	Pass
SVM	Unit test	Check result after calculate with coeficient svm, input normalized feature, output result after svm calculated	Pass
Hog	Integration test	Check result after extact feature input image, output extracted feature	Pass
Hog to SVM	Integration test	Check result after extract feature and calulate with svm_coef, input image output result after svm calculated	Pass

Table 5.1: Test list image processor

In the section below we will present the test hog to svm in detail (highest integration test of 4 test in table 5.1).

5.1.1.1 UVM environment description

Our uvm testbench architecture include:

Design under test (DUT); is the object that needs to be verified (HOG to SVM block) it will receive the input signal from the driver and return the output to the monitor to send to the scoreboard for comparison and evaluation.

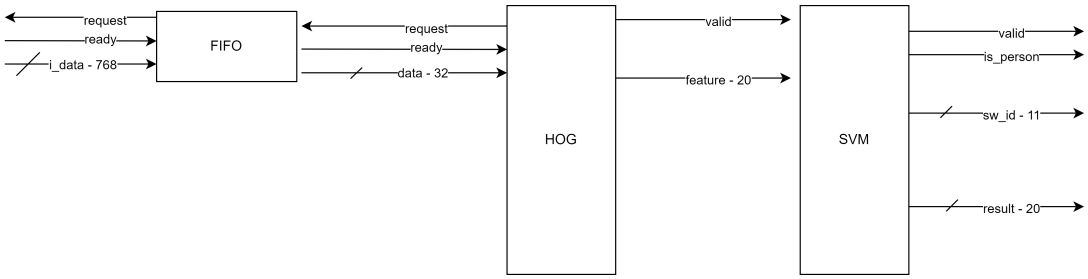


Figure 5.1: Design under test

Item is an object to describe transactions (input output) during the test, so class item in our testbench will include some parameter as image below:

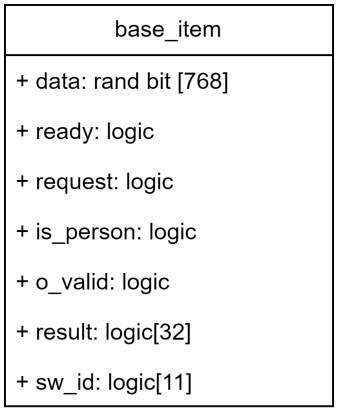


Figure 5.2: Our item

Sequence is where transactions are created to send to the driver so that the driver can drive the signal to dut. In the sequence, 3600 items will be created to send to the driver representing 3 images (each image has 1200 items because the size is 320x240 after scaling 1:2, 1 item is 1 cell (8x8 pixels) and 4 borders of adjacent cells).

Driver is where data is driven into DUT, it will work as follows if request and reset signals are asserted drive ready to 1 and drive item.data into dut's virtual interface when ready and request are asserted.

Monitor is where the signals returned from the DUT through the virtual interface will be captured and sent to the scoreboard for comparison with the golden model. It will work as follow, when o_valid is asserted, it will capture is_person, result, sw_id and send to scoreboard for comparision.

Scoreboard is where the golden model is built to compare with the data captured from the monitor to test the functionality of the HOG to SVM block. The idea of the scoreboard is to wait until there is enough small 64x128 image to perform feature extraction and calculate with the weight of svm and save that answer, then in the extract phase it will compare with the captured result from the monitor.

5.1.1.2 Test with randomized data

Test: As mentioned above, to verify functionality we will send 3 images with random data (using randomization function) then compare the captured result from monitor and the golden model build result on scoreboard.

Result: For each 320x240 image, there will be a total of 495 64x128 images. So when sending 3 320x240 images, our expectation is that we will capture 1485 (495 * 3) results from the monitor. And we will compare the results from the monitor with the golden model with a deviation of 0.01.

```
156841  # Golden size: 1485
156842  # Mon size: 1485
156843  #         0
156844  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
156845  #           1
156846  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
156847  #           2
156848  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
156849  #           3
156850  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
156851  #           4
156852  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
156853  #           5
```

Figure 5.3: Total result build in scoreboard and captured from monitor

Looking at the image above, we actually captured 1485 results from the monitor and in the scoreboard we also built 1485 results corresponding to 3 photos, this shows that it meets our expectation.

The image below shows that the test passes, meaning that the results captured from the monitor and the build results from the scoreboard have a deviation of no more than 0.01.

```
159810  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
159811  #        1484
159812  # UVM_INFO ../uvc/env/base_scoreboard.sv(463) @ 14504000: uvm_test_top.bus_env.scoreboard0 [base_scoreboard] Result PASS
159813  # UVM_INFO ../uvc/tests/base_test.sv(48) @ 14504000: uvm_test_top [base_test] ** TEST PASSED **
159814  # UVM_INFO C:/questasim64_2024.1/verilog_src/uvm-1.2/src/base/uvm_report_server.svh(847) @ 14504000: reporter [UVM/REPORT/SERVER]
159815  # --- UVM Report Summary ---
159816  #
159817  # ** Report counts by severity
159818  # UVM_INFO :13569
159819  # UVM_WARNING :    0
159820  # UVM_ERROR :    0
159821  # UVM_FATAL :    0
159822  # ** Report counts by id
159823  # [Questa UVM]     2
159824  # [RNTST]     1
159825  # [TEST_DONE]     1
159826  # [UVMTOP]     1
159827  # [base_scoreboard]  9963
159828  # [base_test]     1
159829  # [run_phase]  3600
159830  #
159831  # ** Note: $finish    : C:/questasim64_2024.1/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
159832  #    Time: 14504 ns  Iteration: 68  Instance: /top
```

Figure 5.4: UVM report

First the system will wait for the reset signal. When the reset is successful it

will start driving the signal into the DUT at each positive clock. When enough cells have been received to form a block the o_valid signal of the hog will be asserted. In addition, when there is enough 64x128 frame the o_valid signal of the svm will be asserted and then we will check the is_person signal to see if there is a person or not. We will see the system behavior waveform as shown in the figure 5.5:

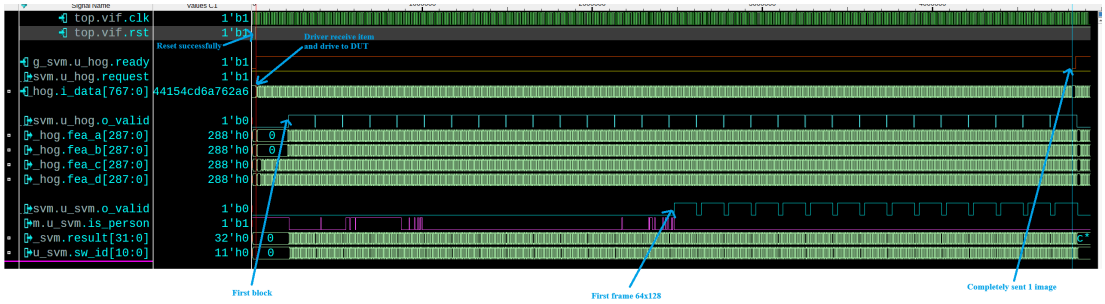


Figure 5.5: First image waveform

Finally, when sending enough 1 image the ready signal of the fifo will be de-asserted and wait until the fifo is ready to send the second image. This behaviour is shown as waveform in the figure 5.6:

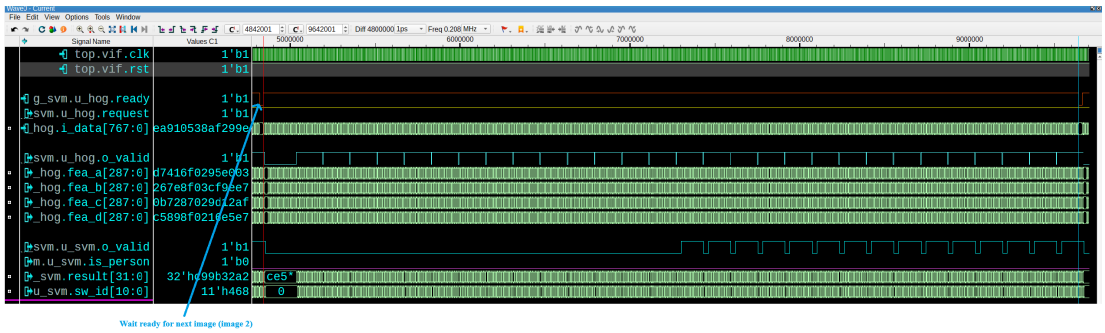


Figure 5.6: Second image waveform

5.1.2 Test with images without people

In this test, we will take 2 pictures without people, convert them into hex and upload them to the system and compare the results obtained from the design system with the results returned from python.

The figure 5.7 includes a photo not having people and the gray-scale:

Figure 5.7: Image without person 1

Then from the gray-scale image we convert it to hex format and load it into the designed system to compare the results with the system implemented on python. And below are the results obtained from the system implemented on python and the system implemented on system verilog:

In general, we can see that the results obtained from the system on python and the system implemented in system verilog give relatively accurate results. The total number of people detected by the image in this case is 0. The result is the image below:

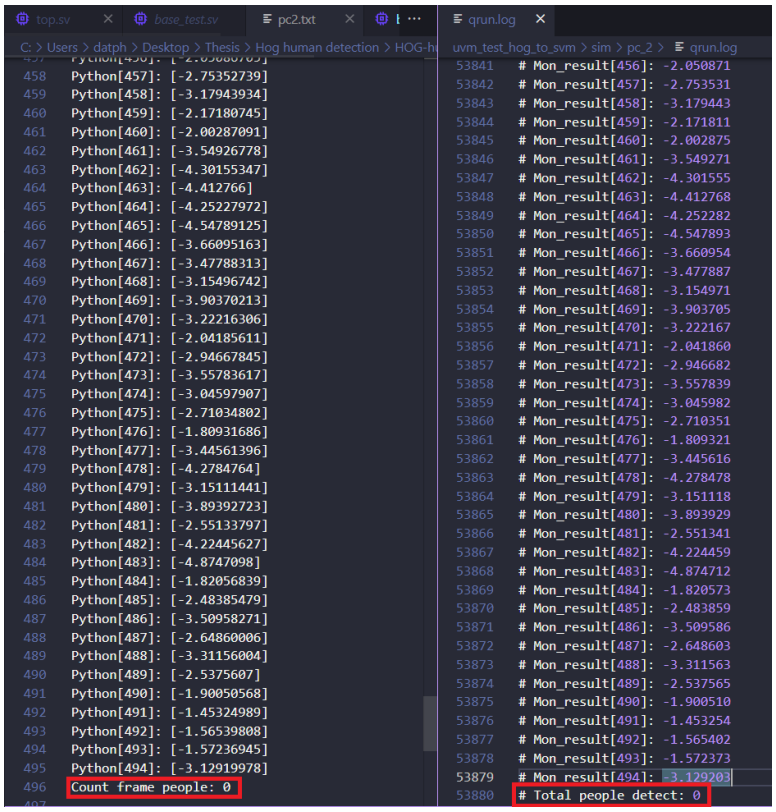


Figure 5.8: Log file python and verilog 1

The figure 5.9 is another image without people and its gray-scale

Figure 5.9: Image without person 2

Here is the result of it:

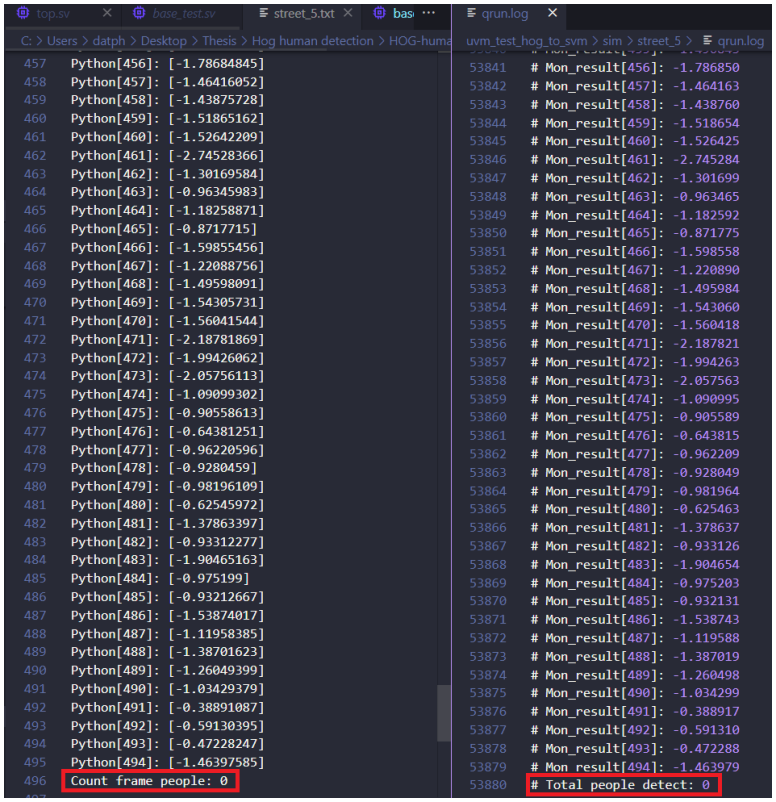


Figure 5.10: Log file python and verilog 2

5.1.3 Test with images having people

In this test, we will take 2 pictures people, convert them into hex and upload them to the system and compare the results obtained from the design system with the results returned from python.

The figure 5.11 includes photo with people and it's gray-scale:

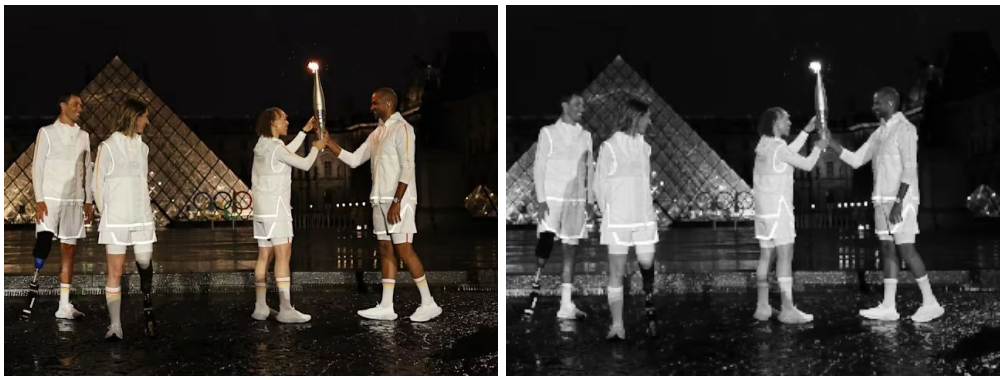


Figure 5.11: Image with people 1

After detection we detect 18 people (overlap not processed) and the result is shown as the figure 5.12:

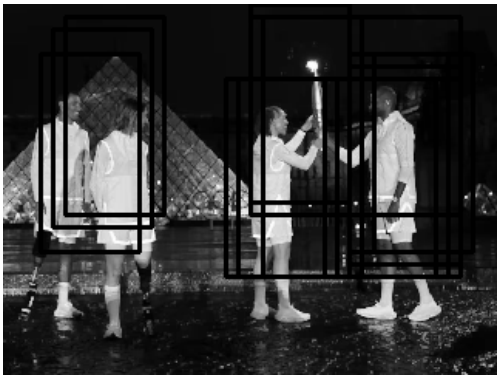


Figure 5.12: Image with people 1 detected

The waveform 5.13 show how the system detect people (when o_valid and is_person are asserted)

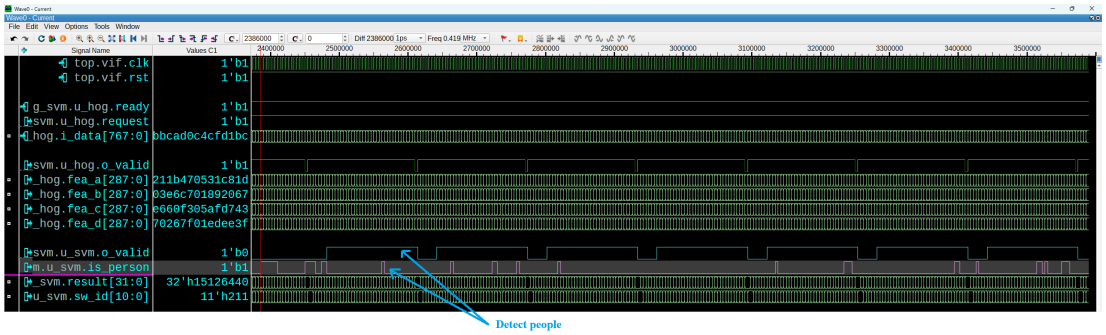


Figure 5.13: Waveform of image with people 1

The log result in both python and verilog is shown as 5.14:

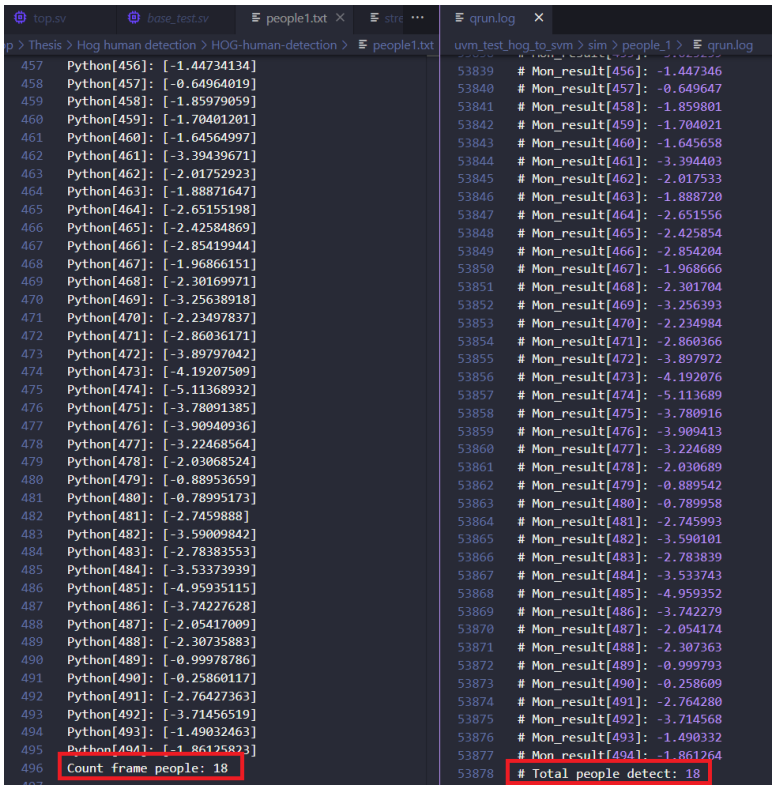


Figure 5.14: Log file python and verilog 3

The figure 5.15 is another image without people and its gray-scale:

Figure 5.15: Image with people 2

After detection, we detect 8 people (overlap not processed) the result is shown as the figure 5.16

Figure 5.16: Image with people 2

This waveform describes system behavior when detect people is as below:

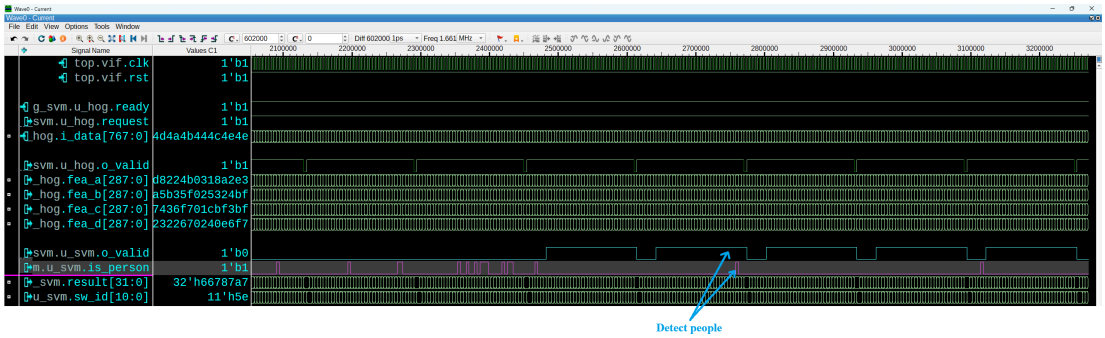


Figure 5.17: Waveform of image with people 2

The log result in python and verilog is shown as the figure 5.18

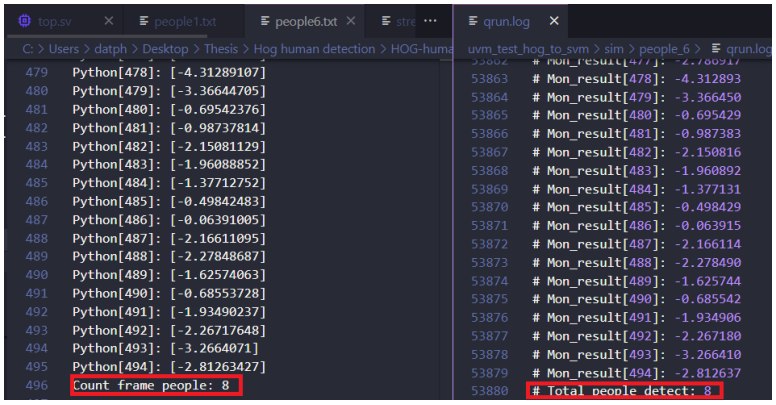


Figure 5.18: Log file python and verilog 4

5.2 DMA module verification

Throughout the design and implementation process, we verified testing at the following modes:

Mode	Description	Result
1-channel cyclic	Verify memory moved correctly	Pass
1-channel 2d transfer	Verify memory moved correctly	Pass
1-channel cyclic & 2d transfer	Verify memory moved correctly	Pass
2-channel cyclic & 2d transfer	Verify memory moved correctly	Pass

Table 5.2: Test list DMA

In this section below, we will present the test of mode 2-channel cyclic & 2d transfer, which is the main mode we use in this project.

5.2.1 UVM enviroment description

Our uvm testbench architecture include:

Design under test (DUT); is the object that needs to be verified (DMA block) it will receive data to config register through axi slave. When config phase is done, it will transfer data (axi master) from source to destination correctly about address and data.

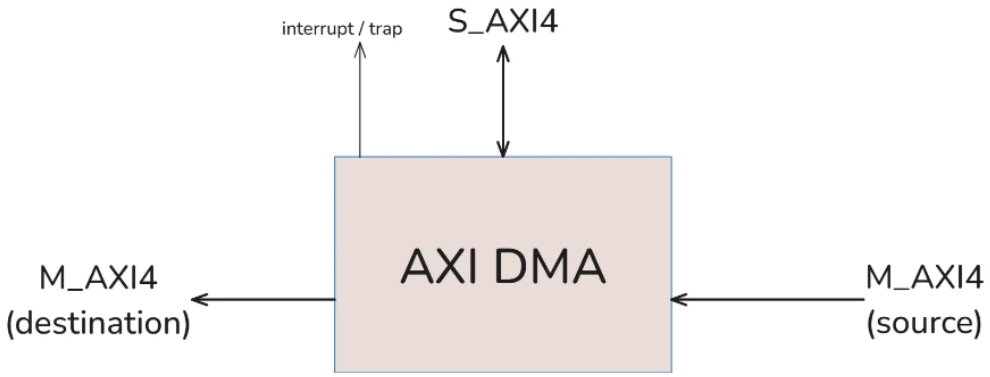


Figure 5.19: Design under test

Item is an object to describe transactions (input output) during the test, so class item in our testbench will include some parameter as image below:

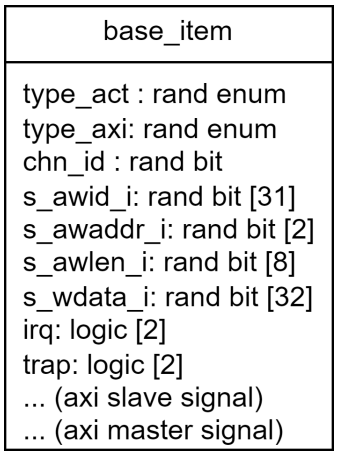


Figure 5.20: Our item

Sequence: creating transaction of axi slave to config register for working mode. Sequence includes type of act (read or write), type of axi (slave or master), channel id (id = 0 for channel 1, id = 1 for channel 2), address of configured register, burst size, length of transaction and data.

Driver is where register is configured, it will check when address channel handshake successfully. After that, it will drive data to correct address (data channel) and wait response (response channel). Additionnally, driver also acts as an axi master so that when there is a transaction (write data) from source to destination. it will get data in source address and send it to destination address.

Monitor is where monitoring data go through destination channel. It will check when address channel (destination channel) handshake successfully, it will capture some info includes (awid, awaddr, awlen, awburst, and data), and send it to scoreboard for comparision.

Scoreboard is where the golden is built to compare whether the data sent from the source channel is going to the correct destination channel. Also, check whether the sending address (source and destination) is correct compared to the configured register. Also, check that modes like cyclic and 2d transfer work as expected.

5.2.2 Test result (randomized data)

Test: To verify functionality we will config working register mode is cyclic & 2d-transfer, randomize xlen, ylen, write_data_per_burst. Then, compare the monitored result with golden model.

Result: Test as image 5.21 that after comparing monitored result and golden model give the same result => test pass

```
# PASS
# PASS
# PASS
# PASS
# PASS
# UVM_INFO ../uvc/tests/base_test.sv(84) @ 156639000: uvm_test_top [base_test] ** TEST PASSED **
# UVM_INFO C:/questasim64_2024.1/verilog_src/uvm-1.2/src/base/uvm_report_server.svh(847) @ 156639000: reporter [UVM/REPORT/SERVER]
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 1130
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Handshake successfully slave]    33
# [MEM ACTUAL CHANNEL 1 EXTRACT_PHASE]     1
# [MEM ACTUAL CHANNEL 2 EXTRACT_PHASE]     1
# [MEM EXPECTED CHANNEL 1 EXTRACT_PHASE]     1
# [MEM EXPECTED CHANNEL 2 EXTRACT_PHASE]     1
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# [base_scoreboard]  1054
# [base_test]     1
# [run_phase]    33
```

Figure 5.21: UVM enviroment for DMA result

Firstly, we will config register for working mode as image 5.22:

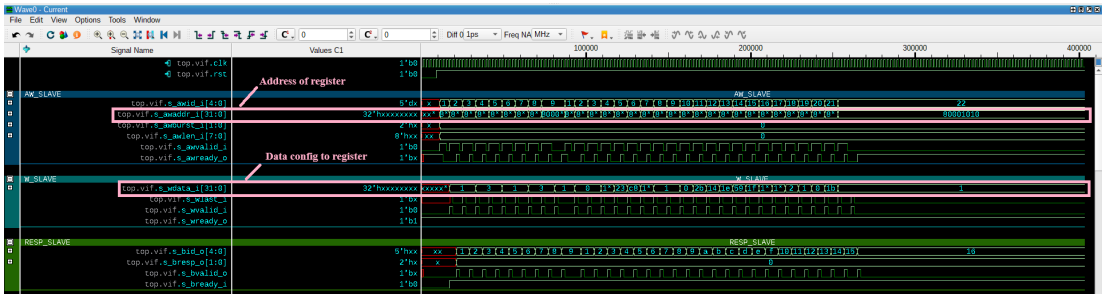


Figure 5.22: Waveform configure register for working mode

Secondly, when configuring succesfully, dma will move data from source address to destination address which is configured in configure stage as image below 5.23:

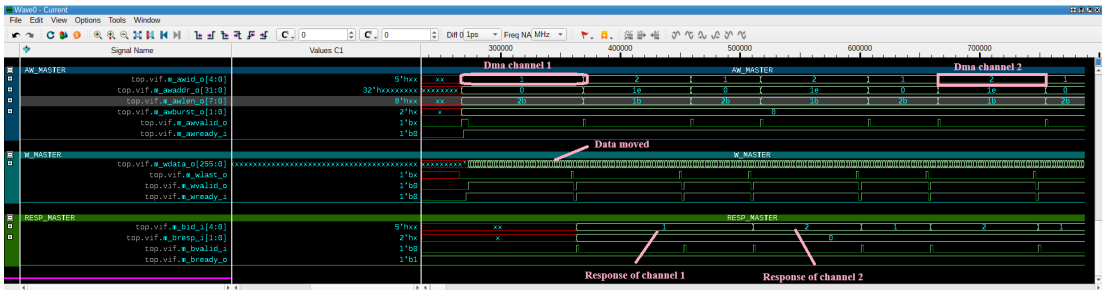


Figure 5.23: Waveform move mem dma

5.3 System verification

5.3.1 Description

We provide the functional verification environment for the entire system. The figure 5.24 show the system's verification environment.

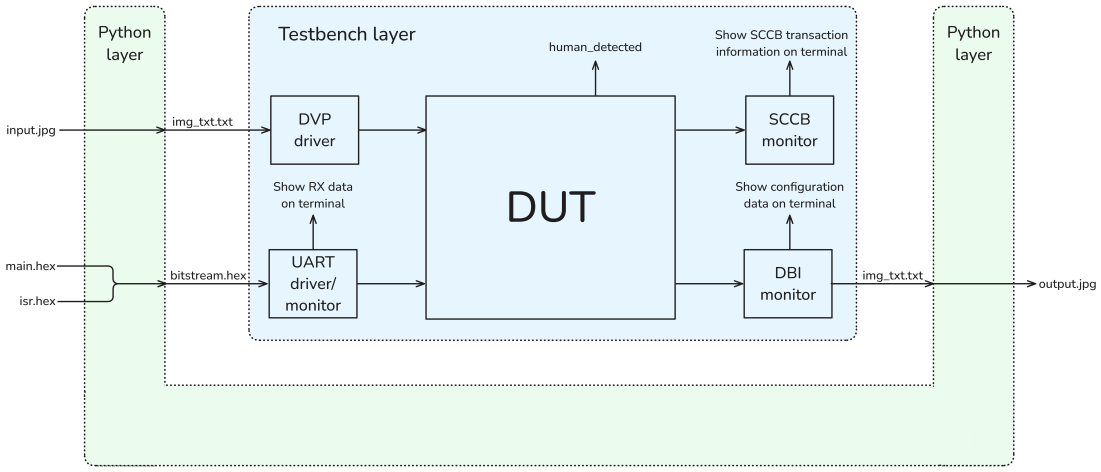


Figure 5.24: The verification environment of the system

To verify the system, we used Python to prepare the image data by converting the image from an image file (.png file) to a hexadecimal text file (.txt file), and convert the image hexadecimal data, which is monitored from the display interface to an image file (.png file).

Then, we used Testbench to drive the prepared data from Python into the system (DUT block in the figure 5.24), and monitor the output from the DUT such as SCCB transactions, and display data.

Firstly, we generate the bitstream file from the main program machine code and the ISR program machine code. The bitstream file's format is shown in the

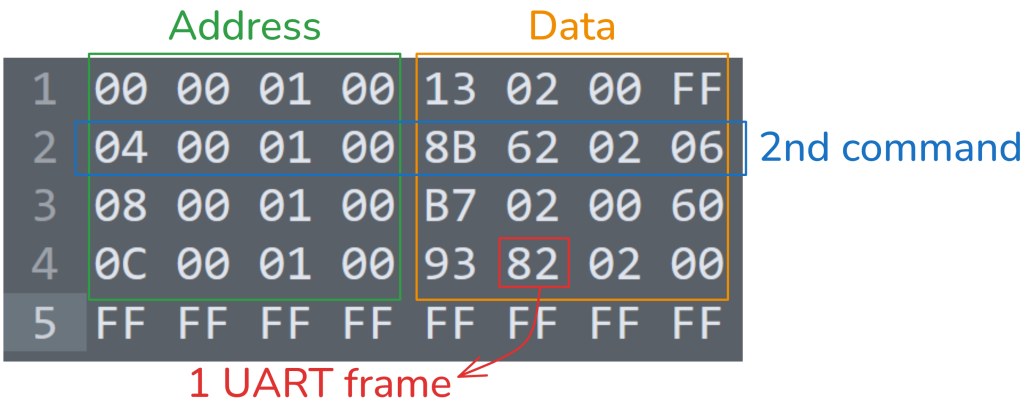


Figure 5.25: The bitstream file's format

figure 5.25. After that, the bitstream.hex file is loaded into the UART driver in the testbench to program the system. More detail about processing the bitstream file is explained in the section 4.10.1

Secondly, we convert the input image file into a text file and load the text into the DVP driver to stream the image data to the system via the camera interface. The figure 5.26 illustrates this step.

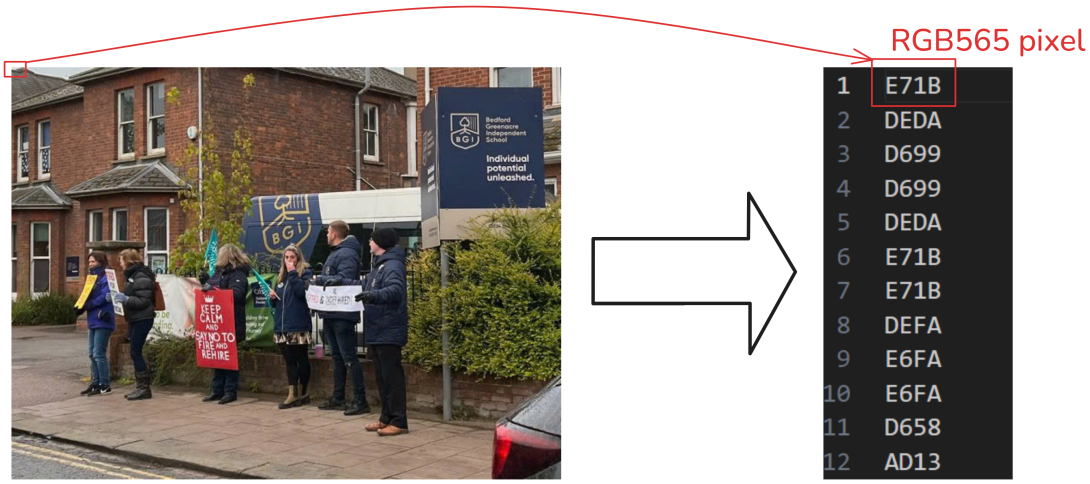


Figure 5.26: The image to text converting

Thirdly, after streaming the entire image into the DUT, wait until the testbench is finished, and use Python to collect the image data from the display interface to convert it to an image file. The figure 5.27 illustrates this step.

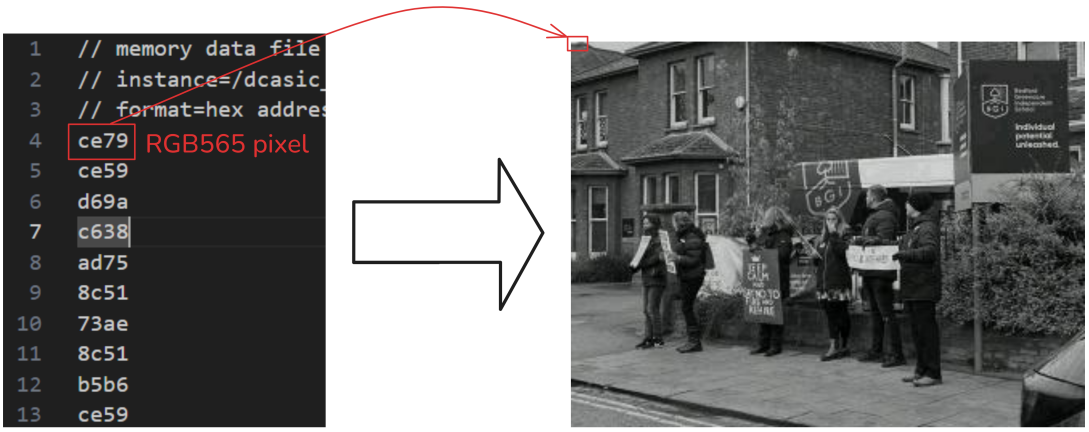


Figure 5.27: The text to image converting

5.3.2 Result

5.3.2.1 Programming phase

To verify the programming phase in the testbench, we use the UART monitor to check the programming's progress and print it to a log file and the terminal. The figure 5.28 shows the terminal result.

```
# [INFO]: Programming DCASIC .... (99.772123%)
# [INFO]: UART RX driver receive 0xff
# [INFO]: Programming DCASIC .... (99.810103%)
# [INFO]: UART RX driver receive 0xff
# [INFO]: Programming DCASIC .... (99.848082%)
# [INFO]: UART RX driver receive 0xff
# [INFO]: Programming DCASIC .... (99.886062%)
# [INFO]: UART RX driver receive 0xff
# [INFO]: Programming DCASIC .... (99.924041%)
# [INFO]: SCCB write transaction with (SLV_ADDR
# [INFO]: UART RX driver receive 0xff
# [INFO]: Programming DCASIC .... (99.962021%)
# [INFO]: Programming done
```

Figure 5.28: The programming phase log

5.3.2.2 Camera configuration phase

To configure the external OV7670 camera, we use the SCCB interface. Therefore, we also verify the SCCB transmission information to make sure that all

configuration transmissions are correct. We implement the SCCB monitor to print all SCCB transmissions to a log file and the terminal. The figure 5.29 shows the terminal result of the camera configuration phase.

```
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 12 - 04)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 11 - c0)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 40 - d0)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 3d - 81)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 6f - 9f)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 13 - e5)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - aa - 94)
# [INFO]: SCCB write transaction with (SLV_ADDR - SUB_ADDR - DATA) (46 - 14 - 18)
```

Figure 5.29: The camera configuration log via the SCCB interface

5.3.2.3 Display configuration phase

Before streaming out the image in the system via the DBI interface, the display LCD should be configured with some parameters such as the image width, image height, start column, start row, etc. Therefore, we monitor the configuration phase of the DBI transmission by the DBI monitor block. The figure 5.30 shows the result of the configuration phase of DBI transmissions.

```
# [INFO]: DBI Command:  0x01
# [INFO]: DBI Command:  0x36
# [INFO]: DBI Data:     0x20
# [INFO]: DBI Command:  0x3a
# [INFO]: DBI Data:     0x55
# [INFO]: DBI Command:  0x2a
# [INFO]: DBI Data:     0x00
# [INFO]: DBI Data:     0x00
# [INFO]: DBI Data:     0x01
# [INFO]: DBI Data:     0x3f
# [INFO]: DBI Command:  0x2b
# [INFO]: DBI Data:     0x00
# [INFO]: DBI Data:     0x00
# [INFO]: DBI Data:     0x00
# [INFO]: DBI Data:     0xef
# [INFO]: DBI Command:  0x11
```

Figure 5.30: Display configuration phase

5.3.2.4 Image displaying phase

When the DBI monitor receives an entire image, it writes the received image data into the img_txt.txt file. After that, we use Python to convert the img_txt.txt file to a image file (.jpg file). The figure 5.31 shows the output.jpg file

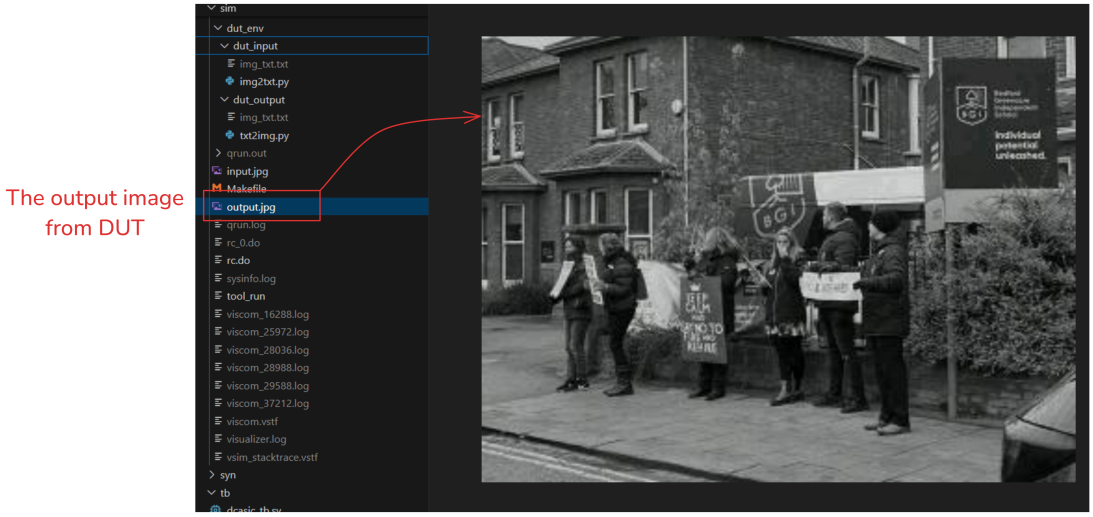


Figure 5.31: The output from the display interface

Chapter 6

Experimental results

This chapter presents the experimental results of the proposed design, including synthesis, FPGA prototype, and GDSII outcomes. Synthesis results provide early estimates of area, timing, and power based on RTL descriptions, offering an initial evaluation of design efficiency. FPGA prototype results demonstrate functional correctness and real-time performance through hardware implementation and testing. Finally, GDSII results detail the post-layout characteristics such as finalized area, timing closure, and power consumption, verifying the design's compliance with specifications at the physical level. Together, these results validate the design across all stages from high-level modeling to silicon-ready implementation.

6.1 Image processor evaluation

6.1.1 Synthesis results

The number of cycle counts was estimated using a Verilog-HDL simulator. With totally pipeline architecture, we can reach up to **76,832** cycles/frame in *image processor module* (frame rate) in the resolution of 320 x 240 pixels.

To evaluate the effectiveness of our approach, we synthesized the proposed architecture onto AMD Vivado Design Suite, the design software for AMD

adaptive SoCs and FPGAs for FPGA implementation. Morerover, we also synthesized our SVM module on the Openlane platform, an automated RTL to GDSII flow based on several components including OpenROAD, Yosys, Magic, Netgen, CVC, SPEF-Extractor and KLayout with *130nm* cell libraries [10]. Resource utilization and comparison to conventional FPGA implementations are presented in Table 6.1, and Openlane synthesis results is detailed in 6.2.

	[11]	[12]	[5]	[6]	Our
Resolution	640x480	320x240	1920x1080	1920×1080	320x240
Platform	Stratix II	Virtex-5	Virtex-5	Cyclone IV	Arty z7-20
LUTs	37,940	17,383	5,188	34,403	6,124
Registers	66,990	2,181	5,176	23,247	4.813
DSP blocks	120	NA	49	68	105
Memory (kBit)	NA	1,327	1,188	630	1,584
F_{max} (MHz)	127	44	270	76.2	67.5
Frame rate (fps)	30	62	64	30	878
HOG method	Window-based		Cell-based	Cell-based	Cell-based
SVM method	Window-based		Cell-based	Cell-based	Cell-based

Table 6.1: Comparison of the resources for different FPGA implementations

	Openlane synthesis result
Resolution	320x240
Logic cells	2,830
Area (nm^2)	30,424.1792
F_{max} (MHz)	200
Frame rate (fps)	2,604

Table 6.2: Openlane synthesis result

6.1.2 GDSII outcomes

After synthesizing the image processor in the previous part, the Openlane will run the hdl-to-gds flow, and finally to layout this intelligent property (IP) automatically, run several physical design rule checkers. The list of checkers is in table 6.3

Finally, this tool will generate *.gds (a database file format), *.lef (a specification for representing the physical layout), *.sdf (representation of circuit

Name	Status
Design rule check (DRC)	Pass
Layout versus schematic (LVS)	Pass

Table 6.3: Physical design rule checkers

delays), *.spice (for simulation). The final UI that displays the GDSII image of our image processor is shown in figure 6.1. In general, The image processor has 5 layers: The clk, and rst signal is placed at the left of IP, hog interface is at the bottom and svm interface will be located at the top and right.

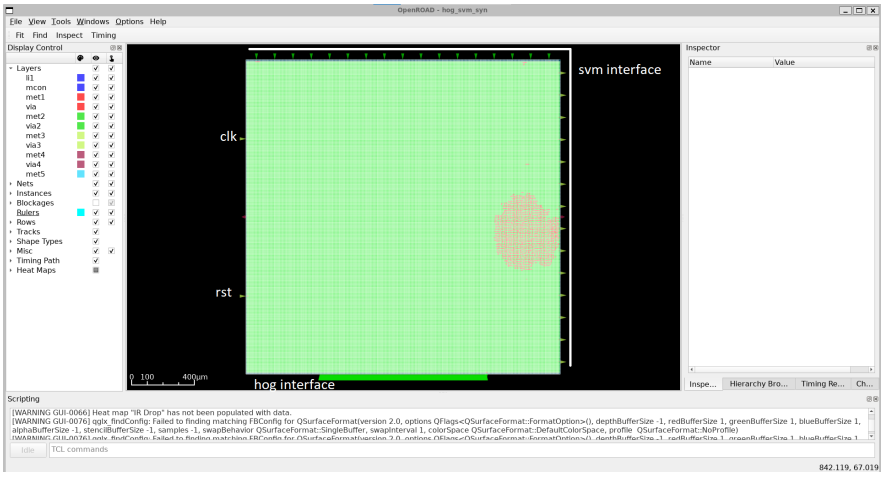


Figure 6.1: GDS UI in Openlane

6.2 FPGA prototype results

For the prototype of human-detection camera ASIC in FPGA, the code will be deployed in **DE-10 nano kit**. The camera we use is the type of **OV7670** and the monitor is **TFT 3.5 inch H896**. The detail of our prototype system diagram, images and results are shown in figure 6.2, 6.3 and table 6.4 respectively.

Prototype results	
Logic utilization	70 %
Registers	62,438
Block memory bits	28 %
DSPs	95 %
Frequency	33 Mhz
Frame rate	30 fps
Test accuracy	97.5%

Table 6.4: FPGA prototype results

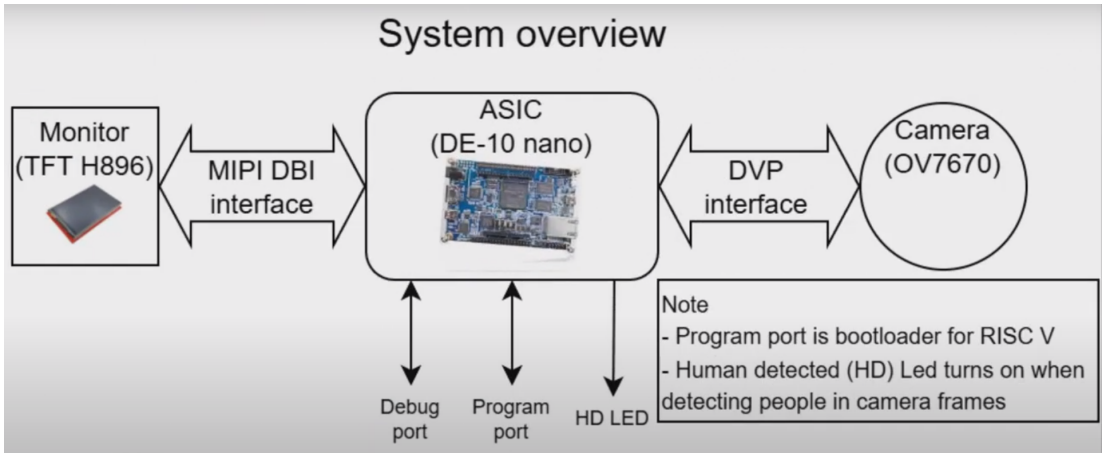


Figure 6.2: Prototype system in DE-10 nano

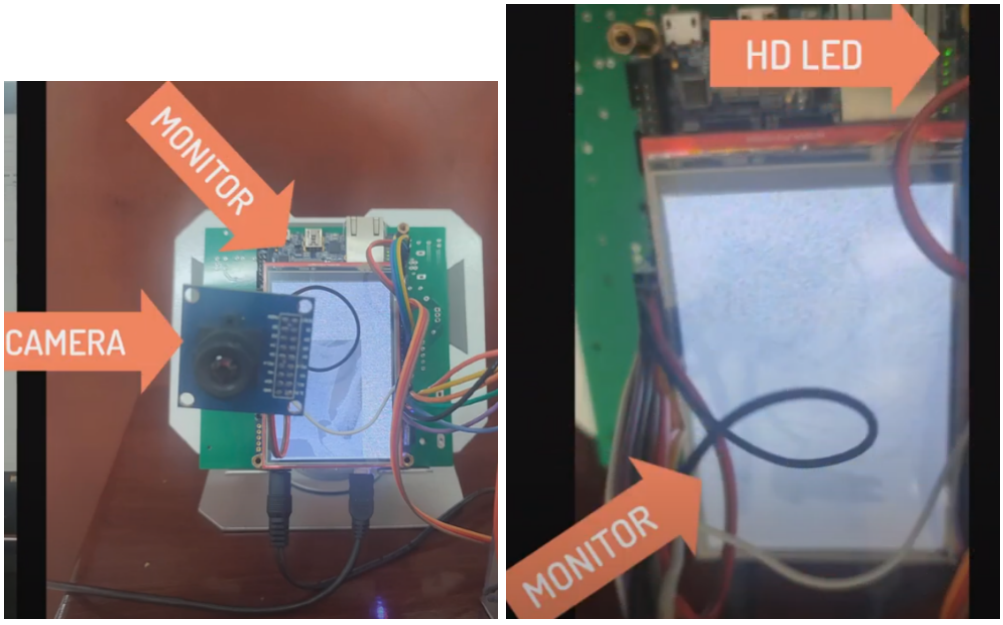


Figure 6.3: DE-10 nano implementation

The final demo results of this work are presented in figure 6.4

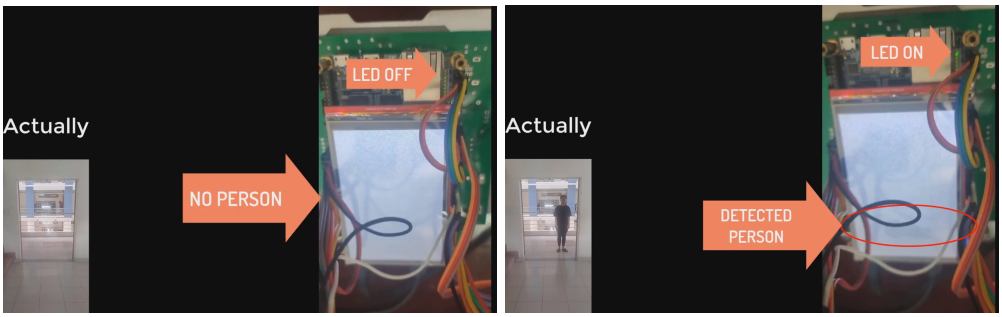


Figure 6.4: Demo results

6.3 Attached links

- Github organization: https://github.com/Digital-Camera-ASIC
- Video demo: https://youtu.be/k6s_WcHvtP4
- Openlane results and GDSII file: Google Drive link

Chapter 7

Conclusion

This work presents an architecture of real-time image processing IP and integrate it with other IPs through AXI protocol. We have a simplified HOG algorithm with cell-based scanning, simultaneous SVM calculation with changeable coefficients RAM, cell-based pipeline architecture. The resource of our work is only 6,124 for LUTs, and 4,813 for registers, while the maximum of frame rate is over 850 fps for resolution of 320 x 240. We also synthesize this IP and generate GDSII file in Openlane where the maximum frequency of this IP can reach to 200 Mhz. Moreover, with RISC-V processor, this ASIC is programmable, and other IPs, such as DMA, or image processor are configurable, so users can change SVM coefficients through bootloader if need.

Parallelized modules greatly accelerate HOG feature extraction and human detection. The proposed architecture on FPGA prototyping board shows the best performance with minimum memory usage and minimum operating frequency, compared with the performance of conventional processors.

References

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks", in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, 2015, pp. 161–170.

[2] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators", in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 2016, pp. 1–12.

[3] F. Sun, C. Wang, L. Gong, *et al.*, "A high-performance accelerator for large-scale convolutional neural networks", in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, IEEE, 2017, pp. 622–629.

[4] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient fpga implementation of yolo cnn for object detection", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[5] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll, "Fpga-based real-time pedestrian detection on high-resolution images", in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2013, pp. 629–635.

[6] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto, "Architectural study of hog feature extraction processor for real-time object detection", in *2012 IEEE Workshop on Signal Processing Systems*, IEEE, 2012, pp. 197–202.

[7] Y. Pang, Y. Yuan, X. Li, and J. Pan, "Efficient hog human detection", *Signal processing*, vol. 91, no. 4, pp. 773–781, 2011.

[8] K. Negi, K. Dohi, Y. Shibata, and K. Oguri, "Deep pipelined one-chip fpga implementation of a real-time image-based human detection algorithm", in *2011 International Conference on Field-Programmable Technology*, IEEE, 2011, pp. 1–8.

[9] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection", in *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR'05)*, Ieee, vol. 1, 2005, pp. 886–893.

[10] M. Shalan and T. Edwards, "Building openlane: A 130nm openroad-based tapeout- proven flow : Invited paper", in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–6.

[11] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura, "Hardware architecture for hog feature extraction", in *2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, IEEE, 2009, pp. 1330–1333.

[12] M. Hiromoto and R. Miyamoto, "Hardware architecture for high-accuracy real-time pedestrian detection with cohog features", in *2009 IEEE 12th International Conference on Computer Vision Workshops, ICCV Workshops*, IEEE, 2009, pp. 894–899.