Exzellenzcluster
Cognitive Interaction Technology
Kognitronik und Sensorik
Prof. Dr.-Ing. U. Rückert

Reconfigurable Vision Processing for Player Tracking in Indoor Sports

zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEUR (Dr.-Ing.)

der Technischen Fakultät
der Universität Bielefeld

genehmigte Dissertation

von

M.Sc. Omar Waleed Ibraheem

Referent:	Prof. Dr.-Ing. Ulrich Rückert
Korreferent:	Prof. Dr.-Ing. Madhura Purnaprajna

Tag der mündlichen Prüfung: 12.10.2018

Bielefeld / October 2018

Acknowledgments

I would like to express my gratitude to Prof. Dr.-Ing. Ulrich Rückert for giving me the chance to do my PhD in his research group and for all his support during my study time. Additionally, I would like to especially thank Dr.-Ing. Mario Porrmann for all his great guidance, support, motivation, and help. Furthermore, a special thank to Jens Hagemeyer for all the unlimited help, support, and time that he generously gave. I also would like to especially thank my dear colleague Arif Irwansyah for all his help, support, and for the brotherhood. Definitely, our work together was a great benefit for me and had accelerated my PhD work. I have learned many invaluable things and gained great experiences from these people during my study at Bielefeld University.

I also would like to thank Professor Dr.-Ing. Madhura Purnaprajna for reviewing my thesis as well as for the time and effort she gave for coming to Germany, participating in my PhD defense. Additionally, a special thank to Professor Dr.-Ing. Franz Kummert for chairing the examination committee, and to Dr. Qiang Li for being an examiner in this committee.

I am also very thankful to René Zorn and Meysam Peykanu for all their great help and support as well as for their friendship and brotherhood. I extend my gratitude to Cordula Heidbrede and Daniel Wolf for their help and support. I also would like to thank all the colleagues in the Cognitronic and Sensor Systems research group at Bielefeld University for giving me the required help and support during all the years of my stay in the group.

My sincere and great gratitude to my family (Mr. Waleed, Mrs. Ghadah, Rana, and Reem) for all their kind support, help, and love throughout my life. Without them, I will not be able to reach this level. Special thanks also to all my friends and all the people who helped and supported me during my PhD study. Finally, all praise to God (Alhamdulillah) for his guidance and for giving me all these big blessings.

Omar W. Ibraheem
Bielefeld, Germany

Abstract

Over the past decade, there has been an increasing growth of using vision-based systems for tracking players in sports. The tracking results are used to evaluate and enhance the performance of the players as well as to provide detailed information (e.g., on the players and team performance) to viewers. Player tracking using vision systems is a very challenging task due to the nature of sports games, which includes severe and frequent interactions (e.g., occlusions) between the players. Additionally, these vision systems have high computational demands since they require processing of a huge amount of video data based on the utilization of multiple cameras with high resolution and high frame rate. As a result, most of the existing systems based on general-purpose computers are not able to perform online real-time player tracking, but track the players offline using pre-recorded video files, limiting, e.g., direct feedback on the player performance during the game.

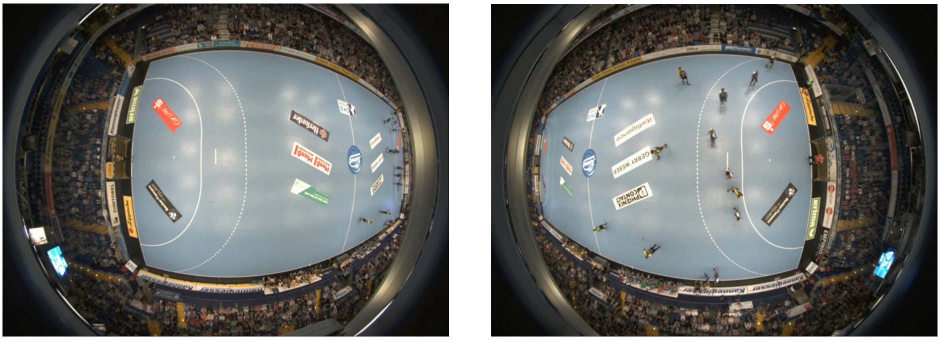
In this thesis, a reconfigurable vision-based system for automatically tracking the players in indoor sports is presented. The proposed system targets player tracking for basketball and handball games. It processes the incoming video streams from GigE Vision cameras, achieving online real-time player tracking. The teams are identified and the players are detected based on the colors of their jerseys, using background subtraction, color thresholding, and graph clustering techniques. Moreover, the tracking-by-detection approach is used to realize player tracking. FPGA technology is used to handle the compute-intensive vision processing tasks by implementing the video acquisition, video preprocessing, player segmentation, and team identification & player detection in hardware, while the less compute-intensive player tracking is performed on the CPU of a host-PC.

Player detection and tracking are evaluated using basketball and handball datasets. The results of this work show that the maximum achieved frame rate for the FPGA implementation is 96.7 fps using a Xilinx Virtex-4 FPGA and 136.4 fps using a Virtex-7 device. The player tracking requires an average processing time of 2.53 ms per frame in a host-PC equipped with a 2.93 GHz Intel i7-870 CPU. As a result, the proposed reconfigurable system supports a maximum frame rate of 77.6 fps using two GigE Vision cameras with a resolution of 1392x1040 pixels each. Using the FPGA implementation, a speedup by a factor of 15.5 is achieved compared to an OpenCV-based software implementation in a host-PC. Additionally, the results show a high accuracy for player tracking. In particular, the achieved average precision and recall for player detection are up to 84.02% and 96.6%, respectively. For player tracking, the achieved average precision and recall are up to 94.85% and 94.72%, respectively. Furthermore, the proposed reconfigurable system achieves a 2.4 times higher performance per Watt than a software-based implementation (without FPGA support) for player tracking in a host-PC.

Contents

1 Introduction

Vision-based player tracking systems are used to provide detailed information about the players' movements in sports games. This information is used to support coaches and sports scientist to evaluate and enhance the performance of the players. Additionally, the tracking results are used by TV companies to provide detailed information (e.g., on the players and team performance) to viewers. These vision systems have the advantage to be non-intrusive, i.e., they do not require extra devices for localization to be integrated in the player's outfit, which is not allowed in some sports regulations [107]. Figure 1.1 shows a handball game captured using two cameras (left and right) equipped with fish-eye lenses, covering the whole playing court.



(a) Left camera

(b) Right camera

Figure 1.1: A handball recorded game using two cameras with fisheye-lenses

The positions of the players carry significant information, which ranges from understanding the team dynamics to extracting statistics of individuals in team sports and understanding the specific pose of athletes. These data can provide valuable insights when it comes to optimizing the training and performance of individuals and teams [31]. Player tracking means finding the position of each player in a sufficient accuracy and frequency so that the path information such as distance, speed, and acceleration can be computed [66].

However, player tracking using vision systems is a very challenging task due to the nature of sports games, which includes severe and frequent interactions (e.g., occlusions) between the players. Additionally, vision-based player tracking has high computational demands since it requires processing of a huge amount of video data based on the utilization of multiple cameras with high resolution and high frame rate [107].

As a result, most of the existing systems based on general-purpose computers are not able to perform online real-time player tracking using live video streams from cameras but track the players offline using pre-recorded video files, limiting, e.g., direct feedback on the player performance during the game [107]. Therefore, hardware accelerators are required for video processing to off-load the CPU and achieve real-time systems. Several types of hardware accelerator architectures for vision processing are available, based on DSPs, GPUs, FPGAs and multi-core CPUs. Every approach has its own strong and weak points [106], and some studies have already been performed to compare their performance [30].

This thesis aims to present a reconfigurable vision system for automatic and online player tracking in indoor sports. The targeted indoor sports games are basketball and handball. Here, reconfigurability refers to the combination of a CPU and a reconfigurable device, namely an FPGA. While the CPU offers the high flexibility of software implementations, the FPGA allows for parallel, high-performance hardware implementations and is used for the various compute-intensive vision processing tasks in the target application. Automatic player tracking means there is no need to initialize the player tracker with the player position. Additionally, it includes the ability to re-track a player if the tracker is lost without re-initializing the tracker with the current position of the player and stopping the video stream. Online player tracking means the system can process and track the player in real-time on a live video stream (e.g., from multiple cameras with their maximum resolution and frame rate).

FPGA technology is used in this work as a hardware accelerator due to its various architectural benefits including: high inherent parallelism [88], flexibility, the capability of direct interfacing to cameras [106], low energy consumption, and suitability for streaming applications as well as efficiency in handling the compute-intensive operations in vision-based systems [87] [110]. In this thesis, FPGA is used to handle the compute-intensive vision processing tasks for player detection, while the less compute-intensive player tracking is performed on a CPU in a general purpose computer, achieving real-time player tracking.

1.1 Contributions

In this thesis, a reconfigurable system is presented to track the players in indoor sports automatically without user interaction. The teams are identified and the players' positions are detected based on the colors of their jerseys. Two GigE Vision cameras are used to provide a complete field of view of the playing court in the indoor sports hall, targeting player tracking for handball and basketball games. An automatic transfer of players between the two cameras is implemented, achieving player tracking for the whole court. Furthermore, FPGA technology is used to handle the compute-intensive vision processing tasks by implementing the video acquisition, video preprocessing, player segmentation, and team identification & player detection modules in hardware, realizing a real-time system. The player detection results are sent from the FPGA to the host-PC where the less compute-intensive player tracking is performed.

The main contributions of this thesis are:

- A complete reconfigurable vision processing system is proposed to detect and track the players in indoor sports automatically and without user interaction. Focusing on a resource-efficient FPGA implementation, a combination of algorithms is proposed that enables online tracking of players using live video streams acquired directly from dedicated cameras as well as process offline video data.
- An FPGA architecture is presented to accelerate the proposed system using dedicated video processing modules implemented in hardware, realizing a real-time system. These modules are: video acquistion, preprocessing, player segmentation, and team identificaion & player detection. To the best of my knowledge, this is the first work utilizing FPGAs to accelerate player tracking for handball and basketball.
- A multi-camera interface is proposed in this thesis. Different number of cameras is supported using the implemented scalable and resource-efficient multi-camera GigE Vision IP core on the FPGA. This core is capable of extracting the raw video data from multiple GigE Vision cameras in real-time with a reduced resources approach.
- A performance evaluation for player detection and tracking is presented, as well as a detailed analysis of the proposed system with respect to resource requirements, maximum achieved frame rates and throughputs, overall latency, power consumption and speedup of the FPGA-based hardware implementation.

1.2 Thesis Organization

Chapter 2 introduces player tracking systems in general, focusing on vision-based player tracking systems. The challenges of these systems are presented. The different existing architectures for vision processing are briefly shown with more focus on FPGA technology and its utilization in vision processing. Additionally, the state of the art of high-speed camera interfaces is depicted, focusing on the GigE Vision camera interface that is used in this work. Finally, the related work for vision-based player tracking systems from both academia and industry is depicted.

In chapter 3, the methodologies and fundamentals that are required to realize the proposed vision-based player tracking system are presented, including video preprocessing algorithms, object segmentation using background subtraction, and graph clustering. Furthermore, the concept of multiple object tracking is presented, focusing on the tracking-by-detection approach. Finally, the hardware platform and the design flow for the realization of the proposed system are presented.

The proposed reconfigurable system is shown in chapter 4. It includes the FPGA architecture and the processing system in the host-PC. The design and the implementation of the IP cores and modules on the FPGA are presented. These modules performs the compute-intensive vision processing tasks. Furthermore, the less compute-intensive player tracking processing in the host-PC is explained in this chapter.

Chapter 5 shows the evaluations of the proposed system. Player detection and tracking are evaluated based on standard metrics. Additionally, the performance of the implemented modules on the FPGA is reported. Furthermore, the acceleration factor using the FPGA technology, the overall system performance, and power consumption analysis are presented.

Finally, chapter 6 concludes the work presented in this thesis. Additionally, future research directions are proposed in this chapter.

2 Vision-based Player Tracking in Indoor Sports

This chapter presents a general overview of player tracking systems, focusing on vision-based player tracking systems in indoor sports. The different challenges involved in these systems are shown. Additionally, the various architectures for vision processing are briefly depicted with focus on FPGAs and their utilization in vision processing. Furthermore, the state of the art camera interfaces are presented, including the GigE Vision camera interface and its features. Finally, the related work for vision-based player tracking systems in indoor sports from both the academia and industry is depicted.

2.1 Introduction

In the last decade, there has been an increasing growth of using player tracking systems in team sports to evaluate and enhance the players' performance [106]. These player tracking systems for indoor and outdoor sports can be divided into two main categories [77]: intrusive systems and vision-based non-intrusive systems, as shown in Figure 2.1. In intrusive systems, extra devices are required for localization (e.g., wireless devices based on RFID, GPS, UWB, etc.) to be integrated in the player's outfit, which is not allowed in some sports regulations [31][77].

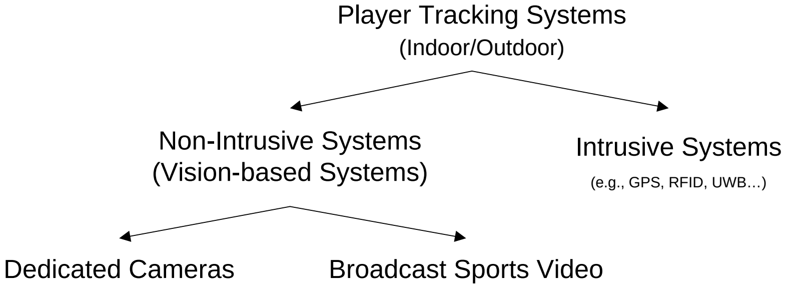


Figure 2.1: Types of player tracking systems

On the other hand, vision-based systems have the advantage to be non-intrusive, i.e., they do not require additional devices to be installed in the player's outfit. These systems usually localize the players using vision processing on video streams from one or multiple cameras. Some of these existing vision systems use broadcast sports video as the input source. This video source is usually acquired from one camera (e.g., pan-tilt-zoom camera [58]). However, using a single broadcast camera does not provide an entire view of the playing area [78]. Other systems are using dedicated cameras installed in different fixed positions in the sports hall, covering the whole sports court.

In this work, the targeted sports are handball and basketball games in indoor sports halls, where the players are tracked using vision-based systems with a dedicated camera setup. In Figure 2.2, the main characteristics of a vision-based player tracking system is shown. Different number of cameras can be used in player tracking systems. Reducing the number of cameras means there is less information that can be used to track the players. Whereas, increasing the number of cameras can be beneficial to increase the player tracking accuracy as shown in [4] as well as to support additional features (e.g., identifying the player from their jerseys' number). However, more cameras result in a higher computational cost in order to process the video data. Additionally, the overall system cost is increased. Besides selecting the required number of cameras, the camera interface should be defined. There are several interfaces available in the market, and every camera interface has its advantages and limitations. The appropriate camera interface must be selected during the system design setup according to the required specifications.

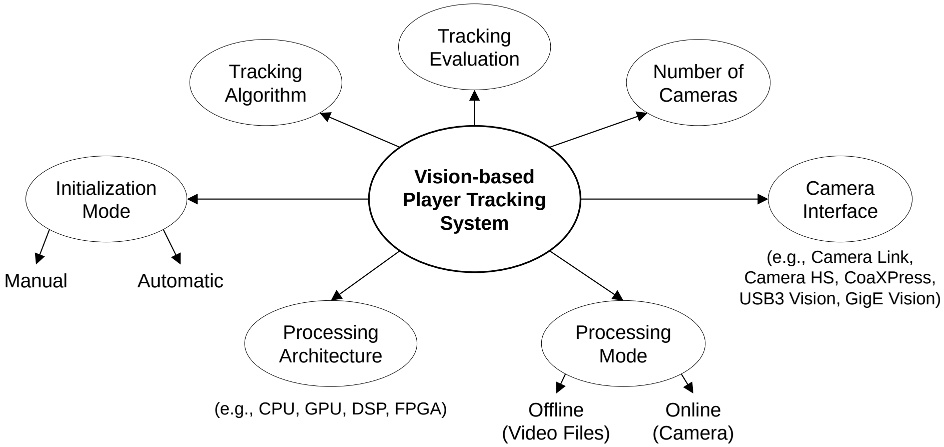


Figure 2.2: Main characteristics of vision-based player tracking systems using dedicated cameras

Processing mode includes offline and online processing. In offline processing, players are tracked using pre-recorded video files, where the video streams from these files can be paused at any time during tracking. Additionally, real-time video processing is not required. On the other hand, online processing mode requires real-time processing on the video streams that are captured directly from the cameras. Several types of processing architectures and hardware accelerators for vision processing are available, based on Digital Signal Processors (DSPs), Central Processing Units (CPUs), Field Programmable Gate Arrays (FPGAs) and multi-core CPUs. More information about these architectures are presented in Section 2.3.

The initialization of the player tracker can be achieved manually or automatically. In the former, the user needs to initialize the tracker manually, e.g., with a mouse click on each player. In some existing systems, if a player tracker loses the player during the game, the video stream must be stopped and the player tracker is reinitialized and corrected using the current position of the player. Therefore, manual tracking cannot be used with online video streams from the cameras, whereas it can be used with offline data from recorded video files where pausing the video stream is possible. In player tracking with automatic initialization, the initial positions of the players are not required for the initialization of the tracker, given some features like the colors of players' jerseys. Additionally, the players are tracked without user intervention and correction. If the tracker loses a player during tracking, this player should be tracked again after some frames.

Tracking algorithm involve the required vision processing operations to realize the player tracking system. For the evaluation of player tracking algorithm, different metrics are proposed in the literature. In this work, precision and recall are used as standard metrics for the evaluation as described in [37]. Precision is the ratio between the number of correctly detected players (True Positives (TPs)) and all the detections (TPs and False Positives (FPs)) as shown in Equation 2.1. Recall (also called detection rate) is the number of the players that are correctly detected (TPs) among the total number of players that should have been detected (TPs and False Negatives (FNs), i.e., the ground truth which represent the total number of players in a team).

$$Precision = \frac{TP}{TP+FP}, Recall = \frac{TP}{TP+FN} \tag{2.1}$$

In the next section, the different challenges in the vision-based player tracking are presented.

2.2 Challenges in Vision-based Player Tracking

Player tracking using vision systems is a very challenging task, especially due to the complicated motion patterns of the players [58]. Another challenge is the nature of sports games, which includes severe and frequent interactions (e.g., occlusions) between the players (as shown in Figure 2.3) as well as the frequently changing speed and direction of the players. Additionally, the sports hall adds further challenges to the player tracking. Typically, it contains spectators, benches for the substitute players of both teams, advertisement/sponsor panels (both digital and fixed) as shown in Figure 2.4. In general, player tracking systems need to be robust against false positives (e.g., from the spectators and substitute players). Furthermore, the system should handle any exchange between a player and a substitute player at any time. In handball as an example, the number of the allowed player substitutions is infinite, and a player exchange can happen at any time while the game is running, requiring the new player to be included in the tracking. Additionally, these vision systems have high computational demands since they require processing of a huge amount of video data based on the utilization of multiple cameras with high resolution and high frame rate.

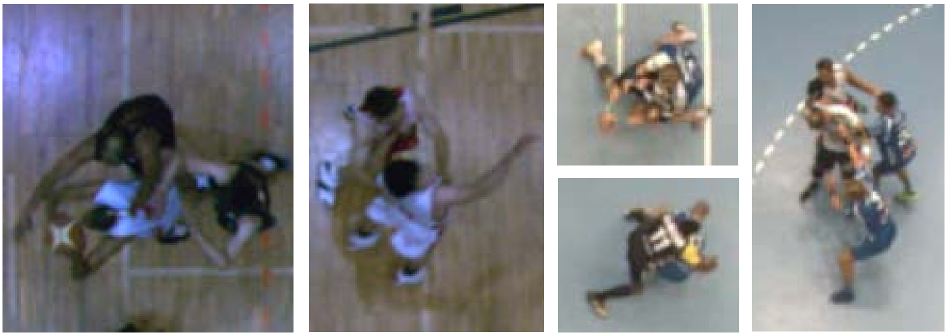


Figure 2.3: Examples of occlusion scenarios between players in basketball and handball

Therefore, the challenges involved in the vision-based player tracking can be classified into two categories: the accuracy of the tracking and the processing speed. The accuracy of tracking means how good and robust the system performs in detecting and tracking the players during the whole game, whereas the processing speed refers to the frame rate (frame per second (fps)), the system can process the incoming video streams. Online player tracking requires real-time processing of the live video streams from one or multiple cameras. In general, increasing the accuracy of tracking usually requires more sophisticated video processing algorithms as well as higher resolutions and frame rates from multiple cameras, significantly slowing down the overall system. As a conclusion, the overall system processing speed is influenced by many factors, including:

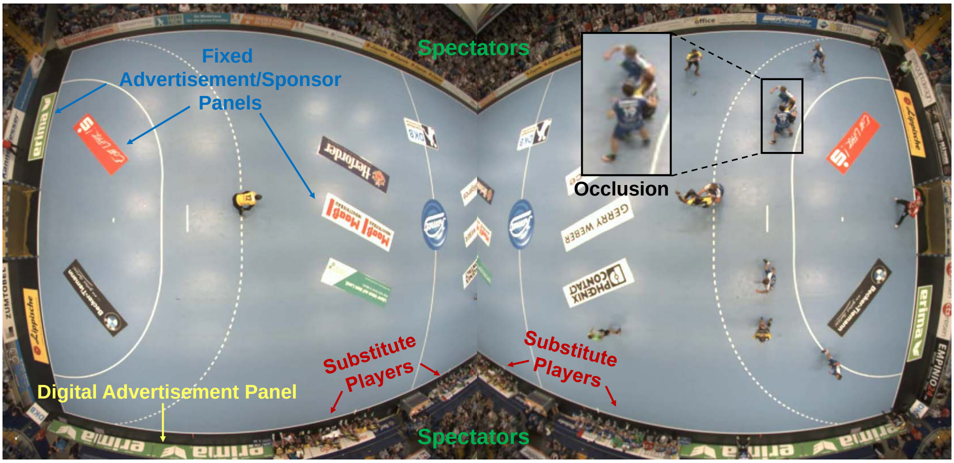


Figure 2.4: A top view of a handball sports hall captured using two cameras demonstrating the challenges in player tracking [107]

- The used algorithms in the video processing chain. More complex video processing algorithms are used to improve the accuracy and decrease the error rate of the tracking. As a result, these algorithms usually require more computational power, slowing down the overall system.

- Camera resolution and frame rate. Player tracking systems using dedicated cameras usually use cameras with high resolutions and frame rates to acquire more video data and improve the accuracy of tracking. As a result, the overall data is increased, and the speed of processing is decreased.

- The number of cameras used in the system. The player tracking systems usually use multiple cameras to track the players, covering the whole sports hall. The higher the number of cameras is, the more data needed to be processed by the computing system, resulting in an increase of the overall processing time.

- The utilized processing architectures. The overall computational speed and frame rate can be increased by using additional processing architecture as a hardware accelerator to off-load the CPU in a host-PC from the compute-intensive vision operations.

In the next section, a general overview on the available vision processing architectures is depicted.

2.3 Architectures for Vision Processing

Different types of processor architectures are available for vision processing. The most widely used are general purpose CPU, Graphics Processing Unit (GPU), and FPGA. Each architecture has its advantages and limitations. Based on the type of the used vision processing algorithms and the system requirements, the right architecture can be selected. If required, multiple processor architectures can be combined into a heterogeneous computing system [33]. In this case, one of the processors can be used as a hardware accelerator to off-load the other processor by performing the compute-intensive tasks of vision processing, while the other processor can be used for different operations (e.g., control tasks) to meet the system requirements (e.g., achieving a real-time system).

A general-purpose CPU is best suited for heuristics, complex decision-making, network access, user interface, storage management, and overall control [33]. The CPU architecture processes a given data using a software implementation sequentially. The performance of the CPU can be significantly enhanced using more CPU cores in a single chip, resulting in a multi-core CPU [45]. However, a general purpose CPU may be used with another architecture to process the compute-intensive vision processing tasks, achieving a better performance [33]. While a CPU consists of few cores optimized for sequential serial processing, a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously [70]. Additionally, GPUs support floating point operations and they are cost efficient [20]. FPGAs have a massively parallel architecture, including millions of programmable gates, hundreds of I/O pins and compute performance in trillions of multiply-accumulates per second (tera-MACs) [33]. FPGA technology has the advantage to simultaneously accelerate multiple parts of a vision processing pipeline. FPGAs have high-speed transceivers and interfaces (e.g., 10 Gigabit Ethernet MAC interface), making them suitable for direct camera interfaces.

A comparison between these architectures for vision processing applications is shown in Table 2.1. CPUs execute programs sequentially on their cores. Parallelism is achieved using multiple cores within a single CPU chip. However, this parallelism is limited by the number of cores (usually few cores are available inside one chip). On the other hand, GPUs have a significantly larger number of cores, achieving parallel processing. FPGAs have a massively parallel architecture by which programmable logic is used. CPU and GPUs are easy to program through the use of different high-level languages. Vision libraries (e.g., OpenCV) are supported on these architectures. Additionally, debugging tools are available for software debugging. FPGAs are more difficult to program. They are usually programmed using Hardware Description Language (HDL) (e.g., VHDL), and therefore good knowledge in hardware and digital system design is required. Furthermore, debugging the hardware design is not a trivial task, and sometimes it requires a significant time during the development process. FPGAs consume low power and achieve high performance per Watt as compared to CPUs and GPUs. Finally, the development time using FPGAs is longer than using CPUs or GPUs. It involves design entry (e.g., using VHDL), several iterations of simulations, debugging, and design modification, and verification until the design is fully functioning and realized in hardware (using FPGA chips).

Table 2.1: Comparison between CPUs, GPUs, and FPGAs for vision processing [86] [45]

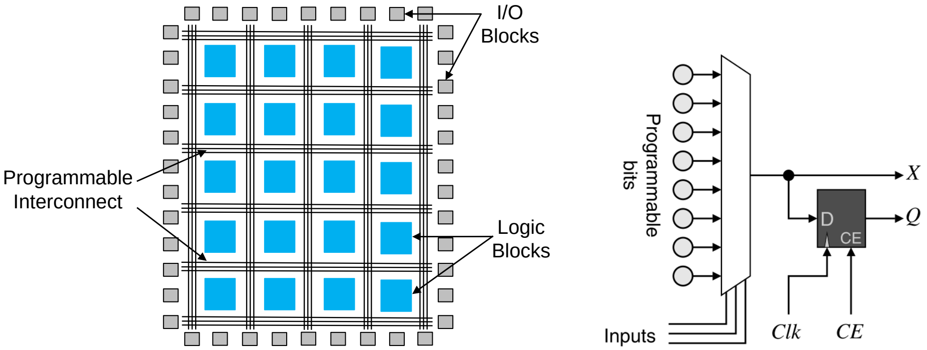
	CPU	GPU	FPGA
Processing type & parallelism	Sequential within a core (limited parallel cores)	High parallel processing (large number of cores)	Massive parallel processing architecture
Implementation type	Software development	Software development	Hardware development
Programming difficulty	Easier to program (e.g., C++, Vision Libraries)	Easier to program (e.g., CUDA, Vision Libraries)	Harder to program (e.g., VHDL)
Debugging	Less difficult	Less difficult	More difficult
Power efficiency	Low	High	Very high
Development time	Short	Medium	Long

In this thesis, a reconfigurable system is proposed, consisting of an FPGA and a general-purpose CPU in the host-PC to track the players in indoor sports. FPGA technology is used as a hardware accelerator to off-load the CPU from executing the compute-intensive vision processing tasks. Therefore, more details about FPGAs and their utilization for vision processing are presented in the next subsections.

2.3.1 Field Programmable Gate Arrays (FPGAs)

Field programmable gate arrays (FPGAs) are digital integrated circuits (ICs) that can be reprogrammed to a desired application or a certain functionality after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific and fixed design tasks [104]. FPGAs contain configurable (programmable) logic blocks along with configurable interconnects between these blocks. FPGAs can be configured to perform a variety of different tasks and custom hardware functionality [62] [69]. Although one-time programmable (OTP) FPGAs are available, most of today's FPGAs are Static Random Access Memory (SRAM) based which can be reprogrammed over and over again [104]. That's why the "field programmable" part of the FPGA's name refers to the fact that its programming takes place "in the field" (as opposed to devices whose internal functionality is hard-wired by the manufacturer) [62].

FPGAs have been introduced to the market by Xilinx in 1984. Currently, Xilinx and Altera (now part of Intel) are the two biggest FPGA vendors. The basic structure of an FPGA consists of three main elements: logic blocks, programmable interconnect, and I/O blocks as shown in Figure 2.5a. The logic blocks (also called Configurable Logic Blocks (CLBs) by Xilinx) consist of LookUp Tables (LUTs) and Flip-Flops (FFs). LUTs perform the logic operations and FFs store the results of LUTs [96]. A LUT and a flip-flop make a logic cell as shown in Figure 2.5b. This cell is the basic and the smallest unit of logic within the FPGA. Usually, multiple logic cells are combined into a logic block [15]. The programmable interconnects and wires connect the different FPGA elements to one another. Finally, the I/O blocks are used as ports to get data in and out of the FPGA [96].



(a) Basic FPGA architecture

(b) A logic cell [15]

Figure 2.5: A basic FPGA architecture, and a logic cell inside a logic block as the basic building unit of an FPGA

Contemporary FPGAs incorporate the basic components with additional computational elements and data storage blocks to increase the performance, computational density, and efficiency of the FPGA device. These additional elements are shown in Figure 2.6, and they include: embedded memories for distributed data storage, Phase-Locked Loops (PLLs) for driving the FPGA fabric at different clock rates, high-speed serial transceivers, off-chip memory controllers, and DSP (multiply-accumulate) blocks [96].

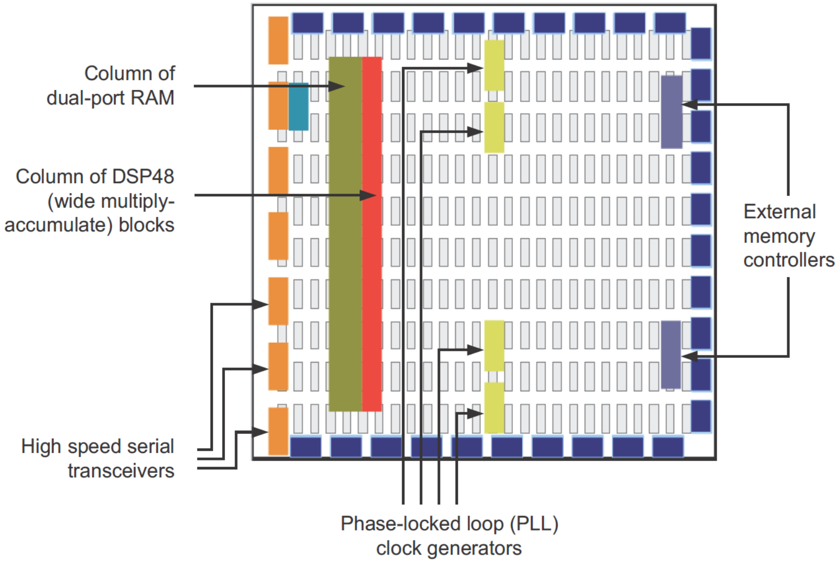

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
Phase-locked loop (PLL) clock generators
External memory controllers

Figure 2.6: A contemporary FPGA Architecture [96]

Since the work in this thesis targets the utilization of Xilinx FPGAs for the proposed player tracking system, a comparison between Xilinx FPGA families is shown in Table 2.2. The Zynq-7000 SoC devices are equipped with a single or dual-core ARM Cortex-A9 processors as a processing system (PS) with a 28 nm based programmable logic (PL) unit, improving the performance-per-watt and increasing the design flexibility [104]. Recently, Xilinx introduced the UltraScale architecture comprises high-performance FPGAs, Multiprocessor System on a Chip (MPSoC), and Radio Frequency SoC (RFSoC) families, focusing on decreasing the total power consumption [101]. A comparison between these UltraScale families is shown in Table 2.3. The Xilinx UltraSclae+ FPGAs include UltraRAM, a large memory block that enables up to 500 Mb of total on-chip storage, resulting in a 6 times increase in on-chip memory as compared with 28 nm Xilinx FPGAs [100]. Additionally, the Zynq UltraScale+ MPSoC combines the ARM v8-based Cortex-A53 processor with the ARM Cortex-R5 real-time processor and the UltraScale architecture, providing lower power consumption, heterogeneous processing, and programmable acceleration. The Zynq UltraScale+ RFSoC integrates multi-giga-sample RF data converters and soft decision forward error correction (SD-FEC) into its MPSoC architecture [104]. More information regarding these FPGA families can be found in their respective datasheets [102] [93] [103] [101]. The next subsection shows why FPGAs are suitable for vision processing.

Table 2.2: Comparison of the Xilinx FPGAs [102] [93] [103]

	Virtex FPGAs			7 Series FPGAs				SoC
	Virtex-4	Virtex-5	Virtex-6	Spartan-7	Artix-7	Kintex-7	Virtex-7	Zynq-7000
Process	90 nm	65 nm	40 nm	28 nm	28 nm	28 nm	28 nm	28 nm
LUT Size	4	6 or 5x2	6 or 5x2	6	6	6	6	6
Max. Logic Cells	200 K	415 K	474 K	102 K	215 K	478 K	1995 K	444 K
Max. Total RAM	9.7 Mb	18 Mb	37 Mb	4.2 Mb	13 Mb	34 Mb	68 Mb	26.5 Mb
Max. DSP Slices	512	1056	2016	160	740	1920	3600	2020
Max. Transceiver Block	24	24	72	-	16	32	96	16
Max. Transceiver Speed	6.5 Gb/s	6.5 Gb/s	11 Gb/s	-	6.6 Gb/s	12.5 Gb/s	28 Gb/s	12.5 Gb/s

Table 2.3: Comparison of the Xilinx Ultrascale FPGAs [101]

	Kintex UltraScale FPGA	Kintex UltraScale+ FPGA	Virtex UltraScale FPGA	Virtex UltraScale+ FPGA	Zynq UltraScale+ MPSoC	Zynq UltraScale+ RFSoC
MPSoC Proc. System	-	-	-	-	✓	✓
RF-ADC/DAC	-	-	-	-	-	✓
SD-FEC	-	-	-	-	-	✓
Process	20 nm	16 nm	20 nm	16 nm	16 nm	16 nm
Max. LUT Size	6	6	6	6	6	6
Max. Logic Cells	1451 K	1143 K	5541 K	3780 K	1143 K	930 K
Max. Block Memory	75.9 Mb	34.6 Mb	132.9 Mb	94.5 Mb	34.6 Mb	38 Mb
Max. UltraRAM	-	36 Mb	-	360 Mb	36 Mb	22.5 Mb
Max. DSP Slices	5520	3528	2880	12288	3528	4272
Max. Transceiver Block	64	76	120	128	72	16
Max. Transceiver Speed	16.3 Gb/s	32.75 Gb/s	30.5 Gb/s	32.75 Gb/s	32.75 Gb/s	32.75 Gb/s

2.3.2 FPGAs for Vision Processing

Generally, vision processing algorithms require powerful computing architecture to realize a real-time vision processing system. An example of a vision-based system implemented on a CPU in a host-PC is shown in Figure 2.7. In this example, there are five vision processing operations (Op1 to Op5), where Op1, Op2, and Op4 are pixel-based operations. Op3 is a window-based (e.g., 3x3) operation, while Op5 is a frame-based operation, requiring a complete frame to be buffered in the external memory. As shown in Figure 2.7, the video frame pixels from the camera are first stored in the external memory of the host-PC. Later, these pixels are read from the memory, and the first operation (Op1) is performed. The results of this pixel-based operation are written back to the external memory. This scenario (reading pixels from the memory, process them, and store the results in the memory) is repeated for the other operations as shown in Figure 2.7. In this case, the overall system performance depends on the number and speed of the memory read and write. Additionally, each operation requires the full frame to be stored in the external memory. Furthermore, every operation has to wait for the previous one to finish and to write its results back to the memory, increasing in the overall execution time.

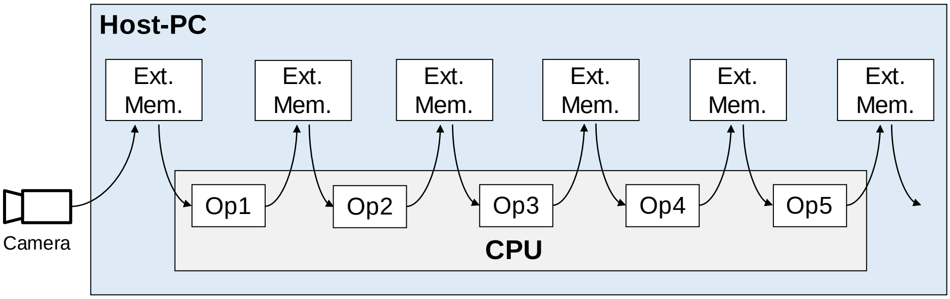


Figure 2.7: CPU-based vision processing system

An FPGA implementation of the previous example is shown in Figure 2.8. Unlike CPU, FPGA have direct connection to cameras. In this example, Op1 is performed directly on the incoming pixels from the camera, achieving stream-based pixel processing. Op2 processes the resulting pixels from Op1 on the fly, without buffering the resulting pixels in an external memory as shown in Figure 2.8. Furthermore, Op2 is started as soon as the first resulting pixel from Op1 is received, without waiting for Op1 to finish processing the whole frame. Op3 consists of a window operation (e.g., 3x3), and therefore it buffers the required data (e.g., two row-buffers) using the FPGA fast on-chip memory. Since Op5 requires the complete frame for its processing, the resulting pixels from Op4 are stored in the external memory and read back by Op5 as shown in Figure 2.8. Depending on the targeted application, the results from Op5 can be sent to a display device for visualization, used to control a device, or sent to a host-PC for post-processing. In the latter case, a CPU can be used to further process the resulting data and to store the results. Here, FPGA is used to off-load the CPU by preprocessing the video data (Op1 to Op5) as shown in Figure 2.8.

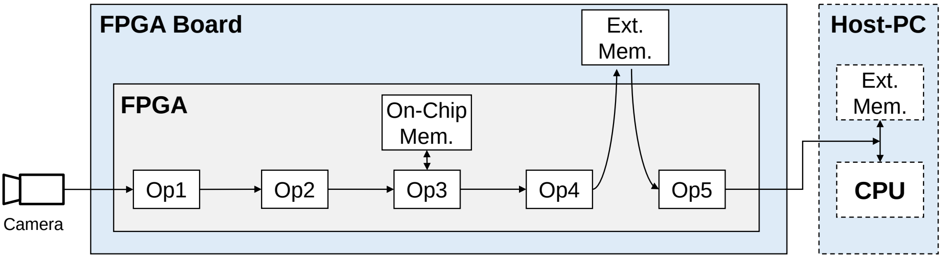


Figure 2.8: FPGA-based vision processing system

Another advantage FPGAs offer is the programmability feature which has many benefits in the vision processing applications due to the continuous evolution of new vision algorithms and standards [90]. As compared to ASICs where the designed functionality is fixed and can not be changed, FPGAs can be reprogrammed if a change in the video processing chain is needed (e.g., an additional video processing core is integrated in the FPGA architecture).

Moreover, image data can take advantage of the parallel processing capabilities offered by the FPGAs [88]. An example of how the parallel architecture of an FPGA can be exploited for vision processing is shown in Figure 2.9. Here, white balancing is applied to the incoming video streams from four cameras. Utilizing the FPGA parallelism feature, four instances of a white balance implementation (or Intellectual Property (IP) core) are used to process the video streams from the cameras in parallel as shown in Figure 2.9. Moreover, in the white balance IP core, the color components (R, G, and B) of each pixel are multiplied by their predefined gain values (G_{Red}, G_{Green}, and G_{Blue}) in parallel.

In addition to the FPGAs massively parallel architectures, FPGAs have efficient DSP resources that can be used to implement different arithmetic operations in vision algorithms. For the example shown in Figure 2.9, the three multiplications can be efficiently mapped to DSPs of an FPGA. Another advantage of FPGAs is the low energy consumption. FPGAs consume significantly lower energy than CPUs and GPUs and they achieve high performance per Watt [45]. In addition to that, FPGAs have large amounts of on-chip memory which can be used for the buffering of image pixels. Furthermore,

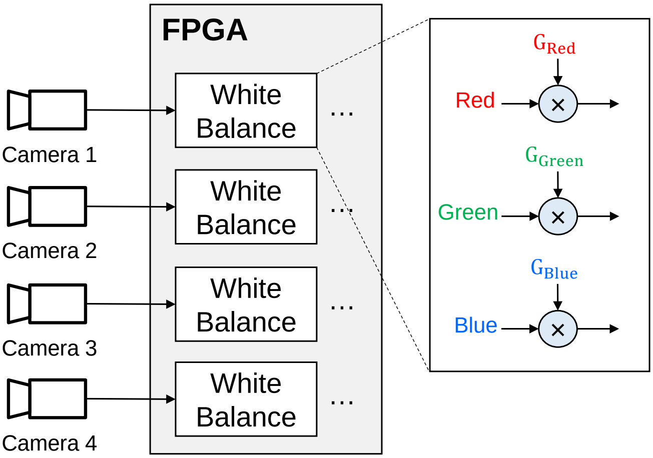


Figure 2.9: An example illustrating the utilization of an FPGA parallelism for vision processing

FPGAs are equipped with very high-speed interfaces and general purpose input/output pins, providing capabilities of direct interfacing to cameras [106] [89]. Moreover, FPGAs are suitable in smart cameras, where image acquisition from the sensor, image sampling, and application-specific preprocessing are performed before the frame data transmission to a host [90].

For embedded vision systems, traditional DSP processors or microcontrollers do not have the computational power in general to achieve real-time vision processing. One solution is to use a System on a Chip (SoC) with programmable integrated logic resources (e.g., Xilinx Zynq SoCs presented in the previous section). Using these SoCs, the system performance can be optimized by implementing the vision processing algorithms using the FPGAs fabrics and the software parts using the hardcore processor (e.g., an ARM processor) of the SoC. This feature makes such SoCs very suitable for embedded vision processing systems. However, Nvidia introduced SoCs, combining an ARM processor and a Nvidia GPU on a single chip, targeting embedded vision applications. One example is the Nvidia Jetson TX1 board, integrating a quad-core 64-bit ARM CPU and a 256-core GPU [71], making it suitable for embedded Artificial Intelligence (AI) computing.

In addition to vision processing, FPGAs are used in various applications, including communication, control, network, medical, robotics, etc. Romoth et al. [76] presented a survey of FPGA applications based on the published research work in the period from 2000 to 2015. Based on this survey, image processing is the second largest application where FPGAs are used, after their utilization in communication applications. It is stated that the main reason for implementing vision algorithms on FPGAs is the parallelism which allows real-time image processing. Additionally, the on-chip memory included in modern FPGA architectures enables the buffering of relevant image information, thus reducing the communication with external memories which can be a potential bottleneck.

The next section depicts the state of the art high-speed camera interface standards. Based on the interface specifications and the requirements of a vision-based player tracking system, the appropriate camera interface must be selected.

2.4 State of the Art High-Speed Camera Interface Standards

In general, high-speed cameras with long distance cable length are required for the player tracking systems. Different cameras with various interfaces are available. A camera interface standard is used to define the camera specifications and to provide a standard output to a processing architecture that can be used for subsequent vision processing, storage, or display. The standard specifications play an important role in the selection of the appropriate camera for a specific application. These specifications define many factors including the maximum throughput or bandwidth, maximum cable length used for the video data transfer, power requirements and the ability to deliver power over the data cable, etc.

In this section, the state of the art high-speed camera interfaces standards is presented. It includes Camera Link, Camera Link High Speed (HS), CoaXPress, USB3 Vision, and GigE Vision. Additionally, the advantages and limitations of every camera interface standard as well as a comparison between these standards are depicted. More focus is given to the GigE Vision standard, showing why it is relevant for the vision-based player tracking in indoor sports application.

Camera Link standard was initially released in 2000. It defines a complete interface which includes provisions for data transfer, camera timing, serial communications, and real-time signaling to the camera [11] [3]. Camera Link is a non packet-based protocol. It supports real-time high-speed video frame transfer (2 Gbps using one cable, and up to 6.8 Gbps using two cables), easy product interoperability, lower cable prices, single cable power (Power over Camera Link, PoCL®, allows the camera to be powered by the frame grabber through the Camera Link cable, saving space and cost), and PoCL-Lite (smaller connector supporting base configurations for low-cost solutions) [11].

However, the maximum cable length that the Camera Link interface supports is 10 meters. Additionally, a special frame grabber is required for the acquisition of video frames.

Camera Link HS standard was released in May 2012 as an improvement on Camera Link by using off-the-shelf cables to extend the reach and also offering increased bandwidth [3]. It features low latency, low jitter, and real-time data transmission. The interface takes the key strengths of Camera Link and adds new features and functions. Although Camera Link HS is a very capable approach, but it is expensive to implement and maintain when its full benefits are realized [2]. The standard provides [10] [3]: scalable bandwidths from 2.4 to 134.4 Gbps, extremely reliable data delivery, copper or fiber optic cables (up to 15 meters using copper, and up to 5000 meters using fiber cables).

CoaXPress (CXP) standard was released in December 2010, originally hosted by the Japan Industrial Imaging Association (JIIA). It uses a single coaxial cable (75 Ω cable) to transmit video data from a camera to a frame grabber at up to 6.25 Gbps; simultaneously transmit control data and triggers from the frame grabber to the camera at 20.8 Mbps. Link aggregation is used when higher speeds are needed, with more than one coaxial cable sharing the data [3]. The use of coaxial cable by CoaXPress enables automatic equalization of cable losses, allowing it to operate over greater distances. This standard also includes real-time trigger support, making it well suited for time critical applications like fast area scan and line scan [2].

USB3 Vision is a standard interface for vision applications based on the USB 3.0 technology. It allows easy interfacing between USB3 Vision transmitter devices and hosts using standard USB 3.0 hardware [14]. The USB3 Vision standard was initiated in late 2011, with version 1.0 published in January 2013. It provides an easy plug and play installation and high performance. This standard allows off-the-shelf Universal Serial Bus (USB) host hardware and nearly any operating system to take advantage of hardware Direct Memory Access (DMA) capabilities to directly transfer images from the camera into user buffers. For the receiver devices, USB interfaces are built into almost all PCs and embedded systems, i.e., no additional interface card (frame grabber) is required in many situations. USB3 Vision uses a standard passive copper cable with a maximum cable length ranging from 3 to 5 meters. This can be extended with the use of an active copper cable to around 8 meters, and with a multi-mode fiber optic cable to 100 meters [3].

GigE Vision standard is a widely adopted camera interface standard developed using the Ethernet (IEEE 802.3) communication standard. Released in May 2006, the GigE Vision standard was revised in 2010 (version 1.2) and 2011 (version 2.0). GigE Vision allows for fast image transfer (usually 1 Gbps, and up to 10 Gbps) using low-cost standard Ethernet cables over very long distances. It transfers large images quickly in real-time. With GigE Vision, hardware and software from different vendors can interoperate seamlessly over Ethernet connections at various data rates [3].

In the next subsections, the GigE Vision interface standard is described in more details. Its various features are presented. Moreover, a comparison between the state of the art camera interfaces is depicted, concluding the reasons behind the suitability of the GigE Vision interface for the player tracking system proposed in this work.

2.4.1 GigE Vision Standard

The GigE Vision camera interface supports fast image transfer using the Gigabit Ethernet communication protocol. GigE Vision offers many benefits including [12]:

- High bandwidth (1 Gbps, 2 Gbps using two cables, and 10 Gbps is supported by the standard)
- Low cost cables (e.g., CAT5e or CAT6), and standard connectors
- Data transmission up to 100 meters in length using copper cables (can be extended using switches), and 5000 meters using fiber optic
- High scalability due to the fast growth of Ethernet (e.g., GigE Vision over 10 Gigabit Ethernet)
- Network capabilities
- Power over Ethernet (PoE) support
- Standard hardware and cables allow for an easy and low cost integration

GigE Vision systems cover different network topologies. The simplest one is a point-to-point connection between a processing device (e.g., host-PC) and a GigE video streaming source (e.g., camera) using a crossover cable, or over an Ethernet network [13]. Since Gigabit Ethernet interfaces are built into almost all PCs and embedded systems, an additional interface card (frame grabber) is not necessary for many receiver devices. A GigE Vision camera is shown in Figure 2.10.

The GigE Vision standard is based on UDP/IP protocols. A GigE Vision packet is composed of Ethernet, (Internet Protocol) IP, User Datagram Protocol (UDP), GigE Vision headers, and the corresponding data payload as shown in Figure 2.11. The Medium Access Control (MAC) address of the source device (i.e., the camera) and the destination MAC address (Ethernet Physical Layer (PHY) of the receiver, e.g., FPGA) are part of the Ethernet header. The GigE Vision header is either a GigE Vision Control

Figure 2.10: A GigE Vision camera [47]

Protocol (GVCP) or GigE Vision Streaming Protocol (GVSP) header. According to the packet format value in the GVSP header, there are three main packets for the standard transmission mode [106]: data leader, data payload, and data trailer packets as shown in Figure 2.12. A data leader packet starts the transmitted data block and provides information regarding the payload type of the block. This leader packet contains additional information, e.g., the height and width of the transmitted frame. The data leader is followed by the data payload packets which contain the actual video data to be streamed. Finally, a data trailer packet indicates the end of the transmitted data block. Within the data transmission of one video frame, data payload packets must be set sequentially. The sequence of sending these packets in the standard transmission mode is shown in Figure 2.12 [13].

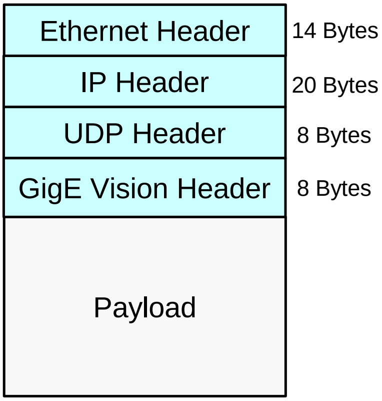


Figure 2.11: A GigE Vision packet

The maximum length of a standard GigE Vision packet is 1514 Bytes including all headers. This length can be extended by using Jumbo packets, significantly reducing

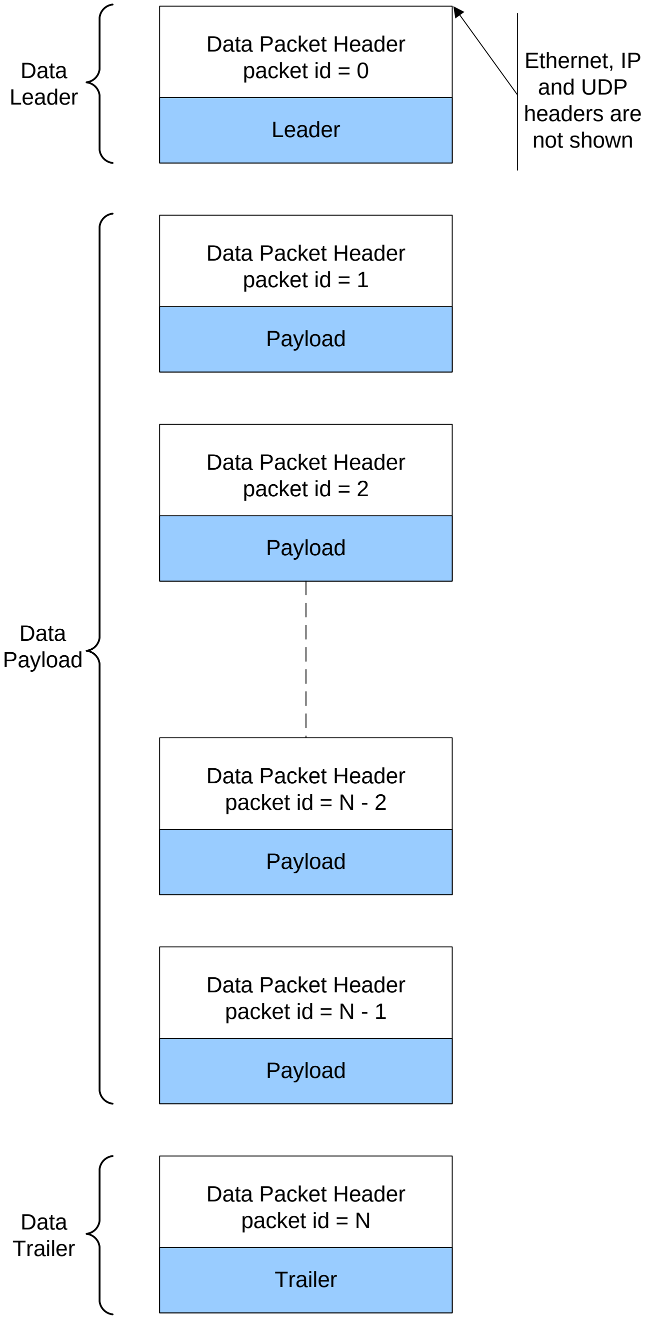

Data Leader
Data Packet Header packet id = 0
Leader
Ethernet, IP and UDP headers are not shown
Data Payload
Data Packet Header packet id = 1
Payload
Data Packet Header packet id = 2
Payload
Data Packet Header packet id = N - 2
Payload
Data Packet Header packet id = N - 1
Payload
Data Trailer
Data Packet Header packet id = N
Trailer

Figure 2.12: Standard transmission mode in GVSP protocol [13]

the transmission overhead [106]. Therefore, for the transmission of one video frame, multiple data payload packets are required as shown in Figure 2.12.

GigE Vision standard consists of four parts [13] [106]:

- Device Discovery
- GVCP
- GVSP
- Bootstrap Registers

The Device Discovery is used for assigning a valid IP address to a new GigE Vision device (e.g., a camera). GVCP allows applications to configure and control a GigE Vision device where the application can get and set various attributes such as image width, height, pixel format, frame rate, etc. Furthermore, GVCP is used to start and stop the GigE Vision device. GVSP defines how images are packetized and the mechanisms by which images can be transferred. Bootstrap registers are defined to enable the configuration of a device by storing the configuration values of the different attributes. These registers can be accessed through their unique addresses [106].

2.4.2 Comparison of the High-Speed Camera Interface Standards

A comparison between the state of art digital high-speed camera interface standards is shown in Table 2.4. As can be seen, all the standards support a high-speed video transfer. However, the allowed cable lengths significantly vary between these standards, ranging from 5 m (e.g., USB3 Vision) to 100 m (e.g., GigE Vision) (excluding the use of fiber cables). In general, the supported cable length is one of the primary factors in selecting the camera interface for vision-based player tracking systems in indoor sports, since long cables offer the flexibility of installing cameras in different positions inside the sports hall. Therefore, the GigE Vision camera interface is chosen for the player tracking system in this work. In addition to the video transmission over long-distance cables, the GigE Vision interface uses the standard Ethernet cables, offering easy interoperability with other devices (e.g., a host-PC).

Table 2.4: State-of-the-art digital camera interface standards comparison [3]

	Camera Link	Camera Link HS	CoaXPress	USB3 Vision	GigE Vision
Initial release	October 2000	May 2012	December 2010	January 2013	May 2006
Current version	2.0	1.0	1.1	1.0	2.0
Latest release	February 2012	May 2012	February 2013	January 2013	November 2011
Topology	Point-to-point	Point-to-point, data-splitting	Point-to-point	Point-to-point, tiered-star	Point-to-point, Network
Bandwidth (c=cable)	2 Gbps, 6.8 Gbps (2xc)	2.4 Gbps, 134.4 Gbps (8xc)	1.25 Gbps, 28.8 Gbps (6 coax in 1xc)	3.2 Gbps	1 Gbps, 2 Gbps (2xc), 10 Gbps
Cable types	Camera Link	CX4, Fiber	Coaxial	USB	CAT-5e, CAT-6a, CAT-7, Fiber
Cable Length	7-15m	10-15m 5000m (fiber)	35-100m	3-5m 100m (fiber)	100m 5000m (fiber)
Power over cable	Optional	No	Mandatory	Mandatory	Optional

2.5 Related Work of Vision-based Player Tracking Systems

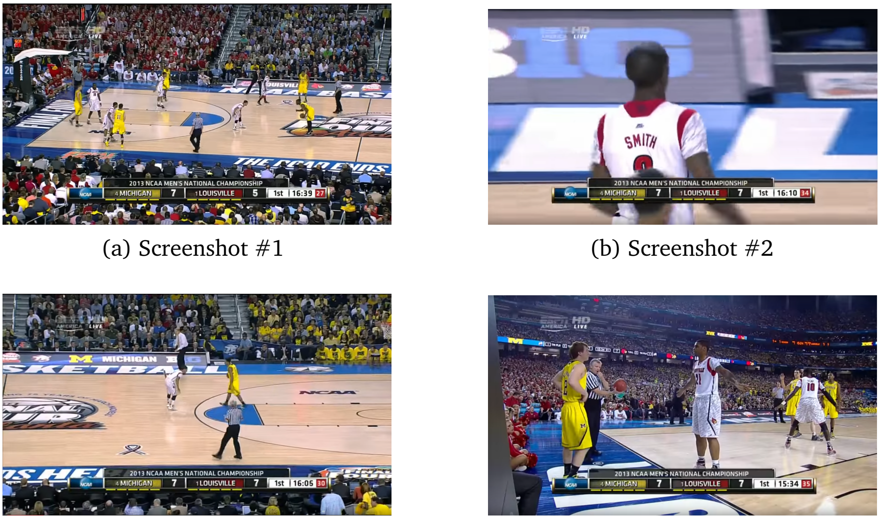
Various vision-based player tracking systems have been proposed in the literature, offering different methods to track the players and supporting different video frame resolutions, and video input sources. Some of these existing systems use broadcast sports video. Other systems utilize dedicated cameras installed at different fixed positions in the sports hall. In this section, the related work for vision-based player tracking systems for indoor sports is presented, including: broadcast video systems and dedicated camera systems from academia as well as commercial products. Finally, related work regarding the utilization of FPGAs for object tracking is depicted.

2.5.1 Broadcast Video Systems

Player tracking can be achieved using a broadcast sports video as the input source. This video source is usually acquired from one camera (e.g., pan-tilt-zoom camera [58]). However, using a single broadcast camera does not provide an entire view of the playing area [78]. Additionally, this broadcast video stream usually focuses (e.g., zoomed-in) on a certain area of the sports hall (e.g., where the ball and the nearby players are located). The drawback of this approach is that the whole playing court is not covered in every frame and not all players can be tracked. Figure 2.13 shows four screenshots captured from a broadcast video of a basketball match using multiple cameras. In these screenshots, many players are not visible, limiting complete player tracking during the whole game.

Hu et al. [41] presented a player tracking system for broadcast basketball videos. A CamShift based tracking method is used to extract the player trajectories from the video. A total of 11705 frames of basketball matches were used for the evaluation. Each broadcast video has a resolution of 720x480 pixels at 29.97 fps. The achieved average precision and recall values are 91.38% and 91.34%, respectively. However, no details on the implementation platform are provided.

Chen et al. [27] proposed a system to detect and classify screen patterns in broadcast basketball video. This system automatically detects the court lines for camera calibration, determines the court region and extracts the players using color information. Furthermore, the extracted players are classified and discriminated into the offensive and defensive teams. Players are distinguished from foreground objects using k-means clustering, while player tracking is achieved using Kalman filter. The used videos were recorded from live broadcast television programs with a resolution of 640x352 pixels and a frame rate of 29.97 fps. The achieved average precision and recall rates



(a) Screenshot #1

(b) Screenshot #2

(c) Screenshot #3

(d) Screenshot #4

Figure 2.13: Screenshots from Louisville vs Michigan 2013 NCAA basketball championship game using broadcast video (Source: YouTube)

are 89.71% and 89.20%, respectively.

Lu et al. [58] proposed a system for learning to track and identify players from broadcast sports videos for basketball using a single pan-tilt-zoom camera. Player detection is achieved using Deformable Part Model (DPM). The detection results are improved by training a logistic regression classifier to perform team classification. Finally, player tracking is performed by associating detections to tracks using bi-partite matching. The matching cost is the Euclidean distance between the detections and the prediction of the tracks' positions using a Kalman filter. The DPM detector has a precision of 73% and a recall of 78%. After team classification, the achieved precision is increased to 97% while the recall retains at 74%. After tracking, the achieved precision is 98%, and recall is 82% for the used basketball broadcast video dataset. The DPM detector used in this work requires 3 to 5 seconds to process a single image with a resolution of 1280×720 pixels. For a cascade structure and a GPU implementation, they estimate a performance of 1 frame per second. The data association and multi-target tracking run in a speed of 5-10 fps [59].

Acuna [1] presents a system for real-time multi-player detection and tracking in broadcast basketball videos using deep neural networks. Their framework is based on

YOLOv2 (a real-time object detection system), and SORT (an object tracking framework based on the Hungarian algorithm for data association and a Kalman filter for state estimation). They used the NCAA broadcast basketball dataset for training and testing. Starting with the detector, they split the dataset into 8787 annotated frames for training, and 1000 annotated frames for testing. They reported an Average Precision (AP) between 63% and 89%.

2.5.2 Dedicated Cameras Systems

In addition to broadcast video systems, player tracking can be achieved using video streams from dedicated cameras. These cameras are installed in different fixed positions in the sports hall, covering the whole sports court. Various systems are proposed using different number of cameras, resolutions, and frame rates as shown in the following.

Monier et al. [66] present a video tracking system to track the players in indoor sports using template matching technique with Closed World Assumptions (CWA). In their approach, the raw video streams are first recorded from two GigE vision cameras (1392x1040 pixels at 30 fps) equipped with fish-eye lenses. Then, their system is used to track the players based on a template (20x20 pixels as shown in Figure 2.14) selected by a human operator, who also corrects tracking errors when needed. The player template is updated to cope with the highly dynamic and changing nature of the players' movements. They reported an average correction rate between 1.9 and 6.7 corrections per player for every 1000 frames, where every frame is processed in about 100 ms (10 fps) using a software implementation. Using CWA and a multicore implementation, the achieved frame rate is 19.6 fps. In [65], Monier presents a single player particle filter-based tracking algorithm. For each tracker (player), 30 particles were used. An error rate of 5.08 errors/minute for a one-quarter basketball game is reported. With particle filter, one frame requires 347 ms to be processed (including pre-processing), which can be reduced to 103 ms (9.7 fps) using CWA and multicore implementation. The block diagram of the proposed system (based on template matching and particle filter) is shown in Figure 2.15. The used computer system is equipped with an Intel Core i7-950 series, a NVIDIA GeForce 480 GTX graphics card, and an 8 GB of DDR3-RAM operating with Windows 7 Enterprise 64-bit.

The system presented by Santiago et al. in [78] uses two overhead cameras with a resolution of 1024x768 pixels at 30 fps with wide-angle lenses to track the players. Players are detected by vest colors, and Fuzzy Logic is used to allow for a given color to be shared by different teams. Player tracking is further enhanced using Kalman filtering. Background subtraction is used in their system, where the background is obtained by capturing an image with the empty court prior to each game. In order to reduce the processing time, the background subtraction is not performed on the entire

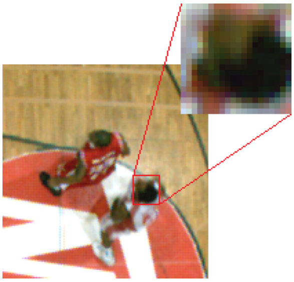


Figure 2.14: Initial template selection for tracking a player in a basketball game [66]

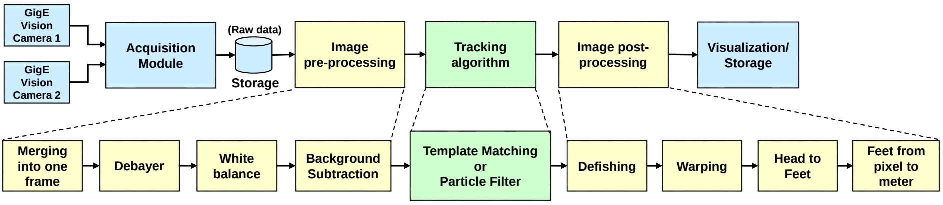


Figure 2.15: Block diagram of the work presented in [66] and [65]

image but locally on predefined regions based on the Kalman filter prediction output. The reported tracking rates range from 95.44 to 99.90%, corresponding to an average of 98.79%. For the processing time, it takes on average about 160 ms to process one frame (i.e., a performance of 6.25 fps) using an Intel i7-7630QM (2.00-2.90) GHz processor. To accelerate the overall processing, they used a GPU with OpenMP and CUDA, achieving a processing time of 62.5 ms and a performance of 16 fps for their system [79].

Both systems mentioned previously are based on the single target tracker approach and they do not provide fully automatic tracking. The user needs to initialize tracking with a mouse click on each player to initialize the tracker (initializing the template in [66] and the color calibration in [79]). Besides that, both systems use offline processing based on recorded video files.

Alahi et al. [4] presented a system to detect and track players with a mixed network of cameras. The performance of their proposed system is evaluated on a basketball game using the APIDIS dataset [7]. The authors reduced the computational requirements by downscaling all video frames to a 320x240 resolution. Their approach for player tracking is based on a sparse approximation of player location points on the ground floor. All players are detected and tracked, given a set of foreground silhouettes. Using one omnidirectional (a top view camera equipped with a fisheye lens) and one planar camera (a side view camera), a precision of 72% and a recall of 76% are achieved. The precision can be increased if additional planar cameras are used (e.g., a precision of 83% can be achieved using one omnidirectional and four planar cameras). No details about the used implementation platform are provided.

H. C. de Padua et al. [32] [72] track futsal (indoor football) players using a single stationary camera mounted in a sports hall. Their system detects the futsal players using adaptive background subtraction and blob analysis. Furthermore, they use particle filters to predict the player positions and to track them. The frame rate of the used camera is 30 fps, and the resolution is 752x480 pixels, which is cropped to 640x480 pixels in order to reduce the amount of computations. They used an official futsal match dataset, which contains 12870 frames. For player detection, they achieve a precision between 77.3% and 90.9%, and a recall between 64.9% and 76.4%. For player tracking, the achieved precision is between 70.3% and 89.3%, while the recall is between 75.9% and 80%. Additionally, 115 ID switches during player tracking is reported. Furthermore, they use a relatively small number of particles (350) to enable fast frame processing. Their proposed system requires 25 ms on average to process a frame using a computer equipped with an Intel Core i7 CPU running at 3.4 GHz, with 8 cores and 8 GB RAM [72].

Parisot et al. [73] introduced a scene specific classifier for players detection in indoor sports from a single calibrated camera. They investigated visual classifier to identify the true positives among the candidates detected by a foreground mask. Their system was validated using the APIDIS and SPIROUDOME datasets. The used videos were recorded from a one side-view still camera with a resolution of 1600x1200 pixels at 30 fps, covering the left half of the sports court for basketball games. The validations proved that their proposed combination of visual and temporal cues supports accurate and reliable player detection in team sport scenes observed from a single viewpoint. For a true positive rate (recall) of around 90%, more than 80% of the false positives from the foreground detector are rejected. In their system, the used processor is a hyper-threaded quad-core Intel i7-4790 CPU at 3.4 GHz.

Morimitsu et al. [67] proposed a novel graph-based approach for exploiting structural relations to track multiple objects with long-term occlusion and abrupt motion. Furthermore, a particle filter is used to track each object individually. The authors used their proposed method on table tennis, badminton, and volleyball games (In these sports, there are no occlusions between players of the opposing teams, while occlusion between players of the same team is very frequent). For the evaluation, Youtube videos recorded using one camera with a resolution of up to 854x480 pixels are used for the table tennis. For badminton, the ACASVA dataset with a resolution of 1280x720 pixels is utilized, while Youtube volleyball videos have a resolution of 854x480 pixels. All the these videos have a frame rate of 30 fps. Their approach outperforms other existed systems for the used datasets. For the table tennis and badminton videos, an average true positive rate (recall) of 89.3% is achieved, and an average false positive rate (the lower value, the better) of 9.6% is achieved. Furthermore, the number of ID switches (ID SW) is 85 (the lower value, the better). For the volleyball video, the achieved average recall is of 66.7%, and a false positive rate of 30.2% is reported with 624 ID switches. The system is implemented using Python on a host-PC equipped with an Intel i5 CPU. The achieved frame rate is 3 to 4 fps for the table tennis and badminton videos, and 1.5 fps for the volleyball dataset.

Furthermore, there are many works in the literature proposing generic tracking algorithms. One example is the work presented by Butt et al. [25], proposing a framework that uses higher-order constraints and Lagrangian relaxation for global multi-target tracking. The authors used two pedestrian datasets, namely the TUD-Crossing (200 frames recorded using one camera with a resolution of 640x480 pixels at 25 fps) and the ETHMS-Bahnhof (the first 350 frames, recorded using a camera with a resolution of 640x480 at 14 fps). A pre-trained pedestrian tracker is used for the detection phase. For the TUD dataset, 14 mismatches and a total number of 819 detections are reported. While for the ETHMS dataset, the number of mismatches and the total number of detections are 23 and 1514, respectively. Their algorithm is implemented in MATLAB, and it took 1.43 seconds to obtain the solution for the TUD dataset (200 frames), while 59.04 seconds are required for the ETHMS dataset (1000 frames).

2.5.3 Commercial Solutions

In this section, some examples of commercial solutions used for player tracking as well as for enhancing the viewers' experience in indoor sports games are presented. However, since these systems are commercial products, no information is available about the algorithms and the hardware used in these systems.

2.5.3.1 STATS SportVU

The STATS SportVU system [85] is used by the National Basketball Association (NBA) starting from the 2013–14 season. STATS SportVU employs a six-camera system installed in basketball arenas to track the real-time positions of players and the ball at 25 frames per second [85] as shown in Figure 2.16. Using this tracking data, STATS can create statistics based on speed, distance, player separation, and ball possession, etc. [85]. Additionally, the tracking results are used for team evaluation, broadcast enhancement, web and mobile game cast.

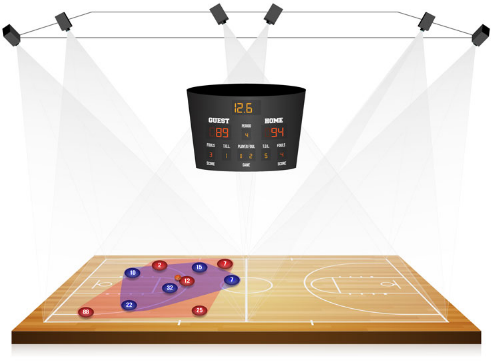


Figure 2.16: STATS SportVU system using 6 cameras at 25 fps [85]

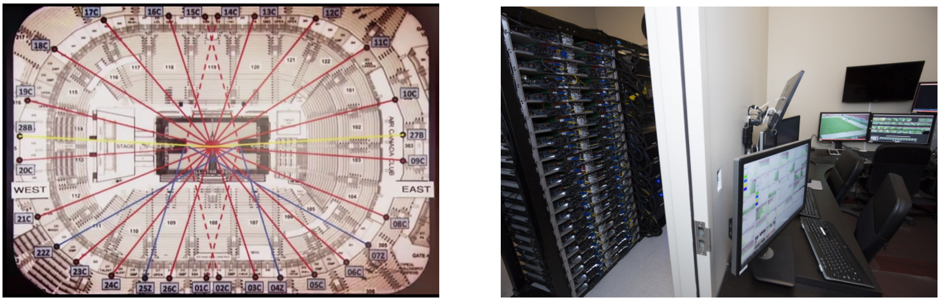
2.5.3.2 TRACAB from ChyronHego

The ChyronHego TRACAB player tracking system uses advanced image processing technology to identify the position and speed of all moving objects inside arena-based sports in real-time. In a typical deployment of TRACAB, an array of portable optical cameras installed at the pitch. This array of cameras captures live and highly accurate three-dimensional coordinates of objects, including a player, a referee, or even the ball at up to 25 times each second. To date, TRACAB has been installed in over 300 arenas and is used in more than 4500 matches per year. Some examples where TRACAB is used are the Swedish Premier Football League, English Premier League, German Bundesliga, Spanish La Liga, Japanese J.League, Danish NordicBet Ligaen, Dutch Eredivisie, and many more sports federations around the world [29].

2.5.3.3 Replay Technologies (freeD)

This product is mentioned here as an example to show how technology is used to enhance the viewers experience in sports games. Additionally, it shows how multiple cameras and a powerful processing platform for the computer-intensive operations are needed in such systems.

Replay technologies [75], founded in 2011, introduced their proprietary freeD™ technology which utilizes high-resolution cameras and compute-intensive graphics to provide a 3-D replay video for offering viewers to experience sports events from any angle. It uses 28 Ultra HD cameras positioned around the arena and connected to Intel-based servers as shown in Figure 2.17. This system allowed broadcasters to give s a 360-degree view of key plays from almost every conceivable angle [24].

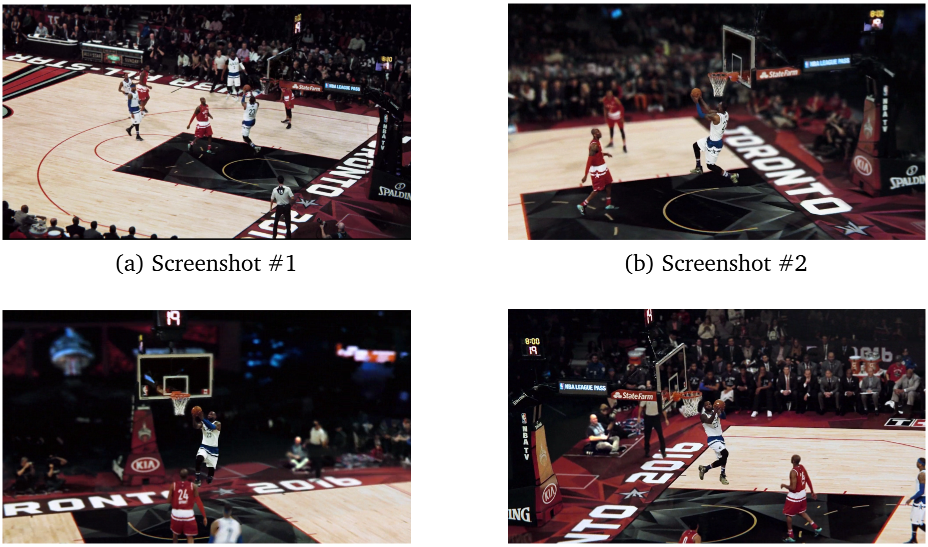


(a) The freeD 28 cameras setup in a basketball sports hall

(b) A freeD™ control room at a recent sporting event. Photo courtesy of Intel

Figure 2.17: System setup of freeD technology

Figure 2.18 shows four screenshots of a replay video taken from different angles. Intel has been collaborating with Replay since 2013 to optimize their interactive, immersive video content on Intel platforms [24]. In 2016, Intel acquired Replay technologies, offering this service to different sports leagues (including the NBA [44]).



(a) Screenshot #1

(b) Screenshot #2

(c) Screenshot #3

(d) Screenshot #4

Figure 2.18: Screenshots using freeD replay technologies

2.5.4 FPGA Accelerated Object Tracking

There is numerous research work utilizing FPGAs as hardware accelerators in object tracking applications. For example, Jacobsen et al. [46] proposed an FPGA accelerated design for an online boosting algorithm that uses multiple classifiers to track objects in real-time. Their FPGA-accelerated design performs tracking at 60 fps, achieving a 30x speedup over a CPU-based software implementation. In [5], an FPGA-based accelerator is proposed for real-time template matching. The proposed architecture achieves up to 30 fps on a 480x240 image, running ten parallel templates on a single Stratix IV FPGA. The used template size is 72x144 pixels for pedestrian tracking.

In [82], a hardware architecture for selected object tracking on embedded systems is presented. This architecture is based on the histogram of oriented gradient (HOG) and local binary pattern (LBP) algorithms. The system is used in traffic surveillance to track cars and pedestrians. It can track partially occluded cars correctly. The proposed architecture is implemented on Xilinx Virtex-4 FPGA, achieving real-time tracking on an input video with a resolution of 640x480 pixels and frame rate of 60 fps.

FPGA support is not yet available for vision-based player tracking in the sports domain, except a recent work by Li et al. [55] who introduced an FPGA-based volleyball player tracker using background subtraction and advanced template matching. Their approach identifies the positions in real-time of the six players of one volleyball team (the team close to the camera view) as shown in Figure 2.19, where the complete system setup is also presented. Here, the volleyball game is captured by a camera attached to the rightmost corner of an indoor sports hall's ceiling. The proposed system is realized using an Atlys board equipped with a Xilinx Spartan-6 FPGA (LX45 FPGA) and an Atlys VmodCAM stereo-camera board equipped with dual MT9D112 CMOS image sensors with a resolution of 800x600 pixels and a maximum frame rate of 30 fps as shown in Figure 2.19. The achieved performance is 100 fps for a resolution of 800x600 pixels using a Xilinx Spartan-6 FPGA. The authors reported a recognition accuracy of 87.1% before a match and 65.7% during a volleyball match for tracking six players of the one team. This recognition accuracy can be increased to 72.2% when they adopt template matching with a moving average filter. The used video dataset of the volleyball match consists of 948 frames (6.6 seconds). In contrast to volleyball, the work in this thesis targets games like handball and basketball, requiring to distinguish between interacting players from opposing teams.

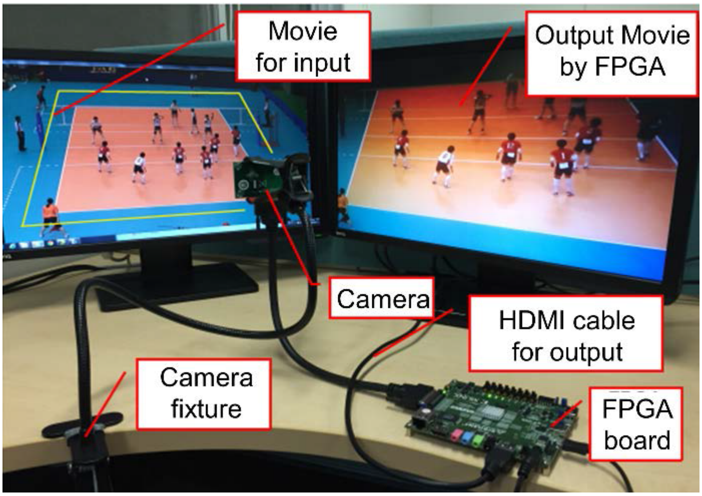


Figure 2.19: FPGA-based volleyball player tracker [55]

2.6 Summary

In this chapter, a general overview of vision-based player tracking systems is presented. Additionally, the main characteristics of player tracking systems utilizing dedicated cameras, and the challenges in these systems are depicted. These challenges concern the tracking accuracy and the processing speed of player tracking systems. The architectures for vision processing are briefly shown with more details on FPGAs and their utilization for vision processing. Moreover, the state of the art camera interfaces is presented, focusing on the GigE Vision standard and its features. GigE Vision standard supports high-speed image transfer over long distance and using low-cost cables. Furthermore, the related work on vision-based player tracking systems from academia and industry as well as FPGA-based systems for object tracking are depicted.

In the next chapter, the methodologies and the fundamentals that are required for this work are presented. This includes the video preprocessing algorithms, object segmentation using background subtraction, and graph clustering. Furthermore, the concept of multiple object tracking is depicted, focusing on the tracking-by-detection approach. Finally, the used hardware platform for realizing the proposed system is shown.

3 Methodologies and Fundamentals

This chapter presents the methodologies and fundamentals that are required to realize the proposed automatic and online vision-based player tracking system for indoor sports. It presents an overview of the different vision processing algorithms that are used in the system, including video preprocessing, background subtraction, and graph clustering. Furthermore, an overview of Multiple Object Tracking (MOT) is presented. Finally, the rapid prototyping platform that is used in this work to realize the proposed FPGA architecture is shown, in addition to the design flow for the implementation of the vision processing algorithms on the FPGA.

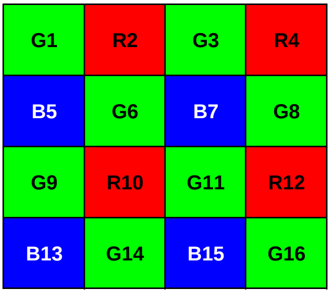
3.1 Video Preprocessing Algorithms

In this section, the vision preprocessing algorithms, which are used in this work, are presented, including Bayer pattern demosaicing, automatic white balancing, and color space conversion.

3.1.1 Bayer Pattern Demosaicing

Most modern color image sensors use a single chip with a Color Filter Array (CFA) to capture the intensity value of a single primary color for each pixel [16]. The most common filter is the Bayer pattern [17]. A Bayer pattern encoded image is shown in Figure 3.1. As can be seen, 50% of the pixels are green, 25% are red, and 25% are blue pixels. To form a full-color image out of a Bayer-encoded image, it is necessary to interpolate the missing values in each of the component images to retrieve the Red Green Blue (RGB) values for each pixel. This interpolation process is called demosaicing. A comparison between different color demosaicing methods can be found in [57]. The simplest form of filtering is the nearest neighbor interpolation. An improvement can be gained by using bilinear interpolation. The bilinear interpolation is an eight neighborhood filter. It obtains the values of the missing colors by calculating the average of the adjacent pixels. As an example, pixel number 6 has the green component (*G6*). For this pixel, the missing color components are red (*R6*) and blue (*B6*) which are calculated using Equation 3.1.

$$R_6 = \frac{(R_2 + R_{10})}{2}, \text{ and } B_6 = \frac{(G_5 + G_7)}{2} \tag{3.1}$$



G1	R2	G3	R4
B5	G6	B7	G8
G9	R10	G11	R12
B13	G14	B15	G16

Figure 3.1: Bayer pattern encoded image

For pixel number 7 where the blue component is available (*B7*), the missing red (*R7*) and green (*G7*) color components are calculated using Equation 3.2.

$$R_7 = \frac{(R_2 + R_4 + R_{10} + R_{12})}{4}, \text{ and } G_7 = \frac{(G_3 + G_6 + G_8 + G_{11})}{4} \tag{3.2}$$

For pixel number 10 which has the red component (*R10*), the missing green (*G10*) and blue (*B10*) color components are calculated using Equation 3.3.

$$G_{10} = \frac{(G_6 + G_9 + G_{11} + G_{14})}{4}, \text{ and } B_{10} = \frac{(B_5 + B_7 + B_{13} + B_{15})}{4} \tag{3.3}$$

3.1.2 Automatic White Balance

Color constancy is one of the most amazing features of the human visual system. When people look at objects under different illuminations, their colors stay relatively constant. This helps humans to identify objects conveniently [51]. In a digital camera, the sensor response at each pixel depends on the illumination when an image is captured. That is, each pixel value recorded by the sensor is related to the color temperature of the light source. For example, when a white object is illuminated with low color temperature light, it will appear reddish in the image. Similarly, this white object will appear bluish under a high color temperature. Therefore, white balance is required to process the image so that visually it looks the same way, regardless of the source of light [105], i.e., to adjust the coloration of images captured under different illuminations [51].

White balancing can be performed either manually or automatically. For manual white balancing, the user presets a certain illumination condition, and the color correction is calculated based on the preset values [52]. In Automatic White Balancing

(AWB), the necessary color correction due to the illumination is determined from the image content. Therefore, an AWB algorithm employed in a camera imaging pipeline is critical to the color appearance of digital images [51]. In an AWB algorithm, the gain values for each channel (e.g., R, G, and B channels) are calculated from the image content. These gain values are multiplied by their equivalent color components of each pixel in the image to adjust the pixel values and achieve white balancing. This process is applied to all the images in a video stream. There are various AWB algorithms proposed in the literature: Perfect Reflector Assumption (White Patch), Gray world, standard deviation-weighted gray world, etc. [105]. However, Perfect Reflector Assumption (PRA) [54] and Gray World Assumption (GWA) [43] are two common methods used to realize automatic white balance algorithms which are explained in this subsection.

In the following, let an image $I(x, y)$ have a size of $M \times N$ pixels, where x and y denote the coordinates of the pixel position. Furthermore, let $I_R(x, y)$, $I_G(x, y)$, and $I_B(x, y)$ denote the red, green, and blue channels of the image, respectively. The computed gain values to perform automatic white balancing for the R, G, and B channels are $Gain_R$, $Gain_G$, and $Gain_B$, respectively. Finally, Q_R, Q_G, and Q_B are the red, green, and blue components of a pixel at (x,y) position after white balancing.

Perfect Reflector Assumption (White Patch)

The Perfect Reflector Assumption (PRA) (also called the White Patch algorithm) is based on the Retinex theory [53] of visual color constancy, which argues that perceived white is associated with the maximum cone signals [54]. To calculate the gain values, first, the maximum values of the R, G, and B channels in an image is calculated using Equation 3.4 [105].

$$\begin{aligned} R_{max} &= max\{I_R(x, y)\} \\ G_{max} &= max\{I_G(x, y)\} \\ B_{max} &= max\{I_B(x, y)\} \end{aligned} \tag{3.4}$$

Then, the gain values for both the red and blue channels are computed using equation 3.5. The green channel is left unchanged (i.e., its gain value is 1).

$$Gain_R = \frac{G_{max}}{R_{max}}, \; Gain_B = \frac{G_{max}}{B_{max}} \tag{3.5}$$

Finally, the red and blue components of each pixel are multiplied by the respective gain values as depicted in equation 3.6. Q_G is equal to $I_G(x, y)$ since its gain value is 1.

$$Q_R = Gain_R \times I_R(x, y), \; Q_G = I_G(x, y), \; Q_B = Gain_B \times I_B(x, y) \tag{3.6}$$

Gray World Assumption (GWA)

GWA algorithm is one of the most frequently used automatic white balance algorithms [26]. The GWA algorithm argues that for a typical scene, the average intensity of the red, green, and blue channels should be equal [52]. For every frame, the average red, green, and blue components are calculated as shown in Equation 3.7.

$$\begin{aligned} R_{avg} &= \frac{1}{M \times N} \sum_{x=1}^{M} \sum_{y=1}^{N} I_R(x,y) \\ G_{avg} &= \frac{1}{M \times N} \sum_{x=1}^{M} \sum_{y=1}^{N} I_G(x,y) \\ B_{avg} &= \frac{1}{M \times N} \sum_{x=1}^{M} \sum_{y=1}^{N} I_B(x,y) \end{aligned} \tag{3.7}$$

If these three values are equal, the image already satisfies the gray world assumption, but in general, they may not be identical [52]. The gain values (which represent the color correction factors) for the red and blue channels are computed using equation 3.8. Similar to the white patch algorithm, the green channel is kept unchanged (i.e., the gain value for the green channel is equal to one). After the gain values are computed, the input image is adjusted by multiplying the computed gain values by the respective color component of each pixel as shown in equation 3.9 (Q_G is equal to the $I_G(x,y)$).

$$Gain_R = \frac{G_{avg}}{R_{avg}},\ Gain_B = \frac{G_{avg}}{B_{avg}} \tag{3.8}$$

$$Q_R = Gain_R \times I_R(x,y),\ Q_B = Gain_B \times I_B(x,y) \tag{3.9}$$

3.1.3 Color Space Conversions

A color space provides a standard method of defining and representing colors. Different color spaces are available, and a color space is selected based on the specific application [39]. In many application, conversion between color spaces is needed. In this work, the conversion from RGB to grayscale and from RGB to HSV color spaces are required, and therefore they are introduced in this section.

RGB to Grayscale Conversion

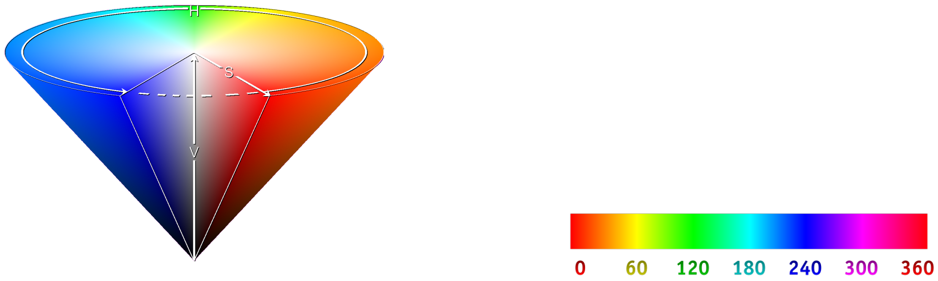
A grayscale image has one channel for intensity with the values ranging from 0 (for black) to 255 (for white). The RGB color spaces is converted to its equivalent grayscale by forming a weighted sum of the R, G, and B color components for every input pixel as depicted in Equation 3.10 [38].

$$I = 0.299 \times R + 0.587 \times G + 0.114 \times B \tag{3.10}$$

where I is the pixel intensity of the resulted gray image.

RGB to HSV Conversion

Previous work [63] proves that the Hue Saturation Value (HSV) color space is more robust than RGB color space with respect to illumination and lighting changes. The HSV color space is represented as a cone as show in FIgure 3.2a and the Hue values ranges from 0 to 360 as shown in Figure 3.2b. In this work, the RGB to HSV color space conversion is used based on the algorithm proposed by Foley et al. [36]. Let $\max(R,G,B)$ be the largest value of the R, G, and B for a pixel, and $\min(R,G,B)$ is the smallest value. The difference between these values is Δ (as shown in Equation 3.11). The Hue (H), Saturation (S), and Value (V) are calculated using Equation 3.12, Equation 3.13, and Equation 3.14, respectively.



(a) HSV cone

(b) H scale

Figure 3.2: HSV color space

$$\Delta = \max(R,G,B) - \min(R,G,B) \tag{3.11}$$

$$H = \begin{cases} 0, & \text{if } R = G = B \\ \frac{60\times(G-B)}{\Delta}, & \text{if } \max(R,G,B) = R \\ 120 + \frac{60\times(B-R)}{\Delta}, & \text{if } \max(R,G,B) = G \\ 240 + \frac{60\times(R-G)}{\Delta}, & \text{if } \max(R,G,B) = B \end{cases} \tag{3.12}$$

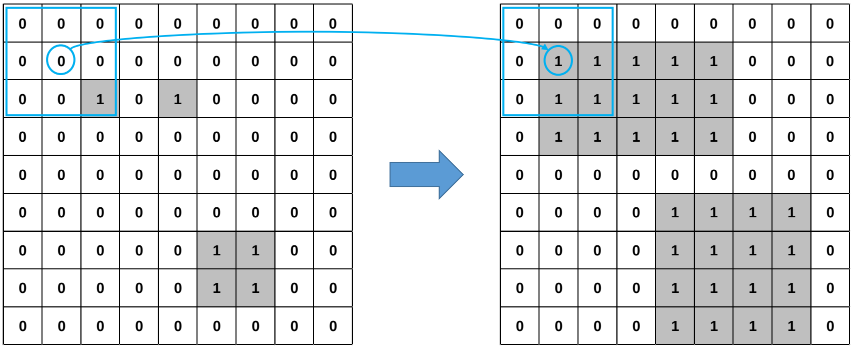
$$S = \begin{cases} \frac{\Delta}{\max(R,G,B)}, & \text{if } \max \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

$$V = \max(R,G,B) \tag{3.14}$$

If the resulting value of H is a negative number, 360 is added. As a result, H is between 0 and 360, S is between 0 and 1, and V is between 0 and 255. These values are normalized by dividing H by 2, and multiplying S by 255 while keeping the V without change.

3.2 Morphological Operations

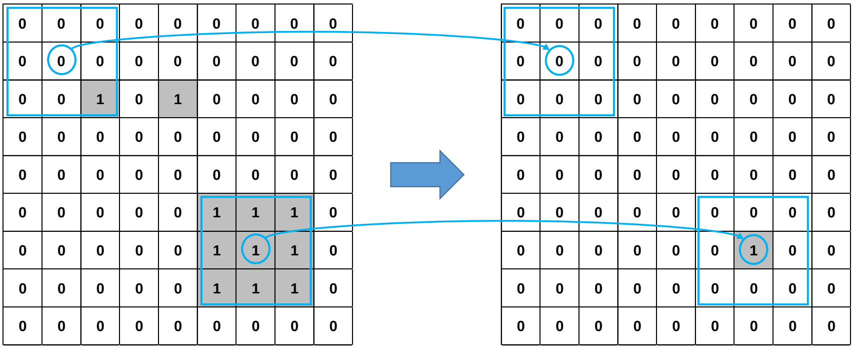
Morphological operations apply a structuring element to an input image for geometrical structure processing. Binary morphology uses a binary input image, where the background pixels are represented by a logic 0 (black) while a logic 1 (white) is used for the foreground pixels. Each input pixel is compared with its neighbors based on the selected structuring element. The two basic morphological operations are dilation and erosion. In dilation, the output pixel is set to 1 if one of the neighboring pixels in the structuring element has a logic 1. In erosion, if one of the neighboring pixels has a logic 0, the output pixel is set to 0. Erosion and dilation are shown in Figure 3.3 and Figure 3.4, respectively. In both figures, a 3x3 structuring element is used (as shown in blue). Dilation adds pixels (with logic 1) to the boundaries of the foreground objects in an image while erosion removes pixels from object boundaries by setting them to logic 0. The number of the added or removed pixels depends on the size and shape of the used structuring element.



0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0
0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0	0
0	1	1	1	1	1	0	0	0
0	1	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	0
0	0	0	0	1	1	1	1	0
0	0	0	0	1	1	1	1	0
0	0	0	0	1	1	1	1	0

Figure 3.3: Morphological dilation of a binary image (Left: input image. Right: output image)



0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	1	1	0
0	0	0	0	0	1	1	1	0
0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Figure 3.4: Morphological erosion of a binary image (Left: input image. Right: output image)

3.3 Image Thresholding

Image thresholding is a simple image segmentation technique used to partition foreground objects in an image based on the pixels values. Each pixel of an input image $(I(x,y))$ is compared to a threshold value (Thr) as shown in Figure 3.5. The output is a binary image (zero for black and one for white) based on Equation 3.15. As a result, all the objects that have pixels values greater than or equal to (Thr) are extracted from the input image and assigned a logic one as foreground objects in the output image.

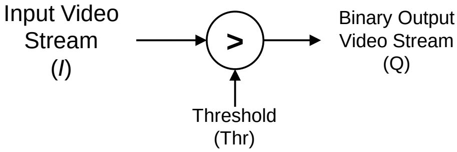


Figure 3.5: Image thresholding

$$Q(x,y) = \begin{cases} 1, & \text{if } I(x,y) \geq Thr \\ 0, & \text{otherwise} \end{cases} \tag{3.15}$$

Additionally, multiple threshold values can be used for specific ranges thresholding. Equation 3.16 shows two threshold values $(Thr_{Low}$ and $Thr_{High})$ used for thresholding an input image $(I(x,y))$.

$$Q(x,y) = \begin{cases} 1, & \text{if } Thr_{Low} \leq I(x,y) \leq Thr_{High} \\ 0, & \text{otherwise} \end{cases} \tag{3.16}$$

3.4 Object Segmentation using Background Subtraction

Background subtraction is a frequently used technique for object segmentation [35]. It is used to detect moving objects in video streams captured by fixed cameras [22]. In background subtraction, the moving foreground (FG) objects are extracted by subtracting the current frame from the background image. Background subtraction consists of two main steps: background initialization, and background update to adapt to possible changes in the scene. There are many background subtraction algorithms proposed in the literature. Some of these algorithms are used for a specific application (e.g., urban traffic [28], video surveillance [48], maritime [22], etc.). A comprehensive review of background subtraction algorithms can be found in [83].

In general, object segmentation using background subtraction consists of five steps as shown in Figure 3.6:

- Background estimation
- Background subtraction
- Morphological operations
- Masking
- Background update

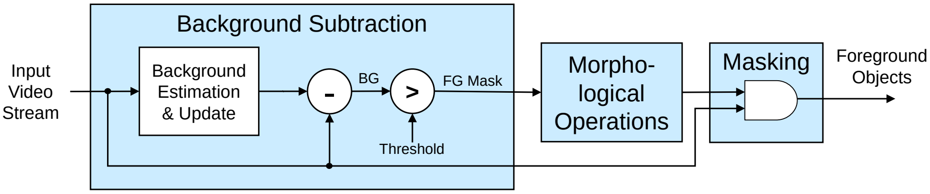


Figure 3.6: Object segmentation using background subtraction

As shown in Figure 3.6, first, the background image is generated. This estimated background is subtracted from the incoming video frames, and the results are compared with a predefined threshold value. If the resulting pixel value is greater than this threshold, the output is set to a logic 1, otherwise 0, resulting in a binary output image that contains the foreground objects. Morphological operations are applied to this binary image in order to fill the gap between pixels that belong to the foreground objects. Afterward, this binary image is masked with the current RGB video frame in the input stream to generate an image with the colored foreground (segmented) objects. Finally, the estimated background is updated to adapt to different changes (e.g., lighting, environmental conditions, etc.).

Background estimation techniques are classified into two broad categories [28]; non-recursive and recursive. A non-recursive technique stores a buffer of the previous L video frames, and estimates the background image based on the temporal variation of each pixel. The disadvantage of this approach is the storage requirement can be significant if a large buffer is needed. Recursive techniques do not maintain a buffer for background estimation. Instead, they recursively update a single background model based on each input frame. Compared with non-recursive techniques, recursive techniques require less memory storage, but any error in the background model can linger for a much longer period of time.

In this work, the players are segmented using background subtraction technique. The approximated median algorithm is selected for background estimation and update, which has been proposed by McFarlane and Schofield [64] as an efficient recursive approximation of the median filter. In median filtering, the previous N frames are buffered, and the median of these frames is used as the background. The median filter approach has been shown to be very robust and to provide a performance comparable to higher complexity methods. While storing and processing many video frames requires a large amount of memory in median filtering, the approximated median does not need to store the previous frames [18]. Furthermore, the algorithm is well suited for stream processing, making it a good choice for FPGAs. The approximated median algorithm for background estimation is shown in Equation 3.17.

$$B_t(x,y) = \begin{cases} B_t(x,y)+1 & \text{if } I_t(x,y) > B_t(x,y), \\ B_t(x,y)-1 & \text{if } I_t(x,y) < B_t(x,y), \\ B_t(x,y) & \text{if } I_t(x,y) = B_t(x,y). \end{cases} \tag{3.17}$$

where:
$I_{t(x,y)}$ is the intensity of the input pixel.
$B_{t(x,y)}$ is the intensity of the background estimation pixel at spatial location (x,y) and time t. For the initial value, zero can be used.

As shown in Equation 3.17, the running estimate of the median is incremented by one if the input pixel ($I_t(x,y)$) is larger than the estimate ($B_t(x,y)$), and decreased by one if smaller. This estimate eventually converges to the median value [28].

3.5 Graph Clustering

Clustering is the task of partitioning a set of unlabeled data into different groups in a way that the data in one group are more similar than those in the other groups. Each group is called a cluster. After applying a clustering algorithm, the data in each cluster have a higher measure of similarity than the data in the other clusters. Graphs are structures formed by a set of vertices (also called nodes) and a set of edges which are connections between pairs of vertices. Graph clustering is grouping the vertices of the graph into clusters considering the edge structure of that graph [80]. Figure 3.7 shows clustering of a data set into three clusters (shown in red, green, and light blue).

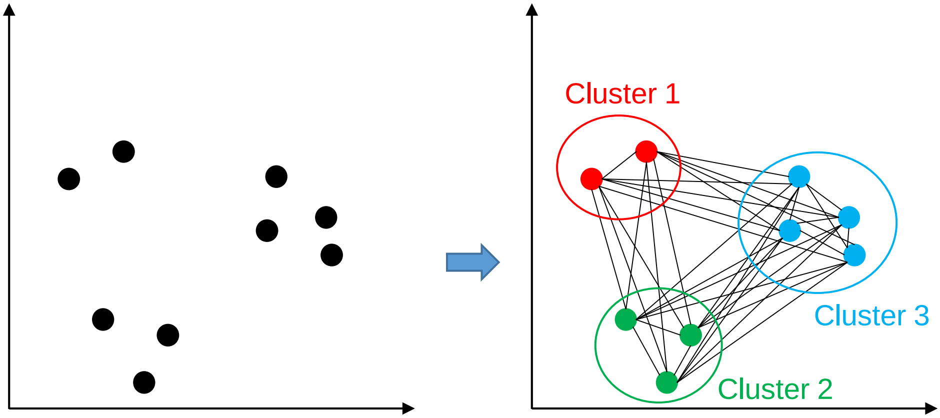


Figure 3.7: Graph Clustering of a data set. Left: before clustering. Right: after clustering

Clustering algorithms can be divided into two broad groups [61]: hard and soft clustering. In hard clustering, each data or object belongs to only one cluster. While in soft clustering, each data or object can belong to more than one cluster. Based on the possible data grouping in the application, a hard or soft clustering algorithm can be selected. There are many clustering algorithms proposed in the literature. Some of these algorithms require that the number of clusters is predefined and fixed, while other algorithms can be applied to data to obtain a variable number of clusters. Furthermore, the centroid of each cluster can be obtained by computing the mean value of the nodes within the same cluster. In this work, graph clustering is used to find the centroids of the players in each team as shown in the next chapter.

3.6 Multiple Object Tracking (MOT)

Multiple Object Tracking (MOT) (also called multiple target tracking) has an important role in computer vision. For an input video, the tasks of MOT include locating multiple objects and maintaining their identities, resulting in the individual trajectories of these objects [60]. MOT has wide applications, e.g., in surveillance, autonomous driving, sports, etc. [92]. Most of the existing MOT algorithms can be categorized into two groups: detection-free tracking and detection-based tracking (also called tracking-by-detection) as depicted in Figure 3.8 [60]. In detection-free tracking, a manual initialization of a fixed number of objects is required as shown in Figure 3.8 (bottom). Some algorithms require the re-initialization of the tracker with the current object's position during tracking, if one of the tracked objects is lost by the tracker. In detection-based tracking (tracking-by detection), the objects are first detected, and then these detections are linked into trajectories. This approach is more popular because new objects are detected and disappearing objects are terminated automatically. A comparison between these two MOT approaches is shown in Table 3.1 [60].

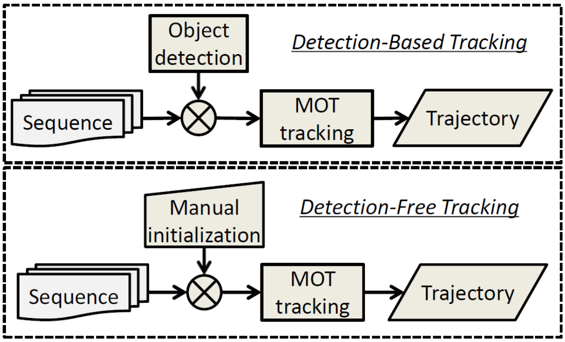


Figure 3.8: Overview of two prominent tracking approaches [60]

Table 3.1: Comparison of the MOT approaches [60]

	Detection-free tracking	Detection-based tracking (Tracking-by-detection)
Initialization	Manual, perfect	Automatic, imperfect
Nr. of objects	Fixed	Varying
Applications	Any type of objects	Specific type of objects (in most cases)
Advantages	No object detector	Ability to handle varying number of objects
Drawbacks	Manual initialization	Performance depends on object detection

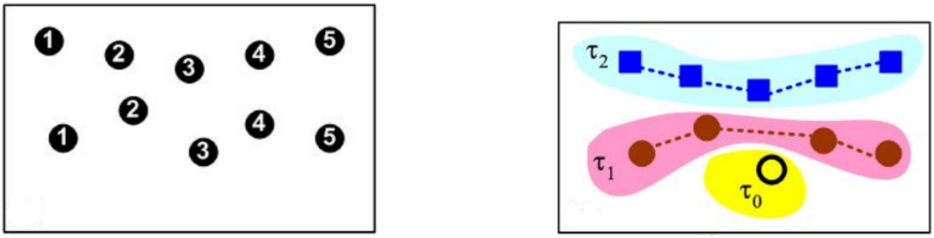
The tracking-by-detection approach is used in this work to track the players in sports. As can be seen in Table 3.1, this approach is suitable to track the players automatically and without user interaction. It tracks a different number of objects automatically.

This feature is required because of the frequent player substitutions in basketball and handball, where the trajectories of players leaving the court are terminated, and new players entering the court are tracked. Furthermore, this approach supports tracking of a specific type of objects (i.e., players in this work). Therefore, more details about the tracking-by-detection approach are presented in the next subsection.

3.6.1 Tracking-by-Detection

Most of the recent state of the art research has focused on the tracking-by-detection approach [92] [40]. This approach supports the automatic tracking of new objects enter a scene as well as the automatic termination of leaving objects [60]. Additionally, it effectively prevents the object's bounding box from being drifted away during object tracking [40]. The problem of multiple object tracking using the tracking-by-detection approach can be divided into two main parts: First, the objects of interest are detected in every single frame. Then, object tracking is achieved by associating the detections to the corresponding objects over time.

An illustration of the tracking-by-detection approach is shown in Figure 3.9 [84]. Each circle in Figure 3.9a represents a detection, and the numbers inside the circle represent the time. In Figure 3.9b, these detections are associated to different objects (t_0, t_1, and t_2) with the time. One detection is assigned only to one object, and an object has only a single detection at any time [59].



(a) Raw detections

(b) Tracking-by-detection

Figure 3.9: The tracking-by-detection approach [84]

After getting all detections in a video frame, the tracking problem then becomes a data association problem to combine detections of the same object into a corresponding trajectory [40]. Many data association logarithms are available in the literature. The munkres' version of the Hungarian algorithm [68] is one of the widely used approaches for data association [42]. It is an online algorithm that is used to find an optimal single-frame assignment [23], i.e., assigning detections to tracks in every frame in an

MOT system. Further information about these assignments is shown in the next chapter. To achieve object tracking over time, a prediction algorithm can be used to predict the object's position in the next frame. This prediction can be used as the object position if the object is not detected in a frame during tracking. One of the most used prediction methods is Kalman filter which is discussed in the next subsection.

3.6.2 Kalman Filter

Kalman filter [50] consists of mathematical equations that provide an efficient computational (recursive) means for estimating the state of a process, in a way that minimizes the mean of the squared error [91]. A broad overview of the high-level operation of a discrete Kalman filter cycle is shown in Figure 3.10. The "time update" equations projects (predicts) the current state estimate ahead in time. The "measurement update" adjusts (corrects) the projected estimate by an actual measurement at that time [91]. The state and measurement equations are shown in Equation 3.18 and Equation 3.19, respectively [91] [34].

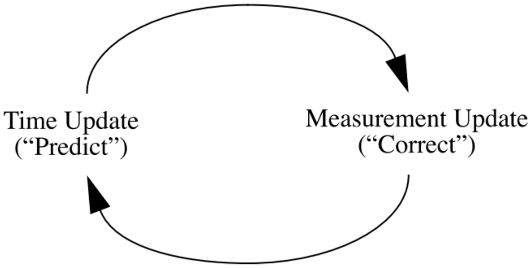


Figure 3.10: Overview of a discrete Kalman filter cycle [91]

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1} \tag{3.18}$$

Here, x_k is the state vector, containing the terms of interest for the system (e.g., position, velocity) at a time step k. A is the state transition matrix between time steps. It is applied to the previous state x_{k-1} and relates it to the state at the current step k. B is the control-input matrix which specifies the transition from control input to state. It is applied to the control vector u_k, and relates the optional control input u to the state x. u_{k-1} is the vector containing any control inputs. w_{k-1} is the process noise vector.

$$z_k = Hx_k + v_k \tag{3.19}$$

In Equation 3.19, z_k is the vector of measurements. H is the transformation matrix. It specifies the transition from state to measurement. v_k is the vector containing the measurement noise for each observation in the measurement vector.

Additionally, in order to predict the next position of an object, an object motion model is required (e.g., constant velocity, or constant acceleration). Since a selected motion model does not describe the object motion perfectly, a noise is added to the model (called a process noise). Furthermore, measurement noise represents the imperfect detections or measurements. To start the tracking process, the initial state value is required. Furthermore, the initial uncertainty is also needed which can be expressed by a Gaussian covariance matrix [49].

3.7 RAPTOR-X64 Rapid Prototyping Platform

The RAPTOR-X64 [74] system is a rapid prototyping platform, which is developed by the Cognitronics and Sensor Systems research group at Bielefeld University. The RAPTOR-X64 is designed as a modular rapid-prototyping system: the base system provides communication and management facilities, which are used by a variety of extension modules, realizing application-specific functionality [74].

The architecture of the RAPTOR-X64 prototyping system is shown in Figure 3.11. The RAPTOR-X64 platform supports up to six FPGA daughterboards (as extension modules), equipped with Xilinx FPGAs and onboard dedicated memory units. Additional extension modules are available offering different interfaces (e.g., Ethernet, display, USB, etc.). Furthermore, the RAPTOR-X64 system has a Peripheral Component Interconnect eXtended (PCI-X) interface used for communication with the host-PC as shown in Figure 3.11. This PCI-X interface is directly connected to the local bus for high-speed communication. Therefore, the RAPTOR-X64 can be used to realize a reconfigurable vision system, by which the compute-intensive vision processing tasks are performed on the FPGA. Additionally, the processed data on the FPGA can be sent through the PCI-X interface to the host-PC for further processing, storage, and display.

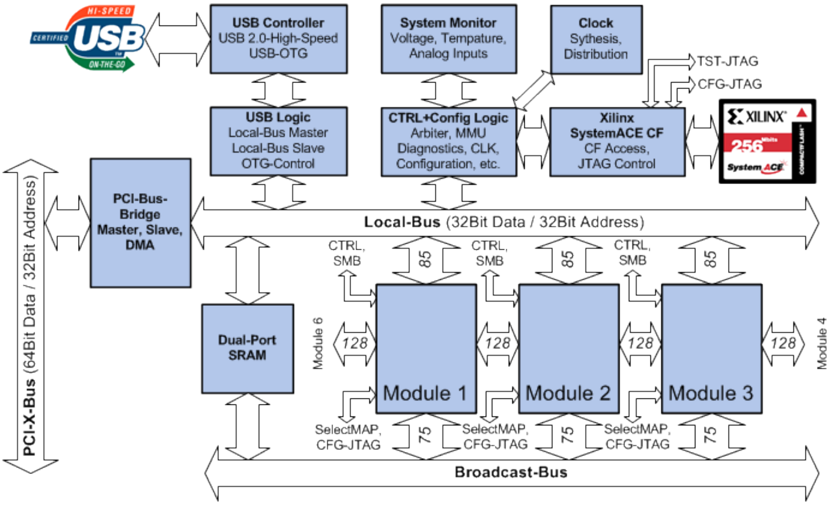


Figure 3.11: Architecture of the RAPTOR-X64 prototyping system [74]

3.8 Design Flow

In the proposed reconfigurable system, IP cores are designed and implemented on the FPGA for various vision processing tasks. Figure 3.12 shows the design flow used for the implementation of the vision processing algorithms on the FPGA. First, the problem is defined, and a vision processing algorithm is selected accordingly. Then, this algorithm is implemented using a high-level language (e.g., C/C++ with the OpenCV library, MATLAB). In this step, the result of the selected algorithm is evaluated. If the results do not fulfill the system requirements, a different algorithm is selected. Otherwise, (if the results satisfy the requirements) the algorithm is chosen for the hardware implementation. The selected algorithm is optimized for hardware implementation if applicable. This optimization involves partially modifying the algorithm for an efficient FPGA implementation. An example of such modification is reducing the number of divisions in the algorithm or avoiding the divisions by converting them to multiplications. This step is optional since it depends on the selected algorithm.

Subsequently, the algorithm is implemented in hardware by creating an IP core using the Very High Speed Integrated Circuit Hardware Description Language (VHDL). This implementation step is accompanied by several simulations using the ModelSim tool, i.e., the VHDL code is modified and the IP core is simulated repeatedly until the desired results are achieved. To verify the results of the IP core, a MATLAB code and a VHDL testbench are written. The MATLAB code is used to convert a test image (e.g., an image for a sports hall with players) into a binary image and store it in a text file. The

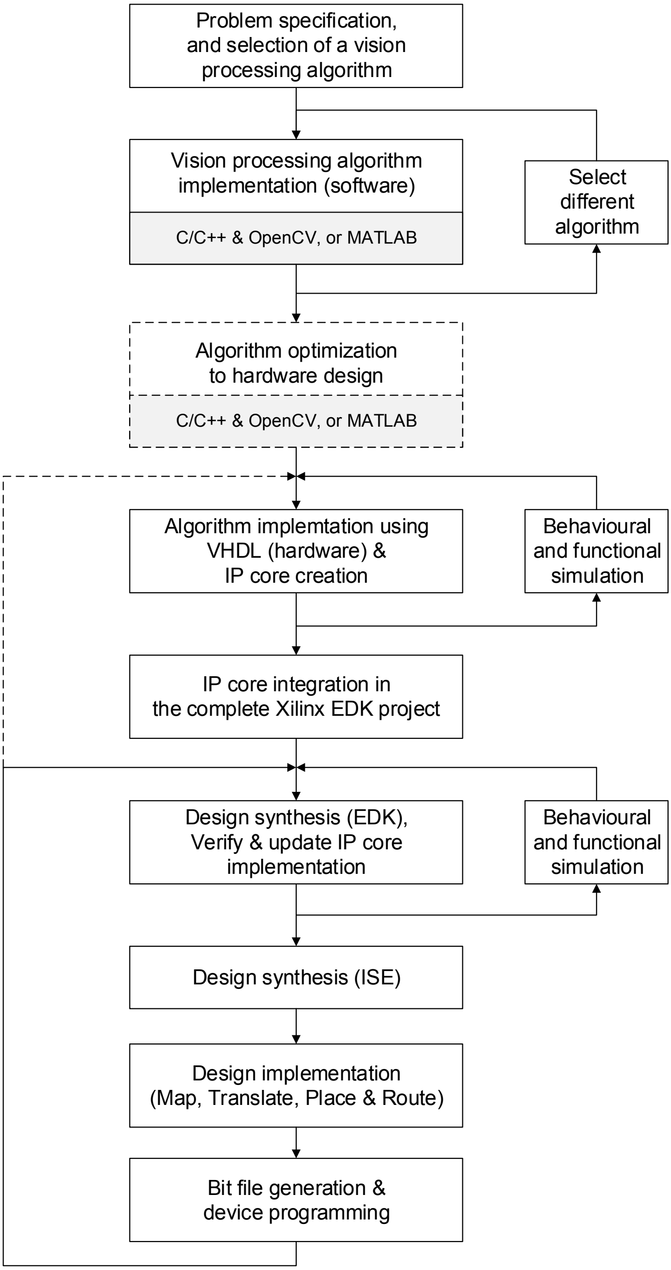

Problem specification, and selection of a vision processing algorithm
Vision processing algorithm implementation (software)
C/C++ & OpenCV, or MATLAB
Select different algorithm
Algorithm optimization to hardware design
C/C++ & OpenCV, or MATLAB
Algorithm implemtation using VHDL (hardware) & IP core creation
Behavioural and functional simulation
IP core integration in the complete Xilinx EDK project
Design synthesis (EDK), Verify & update IP core implementation
Behavioural and functional simulation
Design synthesis (ISE)
Design implementation (Map, Translate, Place & Route)
Bit file generation & device programming

Figure 3.12: The design flow for the implementation of the vision processing algorithms on the FPGA

testbench (executed on ModelSim) reads this text file and converts the binary image data into the utilized input interface (i.e., AXI4-Stream interface) of the IP core. Then, the simulation results from the core are stored in a binary text file. Another MATLAB code uses this binary file to reconstruct and visualize the resulting image. This image, which is the output of the IP core, is compared with the results from the software implementation to verify if it is correct and matches the expected results.

If the simulation results satisfy the requirements, the IP core is integrated into a Xilinx Embedded Development Kit (EDK) project. This is achieved by creating an EDK IP core with all the required ports and interfaces and connecting it to the video processing pipeline in the EDK project. Subsequently, the modified EDK project is synthesized and simulated. The simulation results are validated with the results obtained from the previous step. Then, a Xilinx Integerated Synthesis Enviroment (ISE) (which contains the EDK system, and other components like clock managers, etc.) is synthesized and implemented. The implementation process includes mapping, translating as well as placing and routing the design into the selected FPGA chip. Finally, the bitstream is created and downloaded into the FPGA. If the final results using the real hardware does not match the expected results from the simulation, another debug iteration would be performed by modifying the IP core as needed to achieve the desired results. As an example, an IP core is designed and implemented to perform the demosaicing operation on an input video stream from a camera. The resulting output of this core is the colored (RGB) images. If these results satisfy the requirements, the next vision processing IP core (e.g., white balancing) in the processing chain is designed and implemented, and its results are tested accordingly. This process applies to all the IP cores that are required to realize the targeted system.

3.9 Summary

In this chapter, the methodologies and fundamentals that are required to realize the proposed system are presented, including various video preprocessing algorithms, morphological operations, image thresholding, object segmentation using background subtraction, and graph clustering. Additionally, the concept of multiple object tracking is shown, focusing on the tracking-by-detection approach. Finally, the RAPTOR-X64 system (where the proposed design is realized) and the design flow for the implementation of the vision processing algorithms as IP cores are depicted.

In the next chapter, the proposed reconfigurable vision system for tracking the players is presented in details. This includes the implementation of the different compute-intensive vision processing algorithms on the FPGA and the post-processing on the CPU in the host-PC, realizing a real-time player tracking system.

4 The Proposed Reconfigurable Vision System

In this chapter, the proposed reconfigurable vision system for player tracking in indoor sports is presented. It consists of the FPGA architecture (hardware implementation) and the CPU-based processing system (software implementation) in a host-PC as shown in Figure 4.1. In this system, two stationary GigE Vision cameras attached to the ceiling of the indoor sports hall are used. Each camera has a maximum resolution of 1392x1040 pixels and a frame rate of 30 fps. The cameras are equipped with a Fish-eye lens to have a wider angle and larger coverage of the sports hall. In the proposed system, the compute-intensive vision processing tasks are implemented on the FPGA, while the control and sequential tasks are implemented on the CPU in a host-PC.

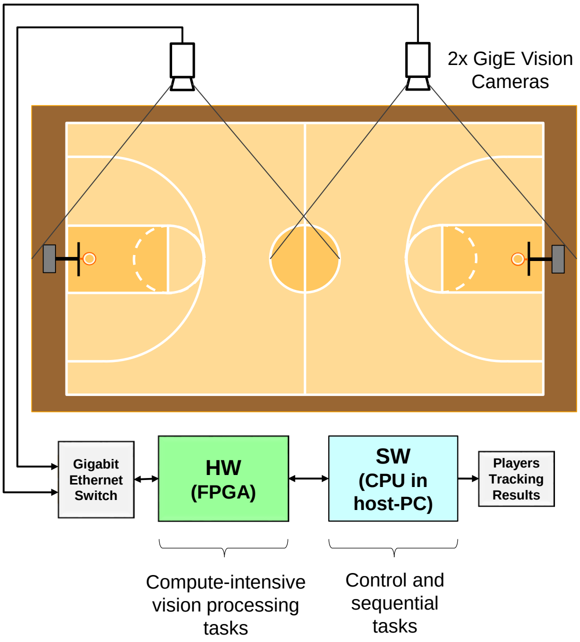


Figure 4.1: A general overview of the proposed system

4.1 System Overview

In this work, player tracking is realized based on the tracking-by-detection approach [60], achieving Multi-Object Tracking (MOT). The task of player tracking can be divided into two main parts:

- Detecting the players in each video frame
- Associating the detections corresponding to the same player over the video frames

In the proposed system, the various compute-intensive pixel-based vision processing operations that are required to detect the players are implemented on the FPGA, while the control-based, less compute-intensive tasks involved in the player tracking are implemented on the CPU in the host-PC. The hardware software partitioning between the FPGA and CPU is shown in Figure 4.2, where the block diagram of the proposed reconfigurable system is presented [107].

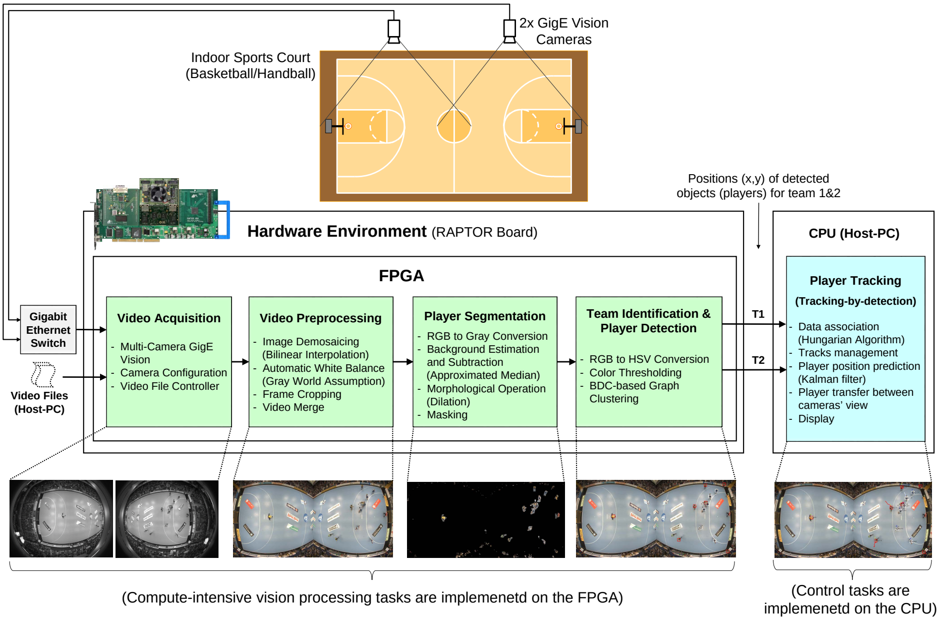

2x GigE Vision Cameras
Indoor Sports Court (Basketball/Handball)
Positions (x,y) of detected objects (players) for team 1&2
Hardware Environment (RAPTOR Board)
CPU (Host-PC)
FPGA
Gigabit Ethernet Switch
Video Files (Host-PC)
Video Acquisition
- Multi-Camera GigE Vision
- Camera Configuration
- Video File Controller
Video Preprocessing
- Image Demosaicing (Bilinear Interpolation)
- Automatic White Balance (Gray World Assumption)
- Frame Cropping
- Video Merge
Player Segmentation
- RGB to Gray Conversion
- Background Estimation and Subtraction (Approximated Median)
- Morphological Operation (Dilation)
- Masking
Team Identification & Player Detection
- RGB to HSV Conversion
- Color Thresholding
- BDC-based Graph Clustering
T1
T2
Player Tracking
(Tracking-by-detection)
- Data association (Hungarian Algorithm)
- Tracks management
- Player position prediction (Kalman filter)
- Player transfer between cameras' view
- Display
(Compute-intensive vision processing tasks are implemenetd on the FPGA)
(Control tasks are implemenetd on the CPU)

Figure 4.2: Top-level block diagram of the proposed reconfigurable system [107]

As shown in Figure 4.2, the proposed system supports two video input sources: Live video acquisition from multiple GigE Vision cameras and recorded games stored in video files. Additionally, the FPGA architecture comprises four modules: video acquisition, video preprocessing, player segmentation, and team identification & player detection. The main outputs of the FPGA are the team identification and the detected positions of the players, which are sent to the host-PC for final player tracking [107].

An overview of the proposed FPGA architecture that is used in the reconfigurable player tracking system is shown in Figure 4.3. The GigE Vision cameras are connected to the FPGA through an Ethernet board equipped with Gigabit physical interfaces. Control packets are sent by the FPGA to configure, start, and stop the cameras, while images are transmitted from the cameras using video packets to the FPGA. The image pixels in these packets are extracted by the video acquisition module. Subsequently, vision processing operations are applied to the extracted pixels in order to detect the players of each team in every video frame.

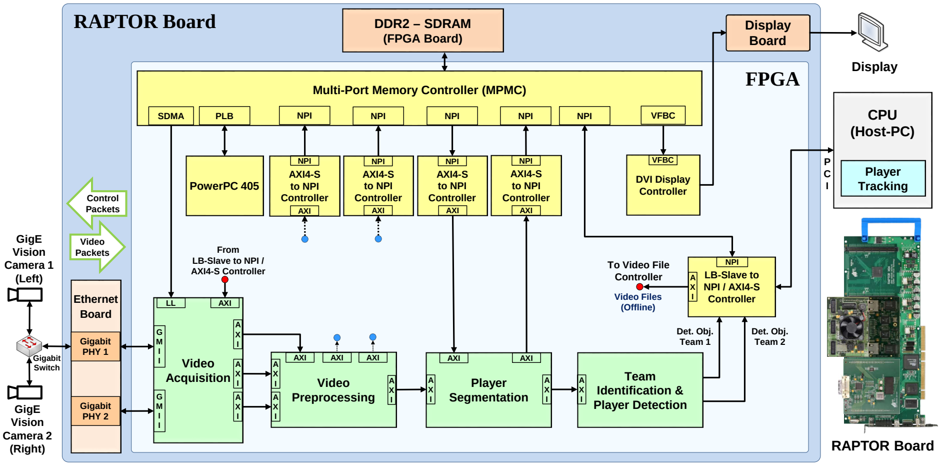


Figure 4.3: An overview of the proposed FPGA architecture

As shown in Figure 4.3, the processing modules (video acquisition, video preprocessing, player segmentation, and team identification & player detection) are connected using a predefined bus interface. Additionally, each one of these modules consists of different IP cores (e.g., performing vision processing operations). In the design of IP cores, it is essential to define and select a bus interface since it does not only define how the different components and IP cores are connected in the system, but also allows the use and re-use of the IP cores. In this work, the AXI4-Stream interface (part of the Advanced Exensible Interface (AXI) protocol [8]) is used for the input and output video streaming in the designed IP cores and the modules. AXI is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) [9], a family of microcontroller buses.

There are three types of AXI interfaces: AXI4 for high-performance memory-mapped requirements, AXI4-Lite for simple and low-throughput memory-mapped communication (e.g., access to control and status registers), and finally the AXI4-Stream for high-speed streaming data. Xilinx adopted the AXI protocol for their IP cores, and it is used in their new FPGA families (e.g., the 7 Series, Zynq, and UltraScale FPGA families).

The AXI4-Stream protocol defines a single channel for streaming data transmission. It can burst an unlimited amount of data [94]. Figure 4.4 shows the AXI4-Stream interface, where two video processing IP cores are connected using this interface. The pixel data are transmitted using the data bus. In this work, different bus width sizes are used based on the function of the IP core, e.g., a data bus of a single bit width is used for a binary frame, 8 bits are used for a grayscale frame, and 32 bits are used for a 4-channel frame with color information. The Start of Frame (SOF) signal is set to logic 1 when the first pixel in every frame is transmitted. End of Line (EOL) indicates the last pixel in each row of a transmitted frame. The "master ready in" and "slave ready out" signals are used for handshaking between two cores. When the slave is ready to receive data, the "slave ready out" signal is set to 1, otherwise a logic 0 is used to halt the data transmission from the master core. Finally, the "data valid" signal indicates that the data on the SOF, EOL, and data bus are valid. These data are processed only when both the "data valid" and the "slave ready out" signals are set to 1 as shown in Figure 4.5, where an example for video data transfer using the AXI4-Stream is presented. In this figure, *P* represents the pixel data of the transmitted video stream.

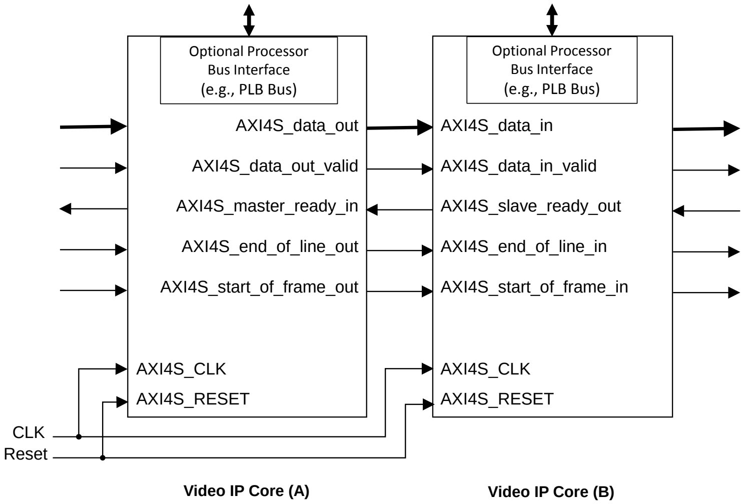


Figure 4.4: Two video IP cores connected using the AXI4-Stream interface

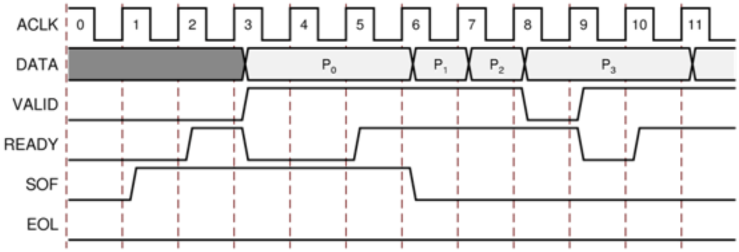


Figure 4.5: An example for pixel data transfer using the AXI4-Stream interface [95]

Figure 4.6 shows the FPGA architecture, where all video processing algorithms in the four modules are realized as IP cores with an AXI4-Stream interface, providing a standardized and easy integration with other cores. These IP cores are designed and implemented using VHDL targeting Xilinx Virtex-4 to 7 Series FPGAs. An embedded processor (either PowerPC (PPC) or MicroBlaze) is utilized to configure the internal registers of the IP cores with the desired parameters [107]. The Xilinx Multi-Port Memory Controller (MPMC) IP core [99] provides access to the external memory (DDR2-SDRAM) through its eight ports. Different interfaces are provided for these ports, including Native Port Interface (NPI), Processor Local Bus (PLB), Video Frame Buffer Controller (VFBC) interface, and Local Link (LL). The Xilinx DVI display controller is used to display video frames that are stored in the external memory. It supports standard video resolutions and frame rates. The LB-Slave to NPI/AXI4-S controller is used for data transfer between the FPGA and the host-PC through the local Bus (LB). This core has an NPI interface for accessing the external memory through the MPMC. Furthermore, it has an AXI4-Stream interface for a direct connection to the Video File Controller IP core. In this case, the video data that are stored in the host-PC can be directly processed by the implemented modules without the need to buffer them in the external memory. Moreover, the LB-Slave to NPI/AXI4-S core is used to transfer the FPGA results (the positions of detected players) to the host-PC for post-processing. The AXI4-S to NPI controller is designed to connect an IP core to the MPMC if the IP core needs to access the external memory (e.g., reading or writing a video frame).

For the evaluations in this work, a Virtex-4 FPGA with an embedded PPC for the control software and a Virtex-7 FPGA are used [107]. In the following, the different modules and the IP cores of the FPGA implementation are depicted, in addition to the player tracking, which is performed in the host-PC.

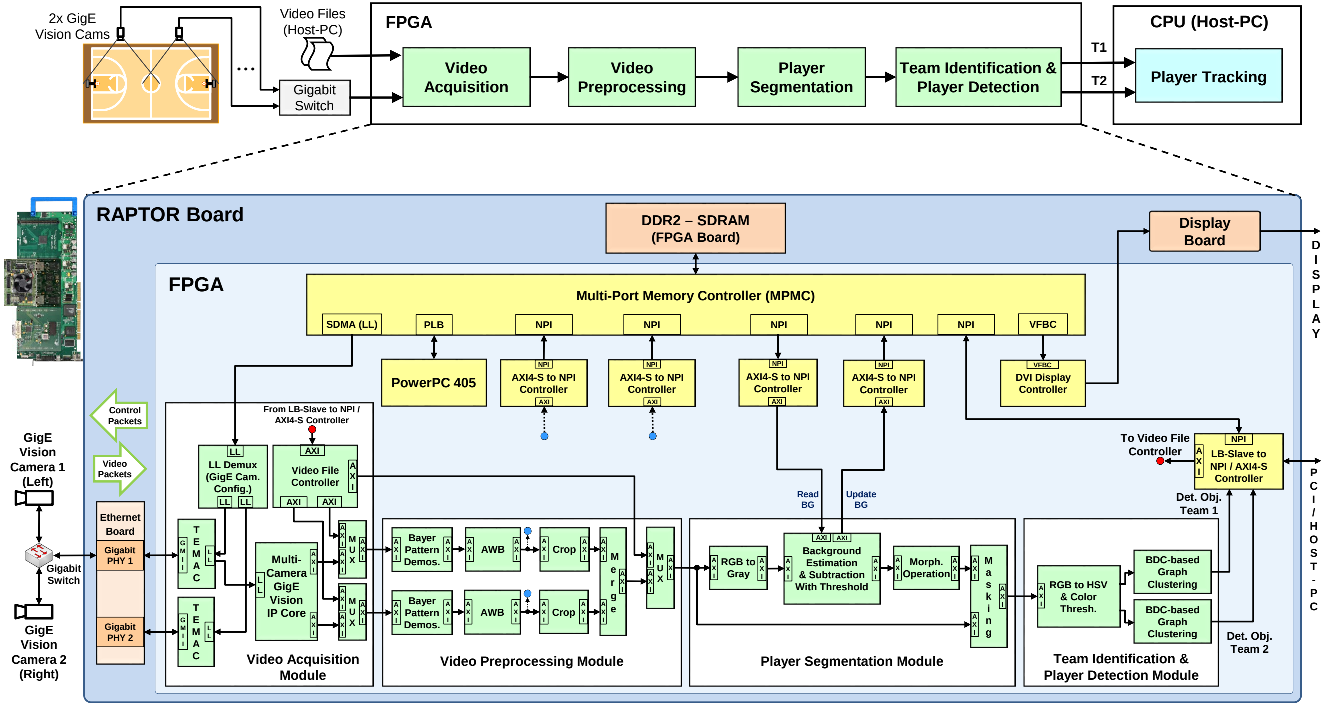


Figure 4.6: The FPGA architecture of the proposed system [108] [107]

4.2 Video Acquisition Module

As depicted in Figure 4.6, this module is used to acquire the live video frames from multiple cameras as well as offline frames from recorded video files stored in the host-PC. The video acquisition module with its various inputs and outputs is shown in Figure 4.7. The Multi-Camera GigE Vision (MC_GigEV) IP core is used to realize the live video frames acquisition from the cameras, while the GigE camera configuration core is used to configure these cameras. The Video File Controller is used for frame acquisition from the offline video files. These IP cores are explained in the following sections.

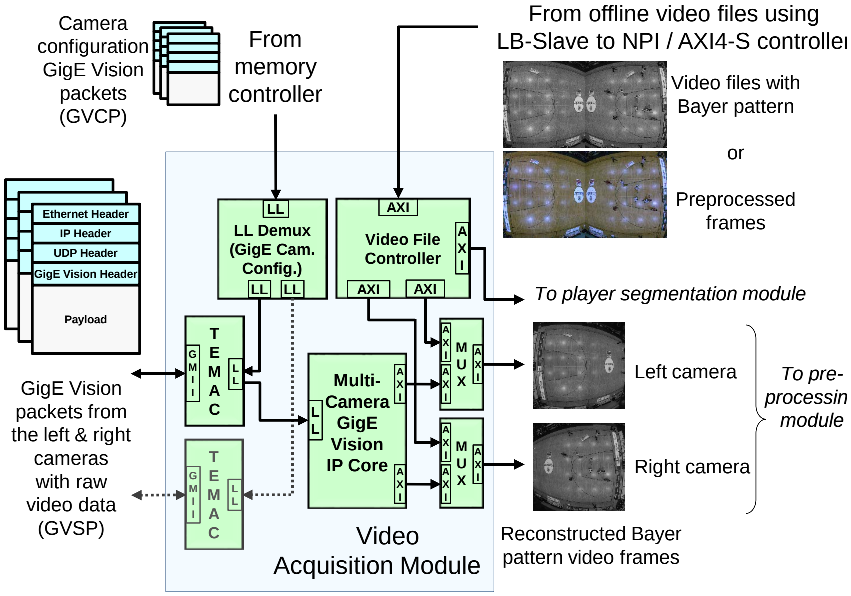
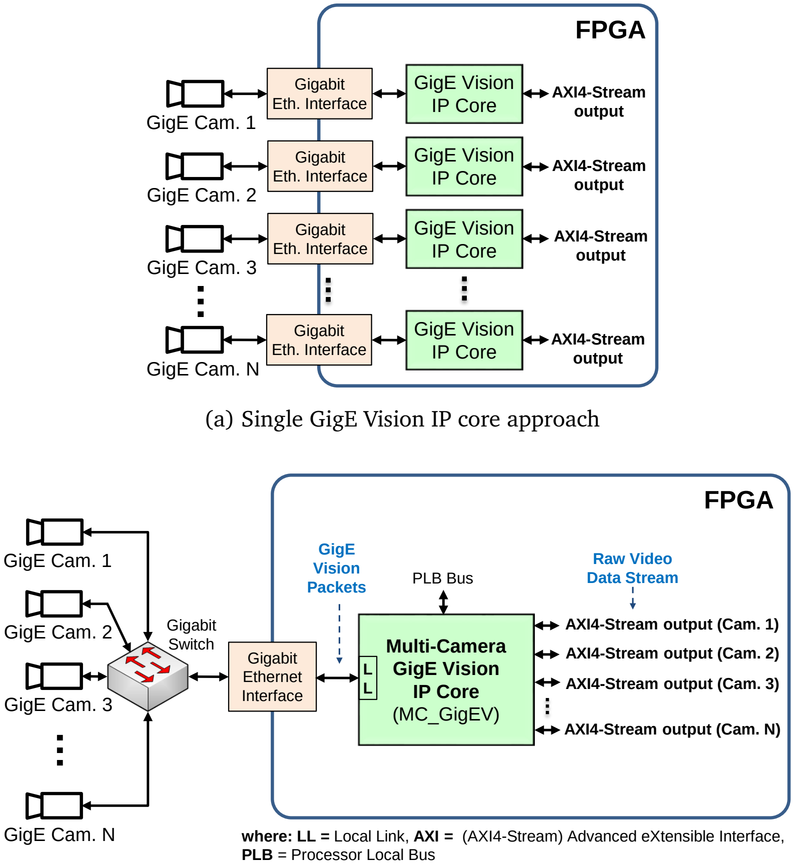

Camera configuration GigE Vision packets (GVCP)
From memory controller
From offline video files using LB-Slave to NPI / AXI4-S controlle
Video files with Bayer pattern
or
Preprocessed frames
Ethernet Header
IP Header
UDP Header
GigE Vision Header
Payload
LL
LL Demux (GigE Cam. Config.)
LL
LL
AXI
Video File Controller
AXI
AXI
AXI
To player segmentation module
G M I I
T E M A C
L L
Multi-Camera GigE Vision IP Core
L L
A X I
A X I
A X I
A X I
M U X
A X I
A X I
A X I
M U X
A X I
Left camera
Right camera
To pre-processin module
GigE Vision packets from the left & right cameras with raw video data (GVSP)
G M I I
T E M A C
L L
Video Acquisition Module
Reconstructed Bayer pattern video frames

Figure 4.7: Video acquisition module

4.2.1 Multi-Camera GigE Vision Core

In this work, a Multi-Camera GigE Vision (MC_GigE Vision) IP core [106] has been developed to realize an online video processing system using multiple cameras. The MC_GigEV IP core is a scalable and resource-efficient core, and it is suitable for space and energy constrained embedded vision systems. Figure 4.8 shows a comparison of possible realizations for multi-camera GigE Vision systems. In Figure 4.8a, each camera is connected to a single-camera GigE Vision IP core. In this case, the complete bandwidth is dedicated to each camera. However, resource requirements are very high since each camera requires its own dedicated GigE Vision IP core and a Gigabit Ethernet interface. The Gigabit Ethernet interface consists of the Ethernet media access controller and the Ethernet PHY. However, in the proposed MC_GigEV IP core shown in Figure 4.8b, only one IP core, and one Gigabit Ethernet interface are needed to connect several cameras. These cameras are connected to the Gigabit Ethernet interface via a Gigabit switch. Here, the one Gigabit Ethernet bandwidth will be shared between all connected cameras, but the resource requirements on the FPGA are significantly reduced compared to the single GigE IP core approach presented in Figure 4.8a [106].



(a) Single GigE Vision IP core approach

where: LL = Local Link, **AXI =** (AXI4-Stream) Advanced eXtensible Interface, **PLB** = Processor Local Bus

(b) The proposed mulit-camera GigE Vision IP core

Figure 4.8: FPGA-based systems for multiple GigE Vision cameras [106]

Figure 4.9 shows the MC_GigEV IP core in a multiple GigE Vision camera system. It consists of the Tri-Mode Ethernet Media Access Controller (TEMAC), the MC_GigEV, and the camera configuration IP cores. In this system, two MC_GigEV IP cores are used to connect a (N+M) number of GigE Vision cameras as shown in Figure 4.9. The video stream from each camera consists of GigE Vision packets, encapsulating the raw video data. These GigE Vision packets are received by the TEMAC IP core [98] through its Gigabit Media Independent Interface (GMII). The TEMAC core is responsible for the implementation of the link and of the physical layers, and it passes the packets from different camera sources to the developed MC_GigEV IP core. The MC_GigEV IP core processes the GigE Vision packets, extracts the raw video data and reconstructs the video frames from each video stream. Finally, the core provides the extracted video data as AXI4-Streams in separated channels for each video stream so that the video data can be easily processed further or stored in the on-board memory. To configure the cameras with the desired frame rates and resolutions, GigE Vision control packets are sent to the desired camera through the camera configuration IP core (Cam_Config) [106]. More details about this camera configuration core are shown in Section 4.2.2.

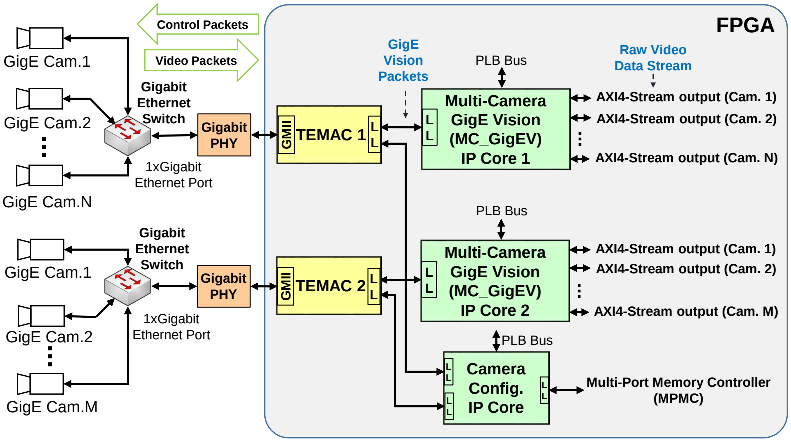


Figure 4.9: A multi-camera GigE Vision system using the MC_GigEV IP core

The proposed MC_GigEV IP core is designed and implemented in hardware on a Xilinx Virtex-4 FPGA to receive and extract the video data streams that are transmitted from GigE cameras using the GVSP protocol. The core can be easily integrated into the Xilinx tool-flow as an EDK IP core utilizing a PLB interface. The core registers are initialized and configured with the desired parameters, e.g., the MAC addresses of the cameras and the number of GVSP packets per frame. This is done by an embedded CPU like Microblaze or PowerPC. For a resource-efficient realization, the core is designed with a generic parameter for the number of connected cameras so that the implementation is generated accordingly [106]. A block diagram of the core is shown in Figure 4.10 for four cameras. A flowchart that shows how the IP core reconstructs a video frame from a GigE Vision camera is depicted in Figure 4.11

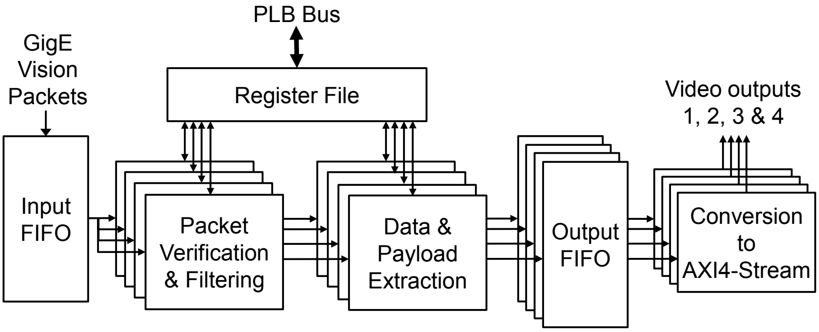

PLB Bus
GigE Vision Packets
Register File
Video outputs 1, 2, 3 & 4
Input FIFO
Packet Verification & Filtering
Data & Payload Extraction
Output FIFO
Conversion to AXI4-Stream

Figure 4.10: The multi-camera GigE Vision core block diagram [106]

As shown in Figures 4.10 and 4.11, the incoming GigE Vision packets from different cameras are buffered in a single First-In First-Out (FIFO) memory and subsequently filtered based on the MAC address of the sending camera. Verification of the IP protocol and UDP source port number are performed. As depicted in Section 2.4.1, there are three types of packets for the standard transmission mode using the GVSP protocol: data leader, data payload, and data trailer packets as shown in Figure 2.12. The core distinguishes between these packets based on the packet_format field in the GVSP header. The frame height and width are extracted from the data leader packet. The raw video data are extracted from the sequential data payload packets based on their packet_id field in the GVSP header. The extracted raw data are buffered in a separate output FIFO for each camera. The core can detect packet losses that may happen during video transmission. Input data width of the core is 32-bit and the output can be 8-bit or 32-bit depending on the system requirements. The core output is designed and implemented using the AXI4-Stream interface to provide a standard and easy integration with other video processing cores. The MC_GigEV IP core works on wire speed and extracts the incoming video data as soon as the video packets are received by the FPGA. It is implemented to handle standard Ethernet packets as well as Jumbo frames up to 9014 Bytes. Additionally, this IP core is validated in real hardware using up to four GigE Vision cameras for one core instance [106].

For the proposed player tracking system in this work, two GigE Vision cameras are used to provide a top-view with full coverage of the court in the sports hall as stated earlier in this chapter. Although two GigE Vision cameras are used in the current system

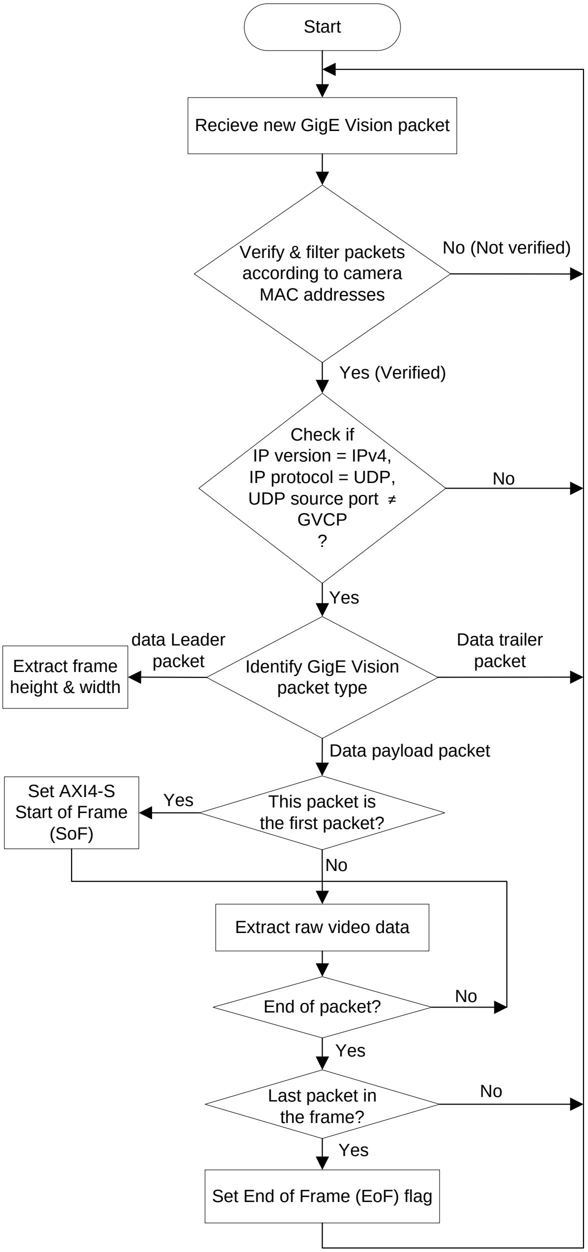

Start
Recieve new GigE Vision packet
Verify & filter packets according to camera MAC addresses
No (Not verified)
Yes (Verified)
Check if IP version = IPv4, IP protocol = UDP, UDP source port ≠ GVCP ?
No
Yes
Extract frame height & width
data Leader packet
Identify GigE Vision packet type
Data trailer packet
Data payload packet
Set AXI4-S Start of Frame (SoF)
Yes
This packet is the first packet?
No
Extract raw video data
End of packet?
No
Yes
Last packet in the frame?
No
Yes
Set End of Frame (EoF) flag

Figure 4.11: Flowchart for a video frame reconstruction from a GigE Vision camera using the MC_GigEV IP core

setup, the implemented MC_GigEV supports live video frame acquisition from multiple GigE Vision cameras. This scalability is required if additional cameras are needed to support more features in the system (e.g., player identification using the digits on the players' jerseys) [107]. The video stream from each camera consists of GigE Vision packets, encapsulating the raw video data. For the used GigE Vision cameras in this work, the total number of GigE Vision data payload packets that are used to transfer one frame from one camera is calculated using Equation 4.1. These packets carry the standard Ethernet payload, which is 1464 bytes for each packet.

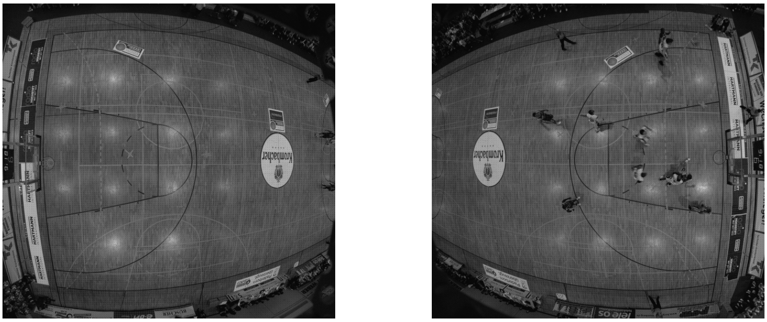
$$\text{Number of data payload packets} = \frac{\text{Camera Resolution} \times \text{Nr. of Bytes/pixel}}{\text{packet payload}} \tag{4.1}$$

$$\text{Number of data payload packets} = \frac{1392 \times 1040 \times 1}{1464} = 989 \text{ packets/frame/camera}$$

Furthermore, These packets are transferred using a Gigabit Ethernet switch which is connected to the Gigabit Ethernet interface of the FPGA board as shown in Figure 4.6 and Figure 4.7. Hence, only one Gigabit Ethernet interface is needed to connect two or more cameras sharing the bandwidth of the Gigabit connection, thus reducing the required resources for interfacing the cameras. In the FPGA, the GigE Vision packets are received by the TEMAC, which passes the packets from different camera sources to the Multi-Camera GigE Vision (MC_GigEV) IP core. The MC_GigEV IP core extracts the raw video data, reconstructs the video frames from the GigE Vision packets (989 packets for one video frame from each camera), and passes these data to the video preprocessing module for further processing [108]. The reconstructed frames, which are encoded using Bayer pattern, are shown in Figure 4.12a and 4.12b.

4.2.2 GigE Vision Camera Configuration

For camera configuration and control channel implementation, a light-weight implementation is chosen to reduce resource requirements and complexity of the system. The light-weight implementation is especially suitable for embedded vision systems with low power and logic resource budget. The implemented Cam_Config IP core sends GVCP packets (which are stored in the external memory) to configure the attached GigE cameras with the required parameters (such as IP address, image resolution, frame rate, and data format). Additionally, it starts and stops the video acquisition from the cameras. This control-dominated task is implemented in software and executed on the embedded CPU in the FPGA [108].



(a) From the left camera

(b) From the right camera

Figure 4.12: Reconstructed frames (Bayer pattern) from GigE Vision packets

As shown in Figures 4.7 and 4.9, this IP core is used as a LL interface multiplexer controlled by the embedded CPU, connecting multiple TEMAC cores to the MPMC controller if additional Ethernet interfaces are needed to connect more cameras. In this case, only one port in the MPMC controller is used to configure multiple GigE Vision cameras. The LL interface specification defines a high-performance, synchronous, point-to-point connection [97]. After the desired TEMAC core is connected to the MPMC, the GVCP packets (cf. Figure 2.11) are sent by the embedded CPU to the desired GigE Vision camera based on the MAC address of the camera. This address is stored in the Ethernet header of the GVCP packet.

An example of a GVCP packet that is used to configure a camera with a frame width is shown in Figure 4.13. This packet is used to write the desired frame width value in the frame width register of the camera, and it is based on the GigE Vision protocol specifications defined in [13]. In this packet, the GVCP packet's header and its payload are shown in detail. The GVCP header is a command header (8 Bytes). The 0x42 is the value used to identify the GVCP packets in the protocol. A *flag* value of 0x01 requires the recipient of this packet to send an acknowledgment packet. The WRITEREG_CMD (0x0082) is used for the *command* field, indicating this packet is for writing a value in a register. The *Length* of the GVCP payload is set to 8 Bytes for the register address and the written value. *Req_id* is a sequential number given to the packet. The GVCP payload consists of a 4 Byte *register_address* (e.g., 0x0000D300 for the frame width register), and a 4 Byte *register_data* (e.g., 1392 or 0x00000570 for the value of the desired frame width). The total size of a GVCP packet for the WRITEREG_CMD command with one register to be written is 58 Bytes as shown in Figure 4.13.

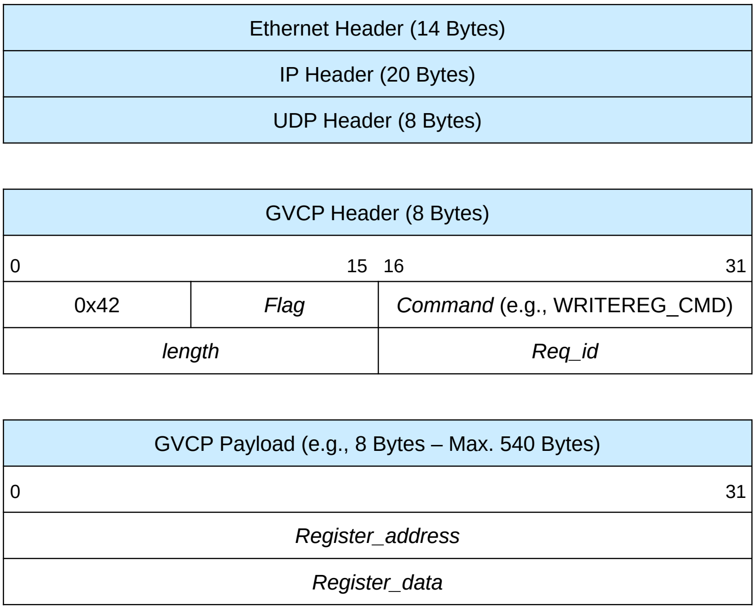


Figure 4.13: GVCP packet with WRITEREG_CMD [13]

4.2.3 Video File Controller

For the offline video file processing, two video files are used, each storing the video data from one camera. The offline video frames are read from the host-PC through the Local Bus (LB)-Slave to Native Port Interface (NPI)/AXI4-Stream controller as shown in Figure 4.6. Then, these frames are sent directly to the video file controller without the need for buffering using an external memory. The video file controller is used to decode the incoming video stream based on the selected video format, and to output the resulted video streams for further processing. In this work, this controller is designed to support two types of data (video formats) in video files: raw data with Bayer pattern (8 bits are used for each pixel), and RGB data (each pixel is represented by 24 bits) after preprocessing as shown in Figure 4.7.

The core receives a video stream through its AXI4-Stream input interface with a data width of 32-bit. For a Bayer pattern video, these 32 bits represent four pixels (8 bits for each). While for the RGB video input, the transferred data in every clock cycle represent color components of two pixels (e.g., RGB values for one pixel (24 bits) and an R value for the next pixel (8 bits)). The core's registers are used to configure the controller with the video format and the resolution of the two video files. The width of these files (image width 1 and 2 for the first and second video file) could be of an equal or different size. However, their height must be the same. The block diagram of the video controller core is shown in Figure 4.14. As can be seen, the core stores the input video stream in an input FIFO, where a Finite State Machine (FSM) is implemented to read this FIFO and process the data based on the selected video format. The flowchart of this FSM is shown in Figure 4.15. If the Bayer pattern video format is selected, the received data is buffered in FIFO1 until the number of buffered pixels is equal to the image width 1. Next, the received data is buffered in FIFO2 until the number of these pixels is equal to the image width 2. The process of writing an incoming video frame with Bayer pattern into these output FIFOs is shown in Figure 4.16. Afterward, the outputs from the two FIFOs are converted to AXI4-Stream with an output width of 8-bit for each video as shown in Figure 4.14. This 8-bit output corresponds to the raw video data (Bayer pattern) of one pixel, which is sent to the preprocessing module. Regarding the clock domains of the core's FIFOs, an independent clock domain is used for the input FIFO since the write clock (Wr_Clk) for this FIFO is synchronized with the clock of the LB-Slave to NPI/AXI4-Stream controller. On the other hand, the read clock (Rd_Clk) of the input FIFO is synchronized with the clock of the output FIFOs, by which a common clock domain is utilized for the writing and reading operations of these output FIFOs as shown in Figure 4.14.

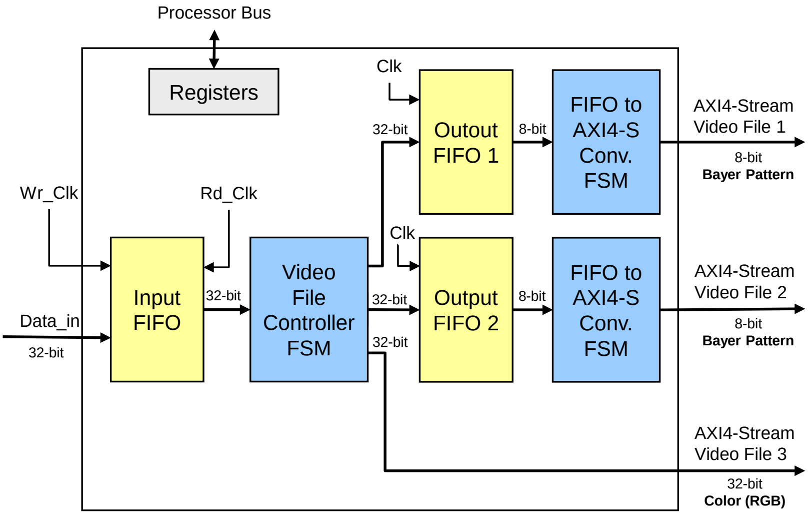


Figure 4.14: Video file controller block diagram

If the RGB video format is selected for the incoming video stream, a color component alignment in the controller's FSM is applied to these data as shown in Figure 4.15.

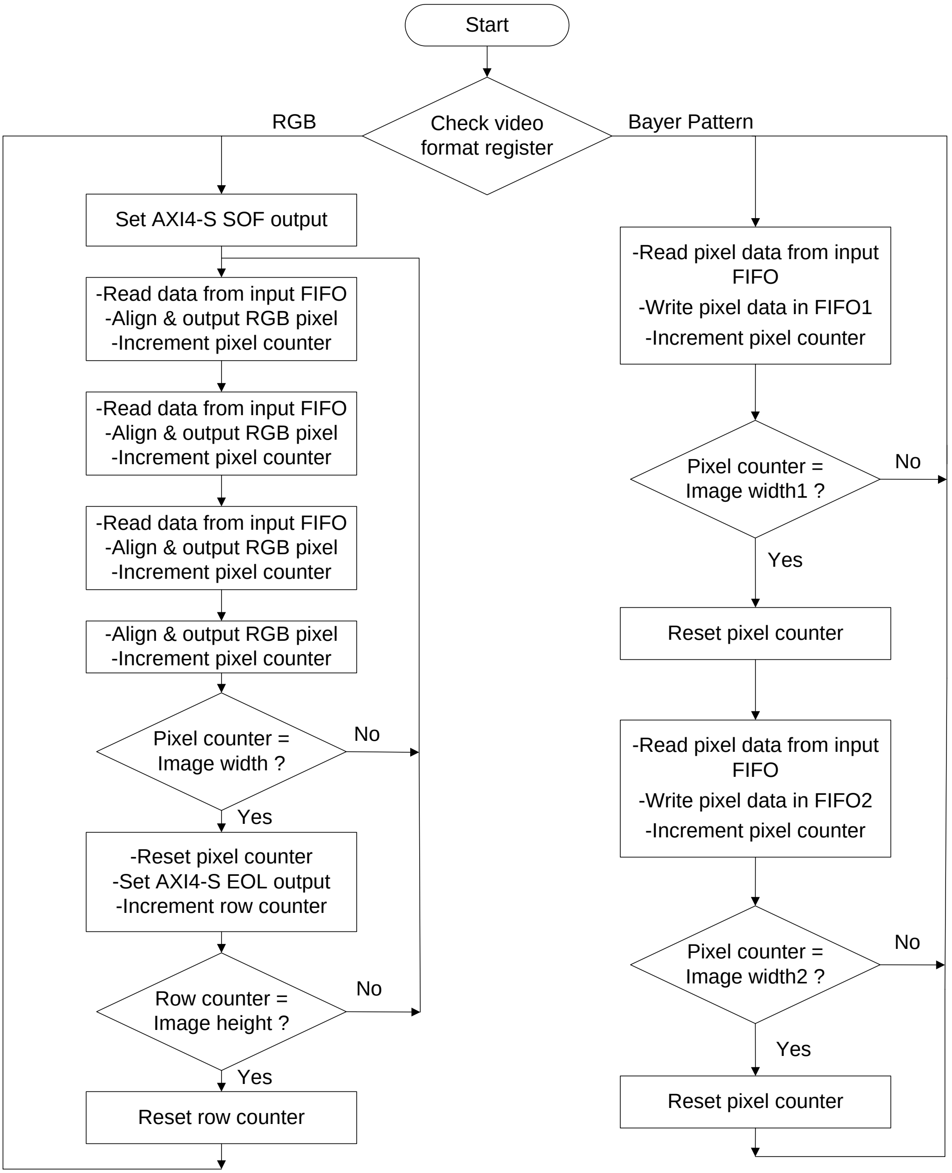

Start
Check video format register
RGB
Bayer Pattern
Set AXI4-S SOF output
-Read data from input FIFO
-Align & output RGB pixel
-Increment pixel counter
-Read data from input FIFO
-Align & output RGB pixel
-Increment pixel counter
-Read data from input FIFO
-Align & output RGB pixel
-Increment pixel counter
-Align & output RGB pixel
-Increment pixel counter
Pixel counter = Image width ?
No
Yes
-Reset pixel counter
-Set AXI4-S EOL output
-Increment row counter
Row counter = Image height ?
No
Yes
Reset row counter
-Read pixel data from input FIFO
-Write pixel data in FIFO1
-Increment pixel counter
Pixel counter = Image width1 ?
No
Yes
Reset pixel counter
-Read pixel data from input FIFO
-Write pixel data in FIFO2
-Increment pixel counter
Pixel counter = Image width2 ?
No
Yes
Reset pixel counter

Figure 4.15: Video file controller flowchart

This alignment is needed since the received 32-bit data contain color components of the current and the coming pixels as shown in Figure 4.17. Therefore, this process produces RGB components that belong to the same pixel (24 bits) appended to an 8-bit value (e.g., zero) for every clock cycle, resulting in a 32-bit output. In this case, the RGB values of one pixel are aligned to one clock cycle. This alignment is required since, in this work, one RGB pixel is processed in every clock cycle. In the next step, the output frames from the video acquisition module are processed by the video preprocessing module.

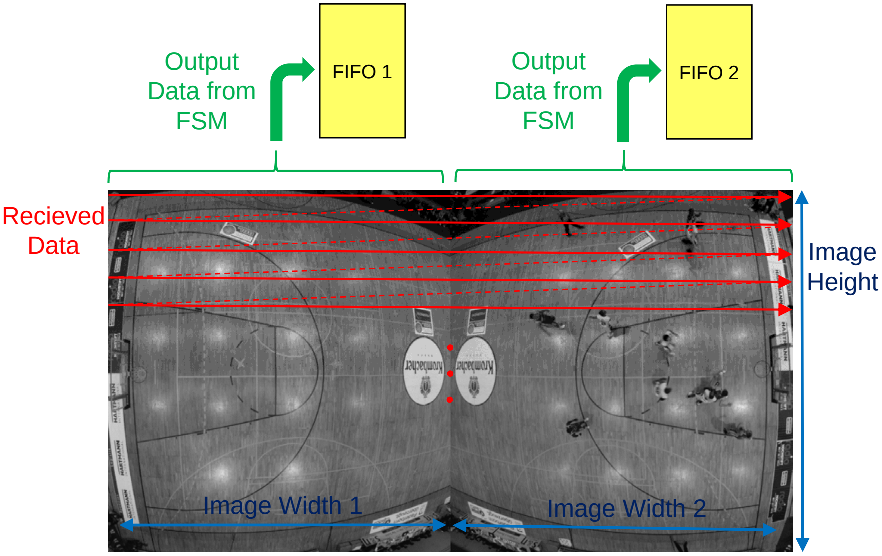


Figure 4.16: Video file controller operation for Bayer pattern input video

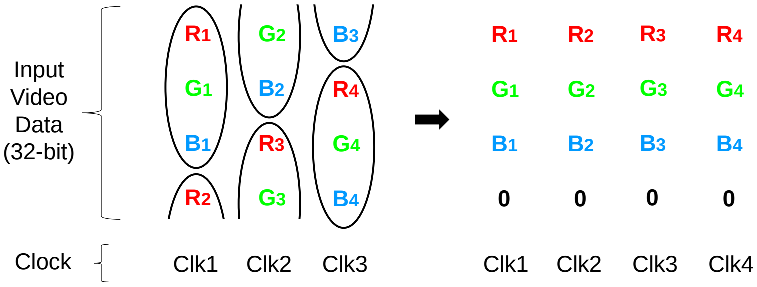


Figure 4.17: RGB color components alignment

4.3 Video Preprocessing Module

The video preprocessing module includes Bayer pattern demosaicing, automatic white balance (AWB), video cropping, and merging IP cores as shown in Figure 4.18.

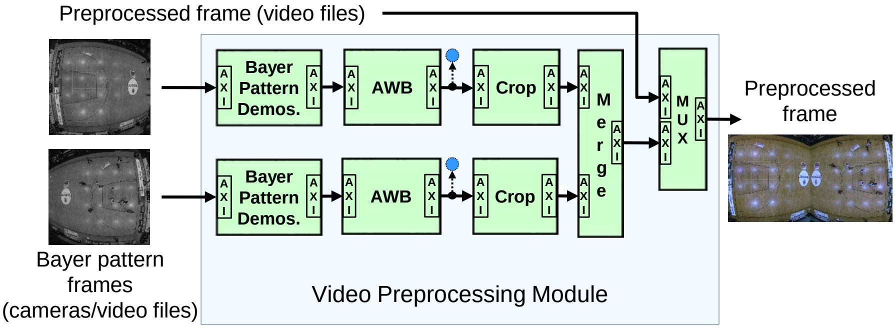

Preprocessed frame (video files)
Bayer Pattern Demos.
AXI
AWB
Crop
Merge
MUX
Preprocessed frame
Bayer pattern frames (cameras/video files)
Video Preprocessing Module

Figure 4.18: Video preprocessing module

4.3.1 Bayer Pattern Demosaicing

As depicted in Section 3.1.1, demosaicing is necessary to retrieve the missing RGB values for each pixel and to form a full-color image. The implemented demosaicing algorithm is using bilinear interpolation with an eight neighborhood filter. A block diagram of the hardware implementation of Equations 3.1, 3.2, and 3.3 is shown in Figure 4.19 [15]. Two row-buffers are used to form a 3x3 window, which is required for the interpolation process. These row buffers are of a variable depth to adapt to different resolutions. The division by two is free in hardware. The multiplexer outputs depend on which pixel is currently processed. Additionally, the core is connected to the embedded PowerPC through the PLB Bus to configure and initialize the core registers with the desired frame resolution and to select one of the Bayer patterns (RGGB, BGGR, GRBG, GBRG) [106]. For every raw input pixel encoded with Bayer pattern (8 bits/pixel), the RGB color components are interpolated, resulting in 24 bits for every pixel. In this implementation, the raw input pixel is not neglected after interpolation, but it is propagated to the core's output. Therefore, it is concatenated with its resulted RGB components forming a 32 bits output as shown in Figure 4.19. This is useful if there is a need to store the raw input video frames from the camera for future usage. The output frames after demosaicing from the left and right cameras are shown in Figures 4.20a and 4.20b, respectively.

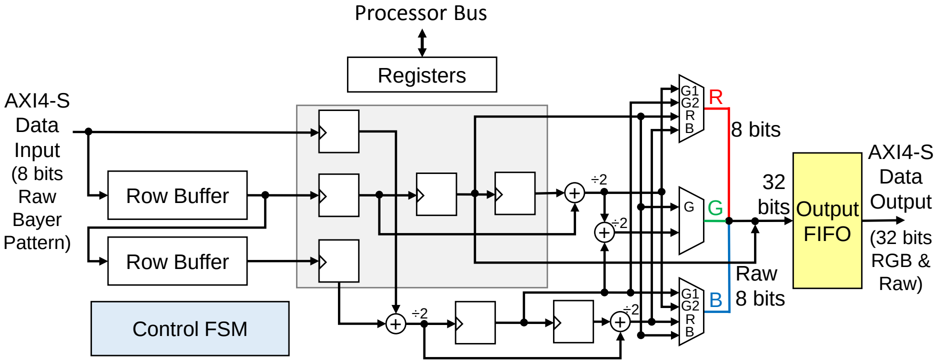
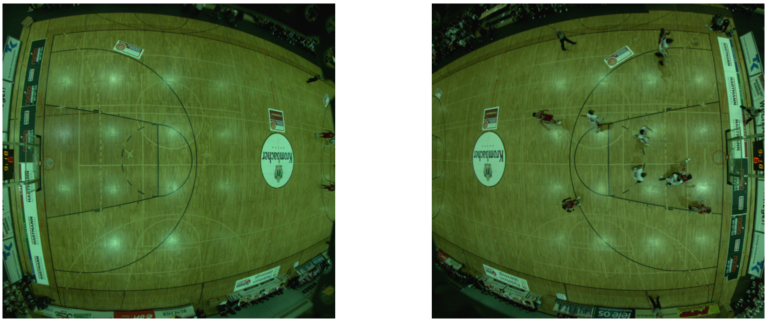


Figure 4.19: Implementation of Bayer pattern using bilinear interpolation [15]



(a) Left camera

(b) Right camera

Figure 4.20: Resulted colored images after Bayer pattern demosaicing using bilinear interpolation [108] [107]

4.3.2 Automatic White Balancing

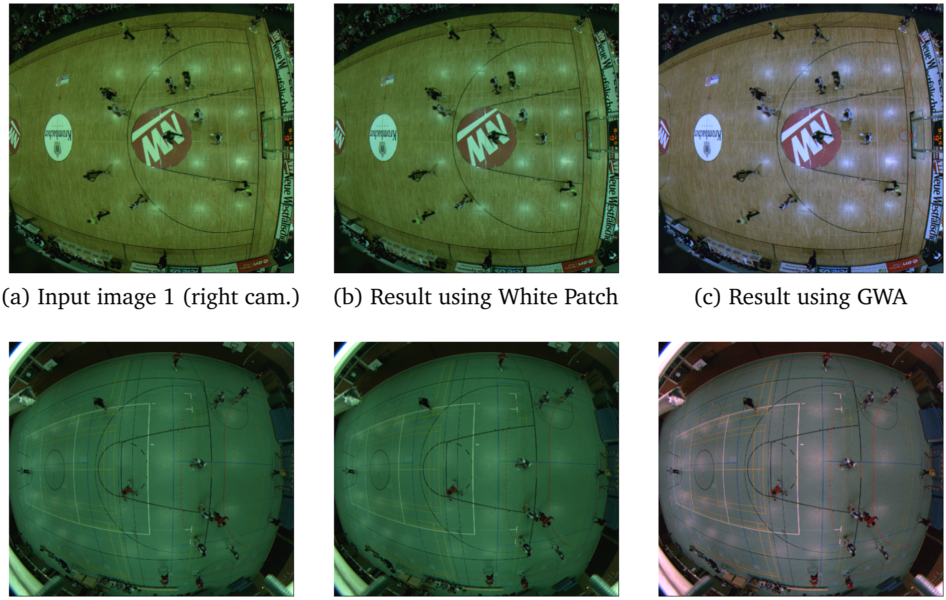
White balance is used to correct the color bias in the images as stated earlier in Section 3.1.2. In this work, two algorithms have been evaluated: the Gray World Assumption (GWA) [52] and the White Patch (WP) [54] algorithms. These two methods have their respective strengths, and a closer investigation is made in [52]. It is observed that for most images, the two methods produce different results [52]. The GWA and the White Patch algorithms are implemented using C++ and OpenCV library, and their performance is evaluated. Figure 4.21 shows images of indoor sports halls and the evaluation results using the two AWB algorithms. As can be seen, the results from the GWA algorithm (cf. Figure 4.21c and 4.21f) are better than the results from the White Patch algorithm (cf. Figures 4.21b and 4.21e). The presence of white pixels (or bright pixels) in the used images, results in the poor performance achieved by the White Patch algorithm. A white pixel (i.e., R, G, and B values = 255), causes the calculated gain values (using Equations 3.4, 3.5, and 3.6) to be equal to one. In this case, the output pixels after white balance are equal to the input pixels values, i.e., these input pixels are not corrected since they are multiplied by 1. To avoid the disturbances to the calculation caused by a few white or bright pixels, clusters of pixels or lowpass the image can be applied [43]. In the used image, white pixels exist due to illumination (e.g., light reflections on the court) in the sports hall. Additionally, there are several objects with white color in these images (e.g., white stripes on the court, and white jerseys of players) as shown in Figure 4.21.

Based on the results of the performed simulations shown in Figure 4.21, the Gray World Assumption (GWA) algorithm (presented in Section 3.1.2) is chosen for the FPGA implementation in this work. As shown in Equations 3.7 and 3.8, five divisions are required to calculate the gain values for the red and blue channels in the GWA algorithm. These divisions require a huge amount of the FPGA logic resources. Therefore, Equations 3.7 and 3.8 are modified for a resource-efficient IP core implementation. This modification is shown in Equation 4.2, where the gain values for the red and blue channels are calculated. In this case, only two divisions instead of five are used, without affecting the algorithm performance. For the green channel, the gain value is equal to 1.

$$Gain_R = \frac{\sum_{x=1}^{M}\sum_{y=1}^{N} I_G(x,y)}{\sum_{x=1}^{M}\sum_{y=1}^{N} I_R(x,y)}, \text{ and } Gain_B = \frac{\sum_{x=1}^{M}\sum_{y=1}^{N} I_G(x,y)}{\sum_{x=1}^{M}\sum_{y=1}^{N} I_B(x,y)} \tag{4.2}$$

where: M and N are the width and height of the image, respectively.

The block diagram of the implemented AWB IP core using the GWA algorithm is shown in Figure 4.22. This implementation is based on Equation 4.2. As depicted in Figure 4.22, the input video stream is buffered in an input FIFO. For each color component



(a) Input image 1 (right cam.) (b) Result using White Patch (c) Result using GWA

(d) Input image 2 (right cam.) (e) Result using White Patch (f) Result using GWA

Figure 4.21: Evaluation results of the AWB algorithms

(i.e., R, G, and B), an adder is utilized to calculate the cumulative sum value of that component for all the pixels in one frame. At the end of each frame, the two dividers use these sum values to calculate the gain for the red and blue channels as shown in Figure 4.22. These gain values are multiplied by their corresponding color components in the next frame since there are usually no significant changes in the illumination between two consecutive frames using a high frame rate video stream (e.g., 30 fps). This process is applied to all frames to adapt to different lighting conditions [108]. The multipliers are implemented using dedicated DSP slices in the FPGA with three pipeline stages for each multiplier, achieving the optimum performance. If there is an overflow in the multiplication results (i.e., the result is larger than 255), it is adjusted to 255 as depicted in Figure 4.22. Finally, the white balanced pixels are written to the output FIFO, and a conversion to the AXI4-Stream interface is performed. Since the core uses stream-based processing, both the input and output FIFOs are implemented as small FIFOs with the depth of 32 Bytes for each FIFO using the FPGA's distributed RAM. To avoid data loss when the output FIFO is full, the programmable full control signal of the output FIFO is used to process the three pixels in the multiplier pipeline. In this case, the programmable full signal (which is driven high before the FIFO's full signal is triggered) is used to halt the incoming video stream so that the intermediate pixels in the pipeline are processed, and the results are stored in the output FIFO. For the implemented AWB core, the total latency is 12 clock cycles. The resulted frames after AWB from the left and right camera are shown in Figure 4.23. If it is desired to store the resulting white balanced frames in the host-PC (e.g., for later evaluation or visualization), the output of the AWB core can be connected to an AXI4-S to NPI controller (cf. Figure 4.6) to buffer the resulting frames in the external memory. After that, the LB-Slave to NPI/AXI4-S controller can be used to read the buffered frames and send them to the host-PC for storage. Finally, video frame cropping is applied to the resulting frames of the AWB core as depicted in the next subsection.

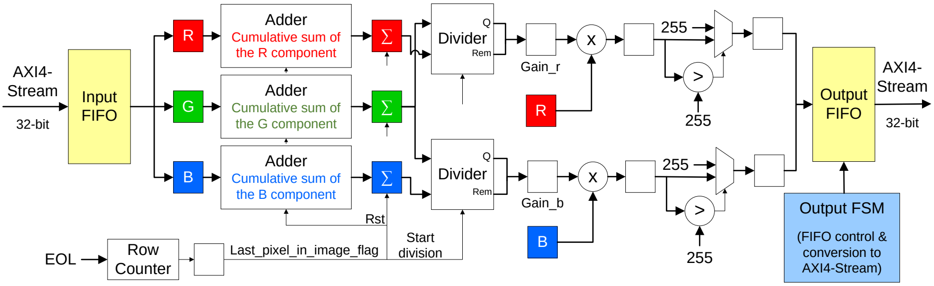
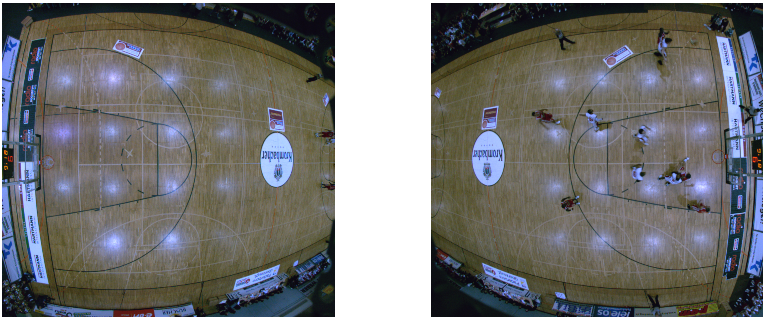


Figure 4.22: Block diagram of the implemented AWB using Gray World Assumption



(a) Left camera

(b) Right camera

Figure 4.23: Resulted colored images after AWB using the GWA algorithm

4.3.3 Video Cropping

Since the input video frames contain information that is not of interest (e.g., the spectators), the frames are cropped so that only the region of interest (the court) is preserved. Additionally, the overlapping region between the left and right half of the court is reduced [108]. The block diagram of the implemented cropping core is shown Figure 4.24. The image cropping dimensions are given to the core through its programmable registers. This includes the x and y coordinates for both the starting and ending of the image cropping. As shown in Figure 4.24, the cropping operation is performed using two counters, a pixel and row counter, controlled by the cropping FSM. Based on these counters values and the given cropping dimensions, the cropping process is applied on the input RGB frame. The cropped images from the left and right camera are shown in Figure 4.25.

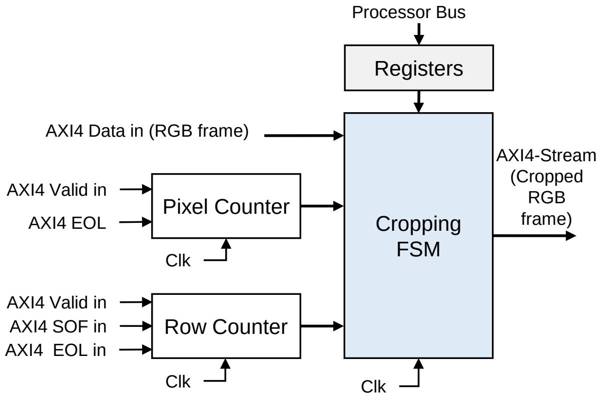
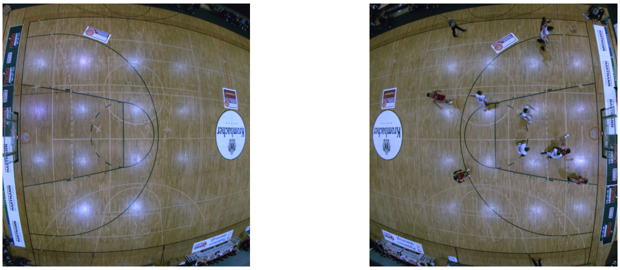


Figure 4.24: Block diagram of the implemented cropping IP core



(a) Left camera

(b) Right camera

Figure 4.25: Resulted images after cropping

4.3.4 Video Frame Merger

Finally, the two cropped video frames are merged using the Video Frame Merger IP core, providing one big frame that covers the whole court as shown in Figure 4.26 [108].

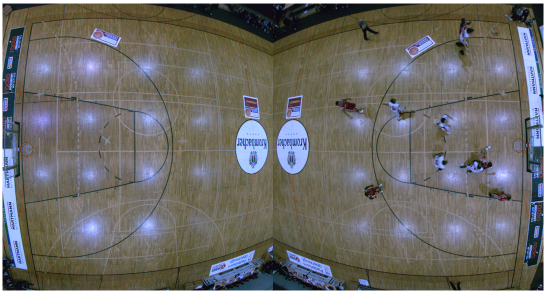


Figure 4.26: Output frame after merging [108] [107]

The block diagram of the Video Frame Merger IP core is shown in Figure 4.27. The core's registers are used to store the frame resolution of the two input videos using a processor bus (e.g., a PLB bus). In this implementation, a varying width size of the two input videos is supported. However, their height must be the same. These input video streams are buffered into two FIFOs as shown in Figure 4.27. An FSM is implemented to read the buffered video data, merge them, and output the merged video as an AXI4-Steam. This merging operation is based on the pixel counter in the FSM, by which this counter is incremented with every pixel being read from the input FIFOs. First, the pixel values are read from FIFO 1 until the pixel counter is equal to the frame width of the first input video stream. Afterwards, the pixel counter is reset to zero, and the FSM starts reading the pixel values from FIFO 2 until the pixel counter is equal to the frame width of the second video stream. These steps are repeated until the row counter is equal to the height of the input videos, indicating the end of the video frame. The output from the video preprocessing module is the merged colored (RGB) frame, which is used by the player segmentation module to extract the RGB foreground (including the players) as explained in the next section.

4.4 Player Segmentation Module

In this work, the approach to segment the players is based on background subtraction. The player segmentation module includes RGB to grayscale conversion, background estimation and subtraction, morphological operation, and masking as shown in Figure 4.28 [107].

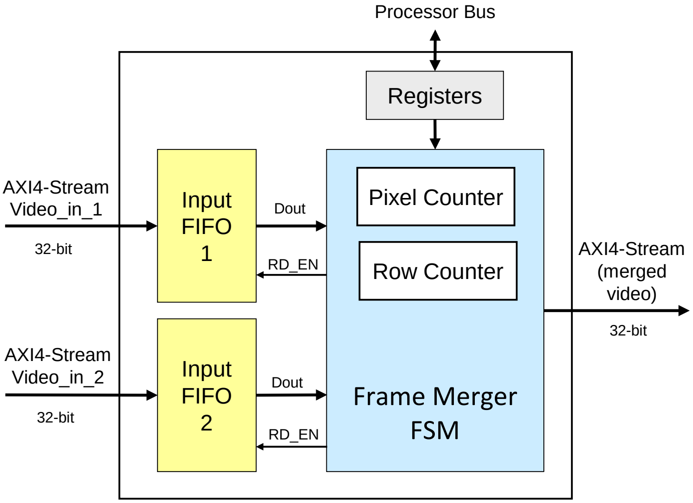


Figure 4.27: Block diagram of the Video Merger IP core

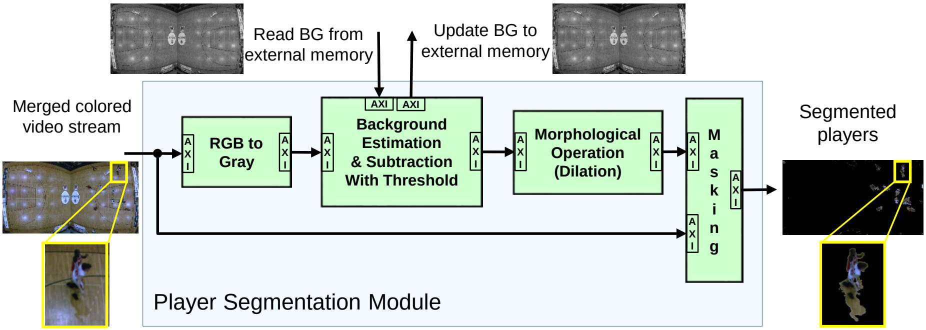


Figure 4.28: Player segmentation module

4.4.1 RGB to Grayscale Converstion

The color space of the merged frame (Figure 4.26) is converted from RGB to grayscale to be used for background estimation and subtraction [107]. The block diagram of the implemented RGB to grayscale IP core is shown in Figure 4.29. It computes the grayscale value for each pixel by forming a weighted sum of the R, G, and B components according to Equation 3.10. The incoming video stream is buffered in an input FIFO. Each color component is multiplied by the appropriate weight value, producing one output pixel per input sample. Three multipliers are used in this implementation, and each multiplier uses one dedicated DSP slice in the FPGA. To achieve the optimum performance for these multipliers, three pipeline stages are used for each one, resulting in three clock cycles of latency. The results of the multiplications are added to form the weighted sum that represents the gray value of the corresponding incoming RGB pixel. This gray pixel is stored in an output FIFO as an 8-bit value. The total processing latency of the RGB to grayscale core is 9 clock cycles. Similar to the AWB IP core implementation, the RGB to grayscale core can process the three pixels inside the multipliers pipeline and store them in the output FIFO. This is achieved through the utilization of the "programmable full" signal in the output FIFO to control the processing flow if one of the FIFOs is full or if the video stream is halted by the processing pipeline. The input and output FIFOs have a small depth size of 16 Bytes, and they are implemented using the FPGA's distributed RAM. The grayscale converted image is shown Figure 4.34b.

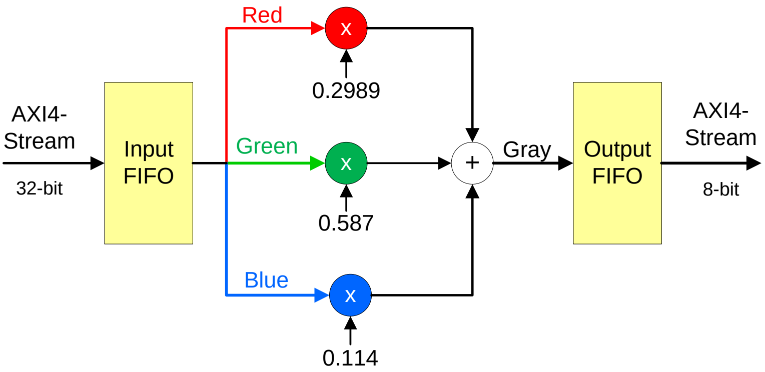


Figure 4.29: RGB to grayscale IP core block diagram

4.4.2 Background Estimation and Subtraction

Players are extracted as foreground using the background subtraction technique. It involves background estimation, update, and subtraction. In this work, the approximated median algorithm is selected for background estimation as depicted in Section 3.4. The FPGA implementation [111] [108] of this algorithm is shown in Figure 4.30. The estimated (and subsequently updated) background is computed using Equation 3.17, and it is stored in the external memory using the MPMC. The core uses a predefined number of input frames (BG_est_frame_nr) to estimate this background. In this application, this number is set to 300 (corresponding to 10 seconds at 30 fps). In order to correctly estimate the background during these 300 frames, either the players should be outside the court, or the players are inside the court, but they are moving. If a player stands motionless during these frames, he will be part of the background. To compensate for moving non-player objects, or players standing for a long time without movement, the background is continuously updated. Later, one frame is used to update the background every (BG_update_freq) frames. In this application, 15 was used. These two parameters are stored in the IP core's registers and can be easily modified by the user [108]. An example of the estimated background is shown in Figure 4.34c.

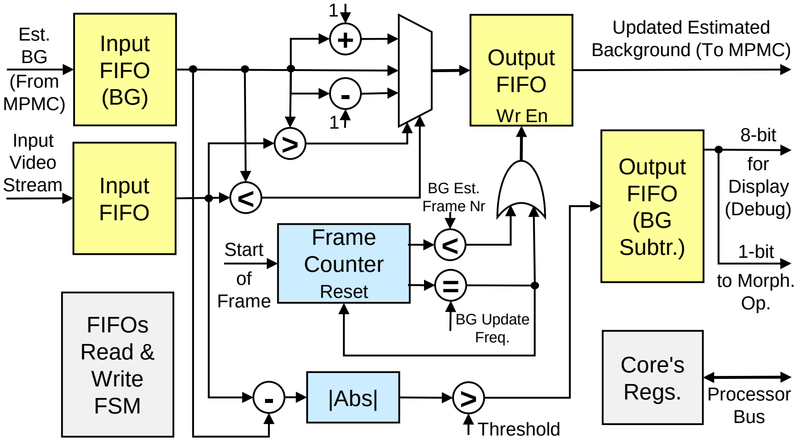


Figure 4.30: Background estimation and subtraction implementation [108] [107]

For every pixel in the input video stream, its value is subtracted from the corresponding pixel in the background estimated frame (that is read from the MPMC) as shown in Figure 4.30. Afterwards, the absolute value of the subtraction result is compared to a user-defined threshold. If this value is greater than the threshold, a binary 1 is written to the output FIFO. Otherwise, a binary 0 is used. Figure 4.34d shows the results of the background subtraction. As can be seen, the result is a binary image, by which the foreground objects (white) have a logic 1, and a logic 0 is used for the background (black). The core outputs this resulted image using two bit-widths as shown in Figure 4.30: The first one is 8 bits used for display (debugging). In this case, 255 (white) represents the foreground, and zero (black) is used for the background.

The width size of the second output is 1 bit, and it is connected to the next processing core (morphological operation IP core) as shown in Figure 4.28.

4.4.3 AXI4-Stream to NPI Controller

If an IP core needs to access the external memory through the MPMC, the AXI4-Stream to NPI controller is designed to convert between the AXI4-Stream interface (which is used by the video processing IP cores) and the NPI interface of the MPMC as shown in Figure 4.6. In this work, this controller is used to read and write the background estimated frames from and to the memory. Additionally, it is used to buffer the resulted video streams from the video processing IP cores to the external memory for display. This is used for debugging purpose, by which the intermediate results from the IP cores can be visualized at any processing step in the video pipeline.

The block diagram of the implemented AXI4-S to NPI controller is shown in Figure 4.31. The AXI4-S to NPI controller is implemented to convert the AXI4-Stream to NPI interface when writing video frames to the external memory (Double Data Rate (DDR)-SDRAM is used in this work). Additionally, it is used to convert the NPI to AXI4-Stream interface when reading video frames from the memory. The core is designed with generic parameters to select between either the writing (AXI4-S to NPI) or reading (NPI to AXI4-S) operation, and therefore the corresponding FSM (NPI write or NPI read FSM) is generated accordingly. Furthermore, a generic parameter is used to choose the desired data width size (8 or 32 bits) of the core's input and output interfaces. Based on that, the appropriate FIFO is generated. Using these generic parameters, the core's resources are generated according to the user-specific requirements for a resource-efficient IP core implementation, hence saving the FPGA's resources. The NPI data width is 64 bits, and the controller uses the maximum burst transfer size of 256 bytes to read and write the data from and to the external memory.

As shown in Figure 4.31, the input and output FIFOs utilize independent clock domains. The input FIFO uses the "FIFO CLK" (100 MHz is used in this implementation) for writing the incoming video data, while the "NPI CLK" (200 MHz) is used for reading the data from this FIFO, and writing them in the external memory using the NPI interface. For reading the video frames from the memory and storing them in the output FIFO, "NPI CLK" is used. Finally, the "FIFO CLK" is used by the output FIFO to read the stored data, and an FSM is implemented to convert this data to an AXI4-Stream output. The required handshaking is implemented in this core, ensuring the synchronization and data integrity between the two interfaces (the AXI4-Stream and the NPI interface).

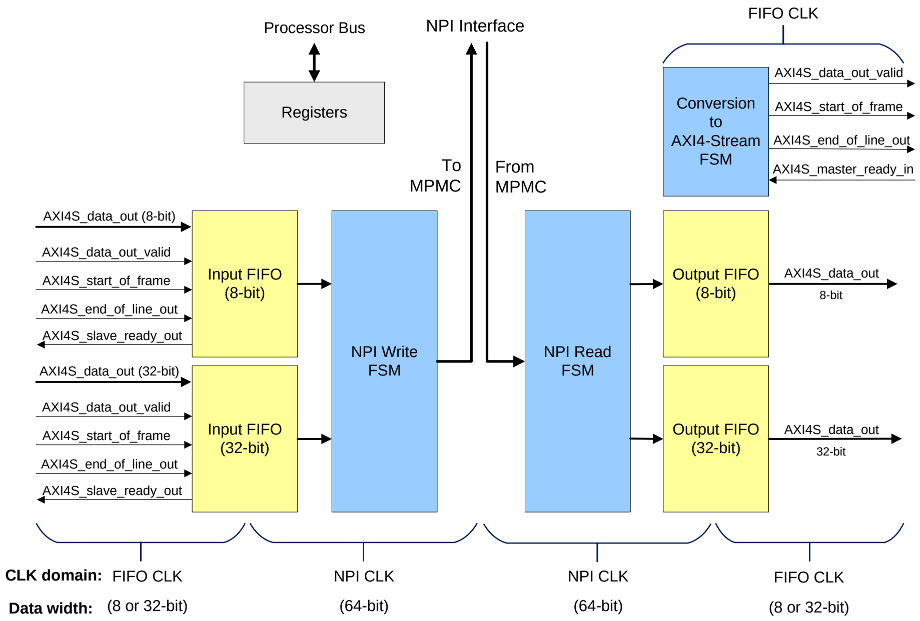


Figure 4.31: AXI4-S to NPI controller block diagram

4.4.4 Morphological Operations

Morphological dilation operation (as shown in Section 3.2) is applied after background subtraction to fill the gaps in the binary mask resulting from background subtraction. The IP core implementation is based on the design presented in [15] utilizing a 3x3 window as shown in Figure 4.32. The resulting frame after background subtraction with thresholding and dilation is shown in Figure 4.34e [108].

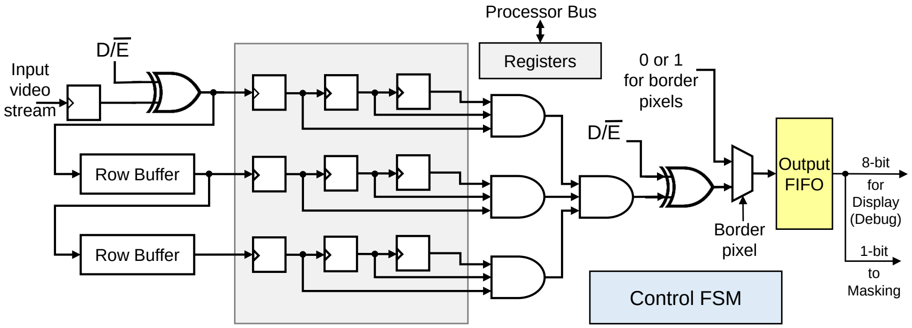


Figure 4.32: Block diagram of the implemented morphological dilation and erosion [15]

As shown in Figure 4.32, the design is used for the morphological dilation as well as erosion operation. The control signal ($D/\overline{E}$) selects between erosion and dilation [15]. The core's input video stream (from the background subtraction) has a 1-bit data width. Additionally, all the row buffers, registers, and combinational logic in Figure 4.32 have a 1-bit width, reducing the required computations and resources for this IP core. For pixels at the border of an input frame, the 3x3 window is not entirely within this frame, resulting in some missing pixels in this window. In this case, a predefined value (0 for black, or 1 for white) is used as the output result. The control FSM manages the row buffers, the border pixels, and storing the results in the output FIFO. The binary output of this core is sent to the masking IP core to obtain the colored foreground mask that includes the players as depicted in the next section.

4.4.5 Masking

As shown in Figure 4.28, the RGB video stream from the video preprocessing module output (Figure 4.34a) is masked with the binary foreground mask (Figure 4.34e) from the morphological dilation output. This is achieved using the Masking IP core to obtain the colored RGB foreground mask that includes the segmented players (Figure 4.34f) [108]. The implementation of the Masking IP core is shown in Figure 4.33.

Input FIFO 1 is used to buffer the incoming binary video stream from morphology, while input FIFO 2 is used to store the RGB frames from the preprocessing module. Additionally, the handshaking between the input AXI4-stream interfaces and these FIFOs is shown in Figure 4.33. The AXI4-Stream valid in signal is connected to the write enable (WR_EN) port, enabling the writing of the incoming data to the FIFO. The negating of the FIFO full signal is connected to the AXI4-Stream "slave ready out". When the FIFO is full, the "slave ready out" is driven low, indicating that the core is not ready to receive additional data. The input video stream is connected to the data input (Din) of the FIFO. Additionally, the AXI4-Stream start of frame (SOF) and end of line (EOL) can be buffered in the input FIFO. The two FIFOs are read if they are not empty and if the "master ready in" signal is active high. The masking operating is achieved using AND gates, by which every color component of each pixel is masked with the corresponding pixel in the foreground mask (from morphology). The resulted colored foreground mask (as shown in Figure 4.34f) is used by the next processing module, where the two teams are identified and the players are detected.

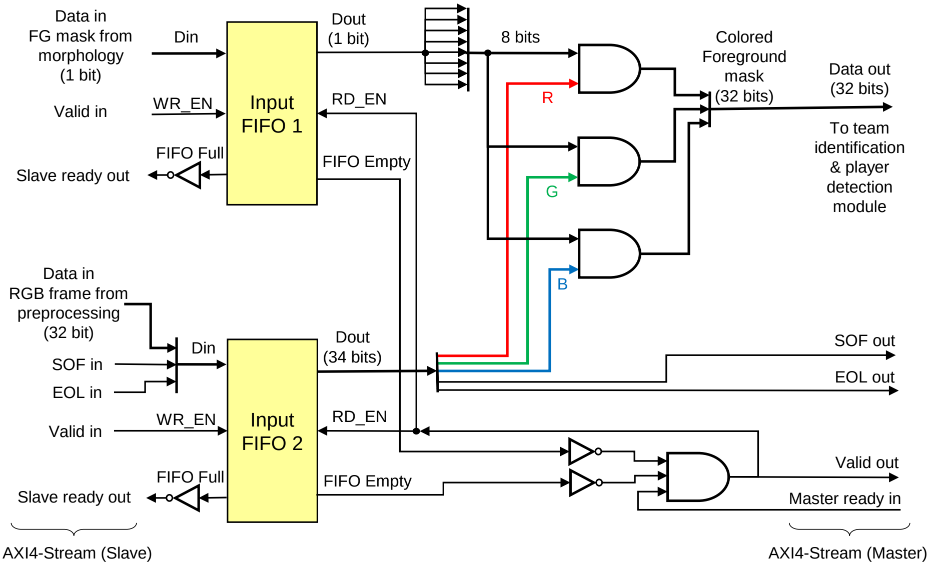
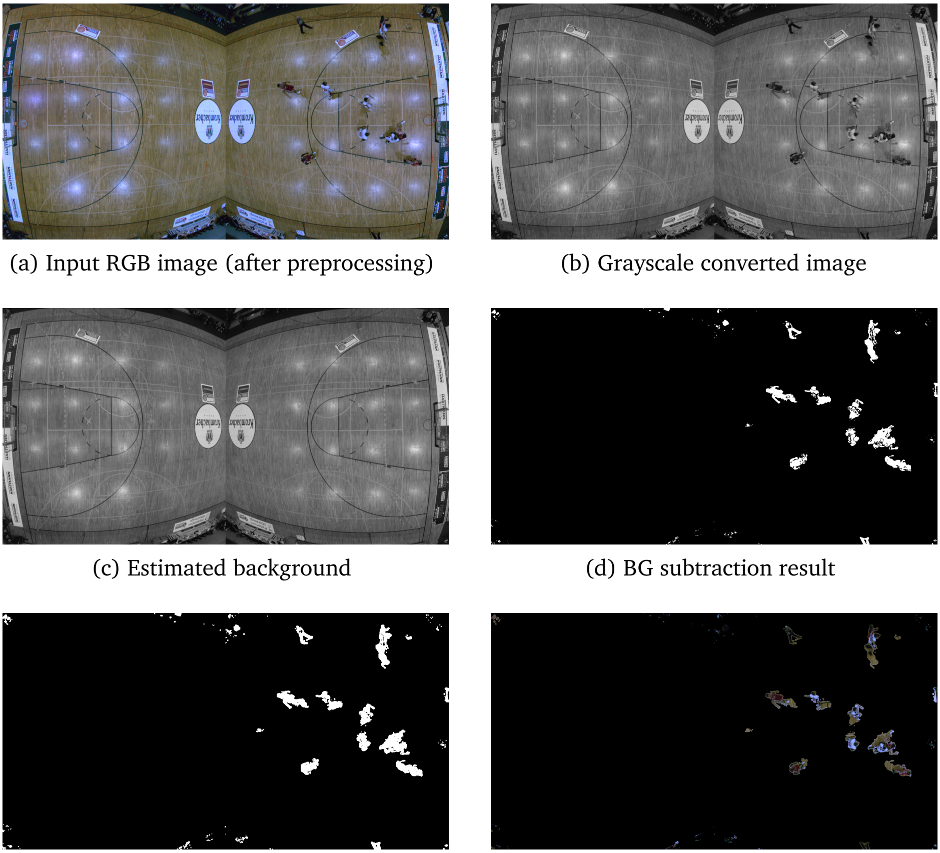


Figure 4.33: Implementation of the masking IP core



(a) Input RGB image (after preprocessing)

(b) Grayscale converted image

(c) Estimated background

(d) BG subtraction result

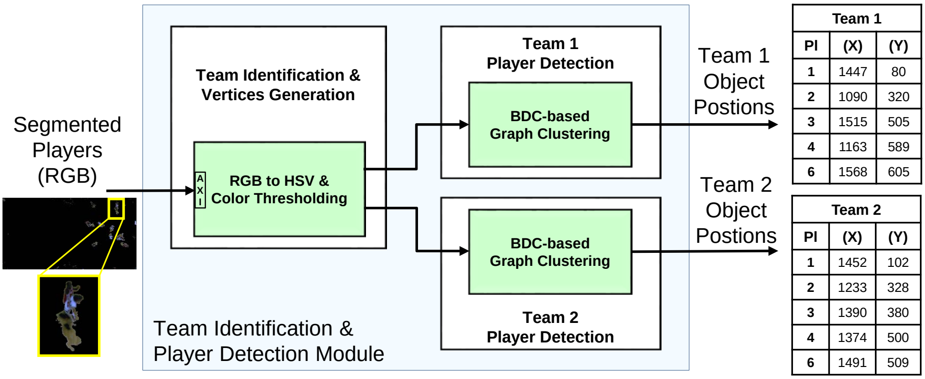
(e) BG subtraction after morphological dilation

(f) Masking

Figure 4.34: Resulting images from the IP cores in the player segmentation module [108] [107]

4.5 Team Identification & Player Detection Module

In this module, the colors of the players' jerseys are used to identify the two teams and to detect the positions of the players. This task is achieved using RGB to HSV color space conversion, color thresholding, and Binary Distance Calculation (BDC)-based graph clustering [108] as shown in Figure 4.35.



Team 1		
Pl	(X)	(Y)
1	1447	80
2	1090	320
3	1515	505
4	1163	589
6	1568	605

Team 2		
Pl	(X)	(Y)
1	1452	102
2	1233	328
3	1390	380
4	1374	500
6	1491	509

Figure 4.35: Team identification & player detection module

4.5.1 RGB to HSV Conversion & Color Thresholding

Previous work [6] [63] proves that the HSV color space is more robust than the RGB color space with respect to illumination and lighting changes. Therefore, the input RGB foreground mask video stream (which contains the segmented players) is converted to the HSV color space [108]. This is achieved using the RGB to HSV IP core. The core is based on the algorithm proposed by Foley et al. [36] as presented in Section 3.1.3. A block diagram of the implemented RGB to HSV conversion & color thresholding IP core [45] [15] is shown in Figure 4.36. In this implementation, a modification is applied to this algorithm to avoid the divisions in this algorithm [45], since divisions require a significant amount of FPGA resources and incur big latency. Therefore, Equation 3.12 is modified to avoid the division by Δ for the calculation of the Hue, resulting in Equation 4.3. Here, if the resulted value of $(\frac{H}{2} \times \Delta)$ from Equation 4.3 is a negative number, then $(180 \times \Delta)$ is added to the $(\frac{H}{2} \times \Delta)$ value. Furthermore, the division by $\max(RGB)$ in Equation 3.13 is avoided for the Saturation, resulting in Equation 4.4 [45].

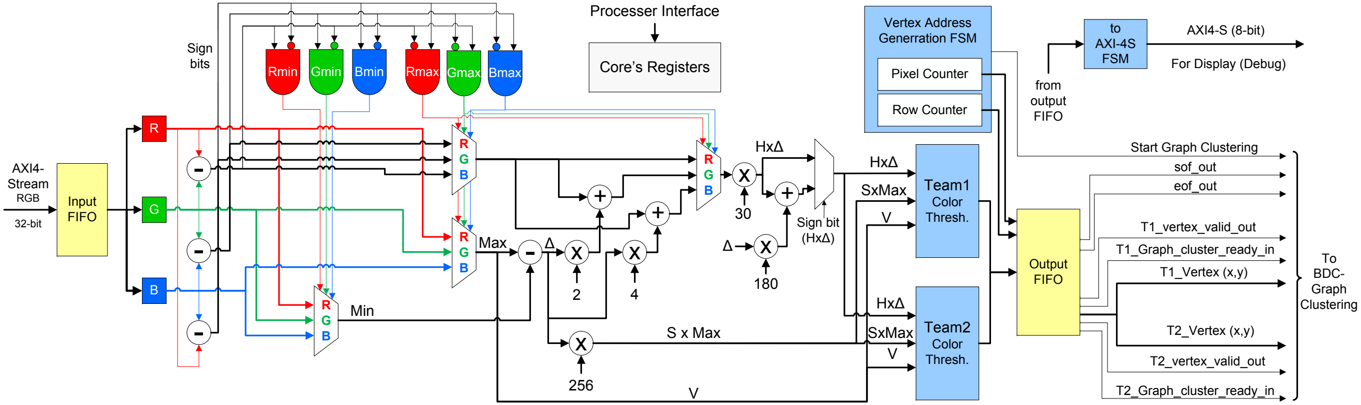


Figure 4.36: Block diagram of the implemented RGB to HSV conversion & color thresholding core based on the design presented in [45] and [15]

$$\frac{H}{2} \times \Delta = \begin{cases} 0, & \text{if } R = G = B \\ \\ 30 \times (G - B), & \text{if } \max(R,G,B) = R \\ \\ 30 \times ((2 \times \Delta) + (B - R)), & \text{if } \max(R,G,B) = G \\ \\ 30 \times ((4 \times \Delta) + (R - G)), & \text{if } \max(R,G,B) = B \end{cases} \tag{4.3}$$

$$S \times \max(RGB) = \begin{cases} \Delta, & \text{if } \max(R,G,B) \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.4}$$

where: Δ is equal to $\max(R,G,B) - \min(R,G,B)$.

As shown in Figure 4.36, the $\min(R,G,B)$ and $\max(R,G,B)$ values are determined using the sign bits of the difference between the input color channels. These differences are required to be calculated anyway for the numerators in Equation 4.3. Therefore, this information is efficiently obtained for free [15]. Furthermore, the resulted H and S are normalized by dividing H by 2, and multiplying S by 255 as depicted in Section 3.1.3. Therefore, after applying this normalization to Equation 4.3, the result is $H \times \Delta$ (instead of $\frac{H}{2} \times \Delta$), while Equation 4.4 is modified to Equation 4.5. In this equation, 256 is used (instead of 255) since multiplication by 256 is free in hardware.

$$S \times \max(RGB) = \begin{cases} \Delta \times 256, & \text{if } \max(R,G,B) \neq 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

The colors of the players' jerseys are used as the threshold values to mask this resulted HSV video stream. The color masking uses up to two colors from the jersey of each team [108]. The number of the used colors and the threshold values for both teams are stored in the core's registers. As an example, Equation 4.6 illustrates this thresholding operation, by which one jersey's color is used for team 1. In this example, Q_{T1} is the output binary mask for team1. It is equal to logic 1, if all the three conditions shown in Equation 4.6 are fulfilled. Otherwise, it is zero. Another example is shown in Equations 4.7, 4.8, and 4.9, by which two colors are used as threshold values for team 2.

$$Q_{T1} = \begin{cases} 1, & H_{T1\_Thr\_low} \leq H \leq H_{T1\_Thr\_high} \text{ AND} \\ & S_{T1\_Thr\_low} \leq S \leq S_{T1\_Thr\_high} \text{ AND} \\ & V_{T1\_Thr\_low} \leq V \leq V_{T1\_Thr\_high} \\ & \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

$$Q_{T2\_C1} = \begin{cases} 1, & H_{T2\_C1\_Thr\_low} \leq H \leq H_{T2\_C1\_Thr\_high} \text{ AND} \\ & S_{T2\_C1\_Thr\_low} \leq S \leq S_{T2\_C1\_Thr\_high} \text{ AND} \\ & V_{T2\_C1\_Thr\_low} \leq V \leq V_{T2\_C1\_Thr\_high} \\ & \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

$$Q_{T2\_C2} = \begin{cases} 1, & H_{T2\_C2\_Thr\_low} \leq H \leq H_{T2\_C2\_Thr\_high} \text{ AND} \\ & S_{T2\_C2\_Thr\_low} \leq S \leq S_{T2\_C2\_Thr\_high} \text{ AND} \\ & V_{T2\_C2\_Thr\_low} \leq V \leq V_{T2\_C2\_Thr\_high} \\ & \\ 0, & \text{otherwise} \end{cases} \tag{4.8}$$

$$Q_{T2} = (Q_{T2\_C1}) \text{ OR } (Q_{T2\_C2}) \tag{4.9}$$

However, since $H \times \Delta$ and $S \times \max(RGB)$ are calculated instead of H and S, the comparison in Equations 4.6, 4.7, and 4.8 must be modified accordingly. Therefore, Equation 4.6 is modified to Equation 4.10, by which the H threshold values are multiplied by Δ, and the S threshold values are multiplied by $\max(RGB)$ [45]. Similar modification is applied to Equations 4.7 and 4.8. The V value (which is equal to $\max(R, G, B)$) is left without modification. The block diagram of color thresholding for team 1 and 2 are shown in Figure 4.37 and Figure 4.38, respectively.

$$Q_{T1} = \begin{cases} 1, & H_{T1\_Thr\_low} \times \Delta \leq H \times \Delta \leq H_{T1\_Thr\_high} \times \Delta \text{ AND} \\ & S_{T1\_Thr\_low} \times \max(RGB) \leq S \times \max(RGB) \leq S_{T1\_Thr\_high} \times \max(RGB) \text{ AND} \\ & V_{T1\_Thr\_low} \leq V \leq V_{T1\_Thr\_high} \\ & \\ 0, & \text{otherwise} \end{cases} \tag{4.10}$$

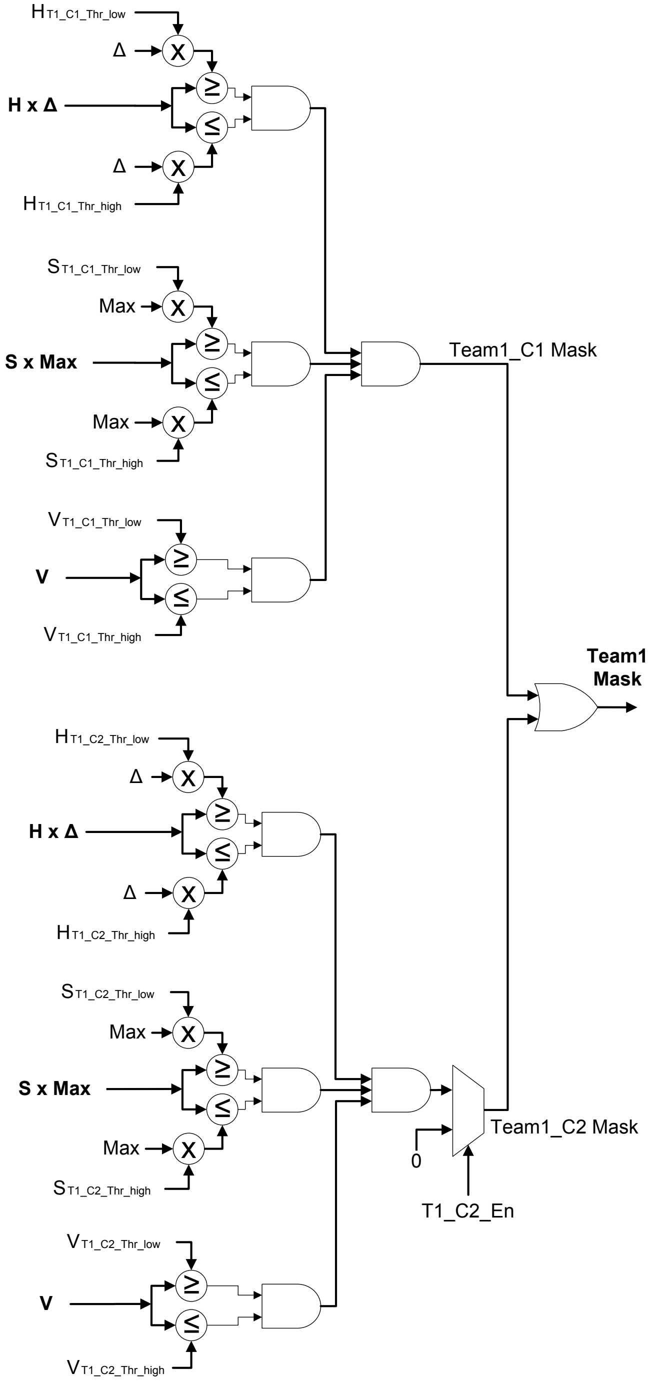


Figure 4.37: Team1 color thresholding

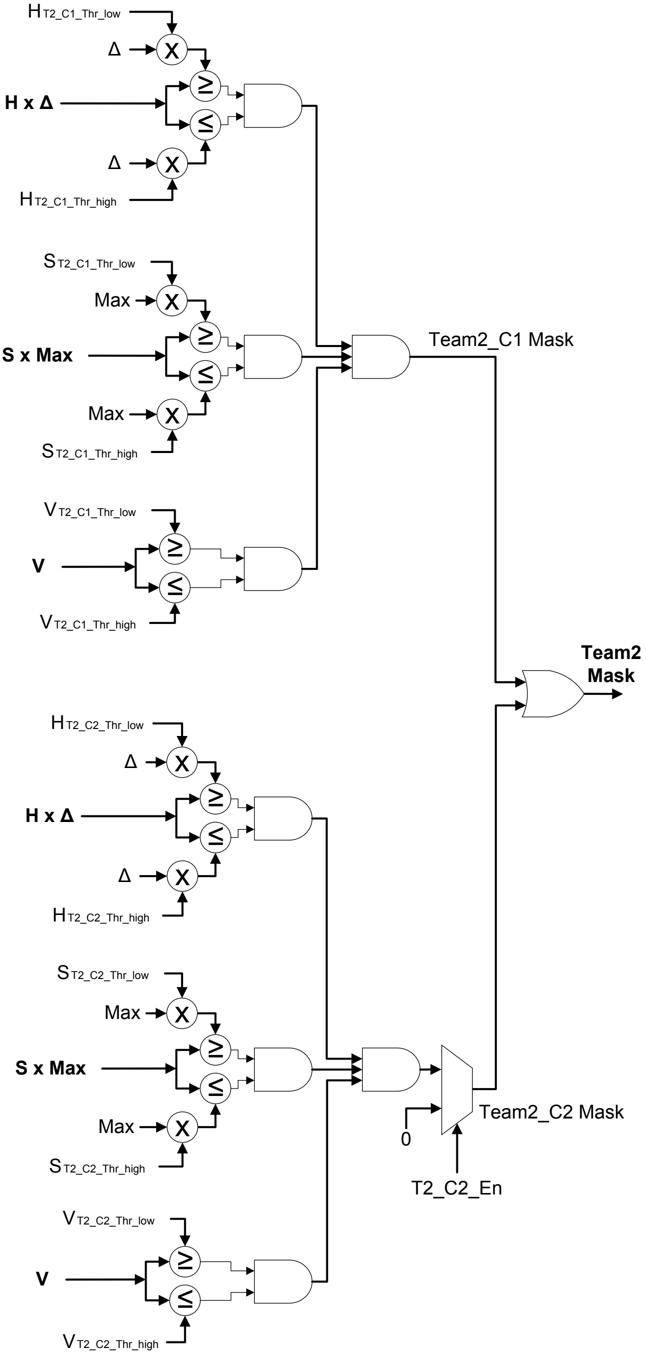
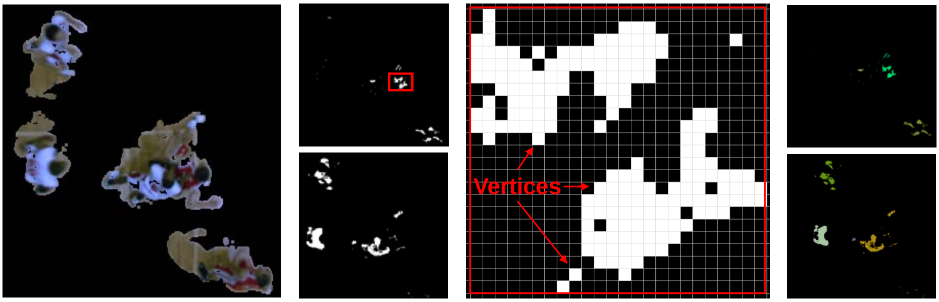


Figure 4.38: Team 2 color thresholding

As can be seen, the outputs of this core are two binary video streams, one for each team. These binary streams contain the vertices (pixels with a binary value of 1). These vertices belong to all the objects (including the players) that share the same color used in the mask [108]. Figure 4.39a shows an example using a basketball dataset, by which red is used for the color masking in team 1, while white is used in team 2. Figure 4.39b shows the results of color masking for the two teams, and a zoomed-in binary image showing the vertices (pixels with logic 1) is shown in Figure 4.39c. Another example from a handball game is shown in Figure 4.40a. Figure 4.40b shows the resulted vertices for team 1 after color masking, while the results for team 2 is shown in Figure 4.40c. Finally, these vertices are used by graph clustering to detect the positions of the players as shown in the next section. The advantage of this approach is that the players from different teams are separated, i.e., each of the two output binary video streams contains only the vertices of the players that belong to the same team. This enhances the detection rate in scenarios where two players from opposing teams are very close to each other (e.g., occluded) [108].

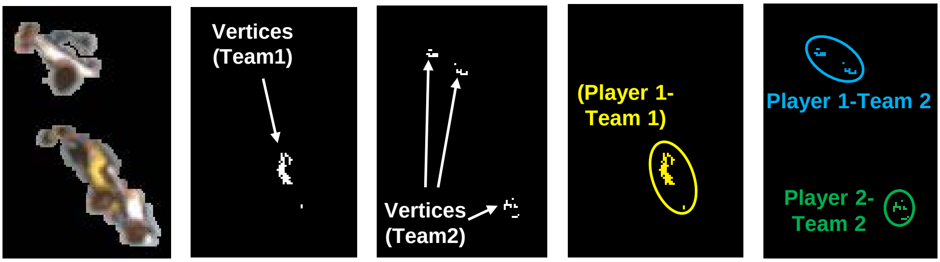


(a) A zoomed-in foreground mask from Figure 4.34f (b) Color thresholding (team 1&2) (c) A zoomed-in binary image from Figure 4.39b (top) showing the vertices (d) BDC-graph clustering (team 1&2)

Figure 4.39: Results from the RGB to HSV conversion & color thresholding and BDC-based graph clustering IP cores using a basketball dataset [108] [107]

4.5.2 BDC-based Graph Clustering

As stated in Section 3.5, graph clustering is grouping the vertices of the graph into clusters considering the edge structure of that graph [80]. In this work, vertices are the pixels with a binary value of 1 that result after color thresholding for each team as shown in Figures 4.39 and 4.40. The edges are the binary distances that are calculated using Equation 4.11 [108].



(a) A foreground mask (handball game) (b) Color thresh. (using yellow color, team 1) (c) Color thresh. (using white color, team 2) (d) Visualized clustering results (team 1) (e) Visualized clustering results (team 2)

Figure 4.40: Results from the RGB to HSV conversion & color thresholding and BDC-based graph clustering IP cores using a handball dataset

$$\mathrm{BDC} = \begin{cases} 1 & \text{if } |X_2 - X_1| < d_{th} \; and \; |Y_2 - Y_1| < d_{th} \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

where: d_{th} is the threshold value for maximum distance.

Due to its low resource requirements and adequate performance, the Chebyshev method is used to implement the BDC calculation. It does not involve multiplication and requires only subtraction and logical *AND* operations as shown in Equation 4.11 [107]. The BDC-based graph clustering IP core has been developed for FPGA-based multi-robot tracking by the Cognitronics and Sensor Systems research group at Bielefeld university [109] [110]. It is modified in order to be used for clustering the vertices in the player tracking application [108]. Two instances of the BDC-based graph cluster IP core are used in the proposed system, one for each team as shown in Figure 4.35. The block diagram of the implemented BDC-based graph cluster IP core is shown in Figure 4.41, while the flowchart illustrating the core's operation is shown in Figure 4.42.

As can be seen in Figure 4.41 and 4.42, the coordinates of the incoming vertices are buffered in an input FIFO. When the first vertex is received, a new cluster is created. For the next vertex, the binary distance is calculated between the coordinates of that vertex and the created cluster. If the BDC equals to 1, this vertex is considered to belong to this cluster, and the centroid of that cluster is updated accordingly. Otherwise, if the BDC value is 0, a new cluster is created. The process is repeated for all the subsequent vertices [108]. This means, when a new vertex is read from the input FIFO, the distance between the centroid of this vertex and the center of the first existing cluster is calculated using the BDC unit. If this distance is 0, the binary distance is

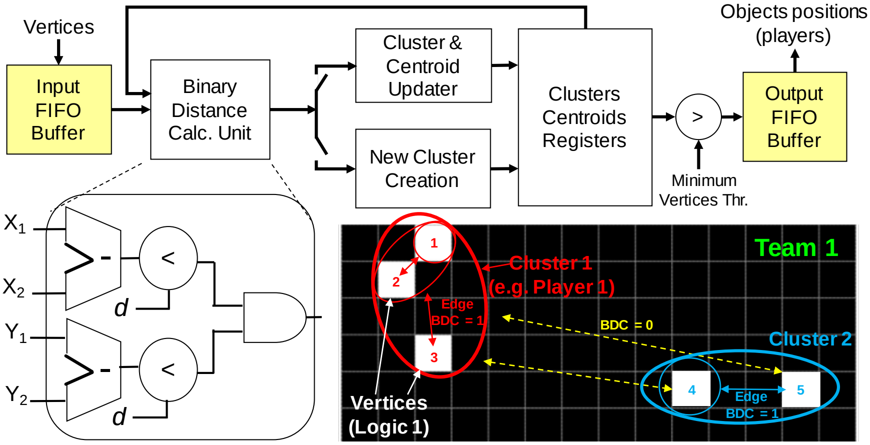


Figure 4.41: BDC-based graph clustering and IP core block diagram [110] [108] [107]

calculated again between this vertex and the next existing cluster. This process is repeated until the end of the last existing cluster if the resulting calculated distance values are 0. In this case, a new cluster is created using the coordinates of that vertex. On the other hand, if a distance value of 1 is found between the vertex centroid and one of the existing clusters, the centroid of this cluster is updated using this vertex coordinates. Subsequently, a new vertex is read from the input FIFO, and the clustering process is repeated until all vertices are processed and the end of the frame is reached as shown in Figure 4.42 [107]. The vertices that belong to the same clusters are shown in Figure 4.41 after clustering is performed. For the basketball example shown in Figure 4.39, the clustering results are shown in Figure 4.39d. While for the handball example (shown in Figure 4.40), the clustering results are shown in Figures 4.40d and 4.40e for team 1 and team 2, respectively. In these examples, different colors are used for visualization, showing the resulted clusters. Finally, the centroid of each cluster is the average of the coordinates of all the vertices that belong to the same cluster as calculated using Equation 4.12 [107]. An example of the calculated centroids of the players for both teams is shown in Figure 4.35.

$$\overline{x}_i = \frac{1}{K}\sum_{j=1}^{K} x_j,\ and\ \overline{y}_i = \frac{1}{K}\sum_{j=1}^{K} y_j \qquad (4.12)$$

where:
$\overline{x}_i$ and $\overline{y}_i$ are the centroid's coordinates of cluster i.
K is the number of vertices in a cluster.
x_j and y_j are the coordinates of vertex j.

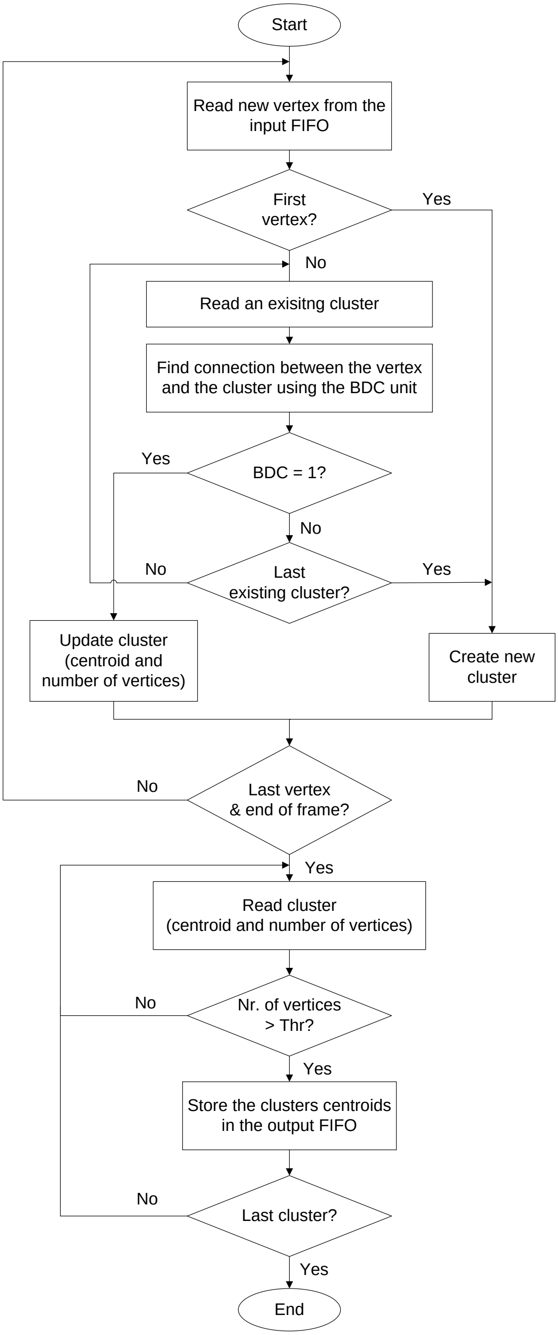

Start
Read new vertex from the input FIFO
First vertex?
Yes
No
Read an exisitng cluster
Find connection between the vertex and the cluster using the BDC unit
Yes
BDC = 1?
No
No
Last existing cluster?
Yes
Update cluster (centroid and number of vertices)
Create new cluster
No
Last vertex & end of frame?
Yes
Read cluster (centroid and number of vertices)
No
Nr. of vertices > Thr?
Yes
Store the clusters centroids in the output FIFO
No
Last cluster?
Yes
End

Figure 4.42: BDC-based graph clustering flowchart [110] [107]

In order to reduce the false positives (non-player detections), only the centroids of the clusters with a number of vertices higher than a threshold value are considered and written to the output FIFO buffer as depicted in Figures 4.41 and 4.42. These centroids represent the positions of the detected objects including the players. Therefore, the output of the first and second BDC-graph cluster IP cores include the players' positions of team 1 and team 2, respectively, and are shown in green and yellow squares in Figure 4.43. Here, fixed height and width values are used for the squares (30x30) since the player size does not change significantly in different frame locations. The centers of these squares are the centroids from the two BDC-graph cluster IP cores. Finally, these centroids are transferred to the host-PC for further processing, including player tracking as shown in Section 4.7 [107].

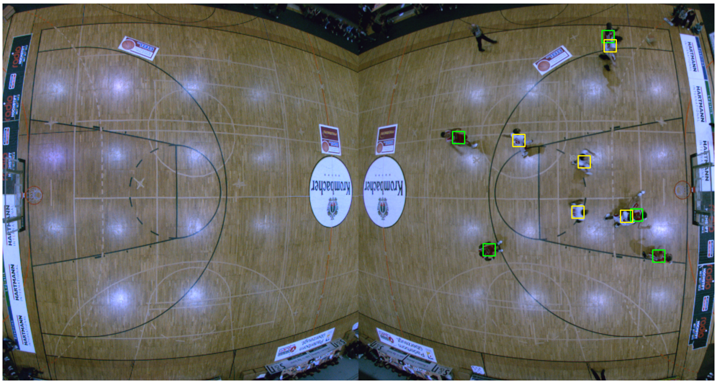


Figure 4.43: Detection results for both teams (green & yellow squares) [108] [107]

4.6 Resource Utilization

In this section, the FPGA resources that are required by the different IP cores for the processing modules are presented. These resources include the amount of Flip Flops (FFs), Look-Up-Tables (LUTs), Block RAMs (BRAMs), and Digital Signal Processors (DSPs). The FPGA architecture is implemented using a Xilinx Virtex-4 (XC4VFX100) FPGA. The resources used by the MC_GigEV IP core are shown in Table 4.1 for one to four camera configurations. As can be seen, the required resources are not doubled if an additional camera is supported. As an example, the percentage of the required FFs, LUTs, and BRAMs using the Virtex-4 FPGA for one camera configuration are 1.6%, 1.9%, and 3.5%, respectively. If the two camera configuration of the MC_GigEV core is used, only 1.8% of the FFs, 2.6% of the LUTs, and 4.5% of the BRAMs are required.

Table 4.1: Device utilization (Virtex-4 FX100-11) for the MC_GigEV IP core with configurations for 1, 2, 3 and 4 cameras [106]

MC_GigEV IP Core	FFs	LUTs	BRAM16s	DSP48s
Virtex-4 (FX100-11)	84,352	84,352	376	160
1 Camera	1317 1.6%	1618 1.9%	13 3.5%	0 0%
2 Cameras	1553 1.8%	2215 2.6%	17 4.5%	0 0%
3 Cameras	1786 2.1%	2547 3%	21 5.6%	0 0%
4 Cameras	2031 2.4%	3049 3.7%	25 6.6%	0 0%

The resources and the maximum clock frequencies (Fmax) of the IP cores in the processing modules of the FPGA architecture are shown in Table 4.2. Here, an operating clock frequency of 100 MHz for the vision processing IP cores is used. As can be seen, the video acquisition module requires less than 10% of the available FFs and LUTs of the Virtex-4 FPGA, while 13.6% of BRAMs is used. These BRAMs are required for the FIFO buffers in the MC_GigEV, Video File Controller (used as row buffers), and Xilinx TEMAC IP cores. For the video preprocessing module, most resources are utilized by the AWB core, since it involves two divisions (cf. Figure 4.22) which require a big amount of FPGA resources. The player segmentation module requires less than 5% of the total resources, while the team identification & player detection module uses a relatively large amount of resources as shown in Table 4.2.

In the team identification & player detection module, there are two instances of the BDC-based Graph Clustering IP core, each uses two divisions (cf. Equation 4.12), requiring a large amount of the FPGA resources. Additionally, logic resources are used by the clustering operation (cf. Figure 4.42) and the registers for clusters' centroids (cf. Figure 4.41). These registers are used to store intermediate values during the clustering process. The number of these registers depends on the maximum supported clusters (objects) that can be detected in one frame. In this implementation, this number is set to 128 to include the players, false positives, and objects correspond to noise detections (i.e., clusters that have vertices less than or equal to the predefined "minimum vertices threshold" value as shown in Figure 4.41). The base system consists of the MPMC, (LB-Slave to NPI/AXI4-S, display, AXI4-S to NPI (4x)) controllers, PPC system and clock management [108]. The complete FPGA architecture requires around 60% of the Xilinx Virtex-4 FPGA. The achieved maximum clock frequencies (Fmax) of the IP cores in

Table 4.2: Device Utilization (Virtex-4 FX100-11)

	FFs	LUTs	BRAMs	DSP	Fmax (MHz)
Virtex-4 (FX100-11)	84352	84352	376 (18Kb)	160	
MC_GigEV (2xCameras)	1553	2215	17	0	150
GigE Camera Config	289	292	0	0	400
Video File Controller	889	703	12	0	240
TEMAC	3188	4161	22	0	140
AXI4-S Mux (x2)	278	152	0	0	N/A
Video Acquisition Module	6197 7.3%	7523 8.9%	51 13.6%	0 0%	140
Demosaicing (x2)	1212	3922	2	0	140
AWB (x2)	5132	5544	4	52	190
Cropping (x2)	704	824	0	0	250
Video Merge	474	458	8	0	210
AXI4-S Mux	139	76	0	0	N/A
Video Preprocessing Module	7661 9.1%	10824 12.8%	14 3.7%	52 32.5%	140
RGB to Gray	161	156	0	3	200
Background Subtraction	639	684	4	0	170
Morphological Operation	211	334	3	0	220
Masking	178	187	9	0	220
Player Segmentation Module	1189 1.4%	1361 1.6%	16 4.3%	3 1.9%	170
RGB to HSV Conv. & Color Thr.	2012	2088	5	18	260
BDC-based Graph Cluster (x2)	16842	12402	26	28	190
Team Identification & Player Detection Module	18854 22.4%	14490 17.2%	31 8.2%	46 28.8%	190
Base System	15096 17.9%	15639 18.5%	98 26.1%	0 0%	N/A
Total	48997 58.1%	49837 59.1%	210 55.9%	101 63.1%	N/A

each module are reported as shown in Table 4.2. Among these values, the lowest Fmax is used as the maximum clock frequency of the module.

In addition to the implementation on the Xilinx Virtex-4 FPGA, the system has been realized on a Xilinx Virtex-7 (VX690T-2), showing the impact of utilizing a more recent architecture and proving the portability of the developed IP cores. In this case, the PLB bus interface (that is used in various implemented IP cores to configure their's registers) is replaced with the AXI4-Lite interface that is supported by the Virtex-7 FPGA [107]. The required resources and the maximum frequencies are reported in Table 4.3. In general, the number of used FFs in the IP cores slightly differs as compared with the required FFs using the Virtex-4 FPGA. However, the utilized LUTs using the Virtex-7 FPGA are reduced compared with the Virtex-4 FPGA implementation since the LUT size in Virtex-7 is 6, while Virtex-4 architecture has a LUT size of 4 (cf. Table 2.2). In Virtex-7 FPGA, the BRAMs size is 36 Kb, and each block can also be used as two independent 18 Kb BRAMs [93]. The used DSPs in the IP cores are equal in both FPGAs. As compared with the Virtex-4 FPGA implementation, the achieved Fmax values of the IP cores are higher using the more recent Virtex-7 FPGA.

4.7 Player Tracking

As shown in Figure 4.2, the compute-intensive operations for the pixel processing to detect the player positions in every frame are handled by the FPGA while the less compute-intensive tracking is done on the host-PC. The host-PC receives the positions of the detected objects from the FPGA for further processing. These detections include true positives (players) and false positives (non-players) for both teams. In the following section, player tracking on the host-PC is explained in detail. Player transfer between the two cameras is explained in Section 4.7.3 [107].

4.7.1 Single Camera Player Tracking

In this work, player tracking is achieved on the host-PC using the tracking-by-detection approach as shown Figure 4.44. Here, the received detections from the FPGA are represented by blue and red circles, corresponding to the positions of players from team 1 and 2, respectively. In Figure 4.44, three players from team 1 and two players from team 2 are moving from the left to the right direction in five frames. To achieve player tracking, association of the detections with the players in these frames is required. In the first frame, the received detections are used to create tracks. Player tracking is achieved by associating the subsequent detections to these tracks as shown in Figure 4.44. The Munkres' version of the Hungarian algorithm [68] is used to solve this data association problem, assigning one detection to one track. First, the Euclidean

Table 4.3: Device Utilization (Virtex-7 VX690T-2)

	FFs	LUTs	BRAMs	DSP	Fmax (MHz)
Virtex-7 (VX690T-2)	866400	433200	1470/2940 36Kb/Kb18	3600	
MC_GigEV (2xCameras)	1444	1525	8/1	0	200
GigE Camera Config	234	183	0/0	0	600
Video File Controller	845	666	6/0	0	280
TEMAC	1700	1400	0/0	0	280
AXI4-S Mux (x2)	182	202	0/0	0	N/A
Video Acquisition Module	4405	3976	14/1	0	200
Demosaicing (x2)	1024	2126	0/2	0	360
AWB (x2)	5236	3554	0/4	52	370
Cropping (x2)	566	416	0/0	0	400
Video Merge	405	376	4/0	0	330
AXI4-S Mux	91	101	0/0	0	N/A
Video Preprocessing Module	7322	6573	4/6	52	330
RGB to Gray	154	104	0/0	3	330
Background Subtraction	584	669	0/4	0	220
Morphological Operation	195	211	0/3	0	240
Masking	169	142	4/1	0	230
Player Segmentation Module	1102	1126	4/8	3	220
RGB to HSV & Color Thr.	1881	1483	2/1	18	360
Graph Clustering (x2)	16426	8590	10/6	28	230
Team Identification & Player Detection Module	18307	10073	12/7	46	230
Total	31136	21748	34/22	101	N/A

distance is calculated between every detection (D) and the current position of each track (T). These distance values are considered as the cost of matching a detection to track, and they are used to build the cost matrix. After that, the Hungarian algorithm is applied to this cost matrix, assigning a detection to each track using minimum cost. If the number of tracks is larger than the detections, the prediction values from Kalman filters are assigned to the tracks that did not have detections assigned to them as shown in Figure 4.44 (frame 3, team 1). If the number of detections is larger than the tracks, the unassigned detections are used to create new tracks as depicted in Figure 4.44 (frame 3, team 2). This process of data association is applied to all the subsequent frames [107]. More details about these data assignments are shown in the next section.

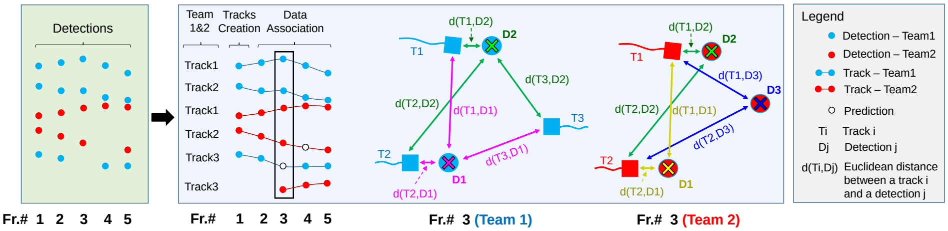
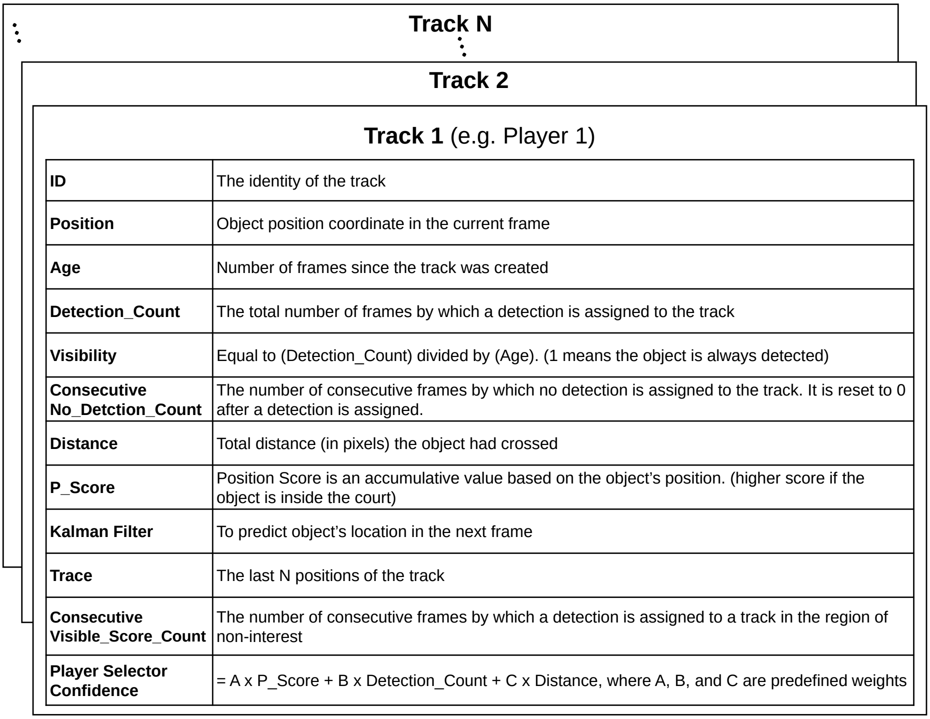


Figure 4.44: Overview of player tracking and data association [107]

Each track consists of several parameters (e.g., *ID*, *Position*, next position (estimated by *Kalman filter*), covered *distance*,...) as shown in Figure 4.45. These parameters contain different information to manage and monitor the track's status. After the tracks are created, and data association is applied, the parameters of the tracks are updated accordingly. As shown in Figure 4.45, the track's *ID* is assigned to each track sequentially according to the created order of the tracks. *Position* stores the current position of the object in a frame. The *age* of the track is the number of frames since it was first created. *Detection_count* is the total number of frames where an object is detected, and this detection is assigned to the track. *Visibility* is the ratio between the *Detection_Count* of the track and its *age*. A visibility of 1 means that the object that belongs to the track is detected in all the frames since the track was created. A lower visibility value implies that the object was not detected in some frames. *Consecutive_no_detection_count* stores the number of frames where the object is not detected in consecutive frames. When the object is detected, the *consecutive_no_detection_count* is reset to zero. The total covered distance (in pixels) that the object had crossed is stored in the *distance* parameter [107].

False positives can be introduced, e.g., through the substitute players (cf. Figure 2.4) who wear the same jerseys as the active players and are located close to the court. If these players move slightly, they will not be considered as part of the background after background subtraction, but they will belong to the foreground objects. In this case,



Track N
Track 2
Track 1 (e.g. Player 1)

ID	The identity of the track
Position	Object position coordinate in the current frame
Age	Number of frames since the track was created
Detection_Count	The total number of frames by which a detection is assigned to the track
Visibility	Equal to (Detection_Count) divided by (Age). (1 means the object is always detected)
Consecutive No_Detction_Count	The number of consecutive frames by which no detection is assigned to the track. It is reset to 0 after a detection is assigned.
Distance	Total distance (in pixels) the object had crossed
P_Score	Position Score is an accumulative value based on the object's position. (higher score if the object is inside the court)
Kalman Filter	To predict object's location in the next frame
Trace	The last N positions of the track
Consecutive Visible_Score_Count	The number of consecutive frames by which a detection is assigned to a track in the region of non-interest
Player Selector Confidence	= A x P_Score + B x Detection_Count + C x Distance, where A, B, and C are predefined weights

Figure 4.45: Parameters of a track [107]

they can be detected as players (i.e., false positives). Additional false positives may arise, e.g., from the digital advertising panels when new content appears, making it part of the foreground. If these contents have the same color as the player's jerseys, these panels can lead to false detection of one or more players. To reduce the effects of these false positives, a predefined Region of Non-Interest (RONI) is used which includes most of the substitute players area and the advertisement panels outside the sports court. This region is user-defined, and it consists of two sub-regions; the hard and soft decision RONI as shown in Figure 4.46. In the hard decision region, there is no overlap between the active players and this region during the game. Therefore, all the detections (which are FPs) in this region are discarded. The soft decision region is slightly larger than the hard region as shown in Figure 4.46, and an active player could be detected in this region. Therefore, a player track should be distinguished from a false track. This is achieved using the *P_score* (position score) parameter. This score is an accumulative value based on the positions of the detected objects in the court. It is incremented by an "P" value if the detected object's position is outside the soft decision RONI region. Otherwise, the *P_score* stays the same. In the proposed system, "P" is set to 1 (a higher value could also be used). As a result, the tracks that correspond to false alarms (non-players) will have a low *P_score* value, whereas the players' tracks will have a high value [107].

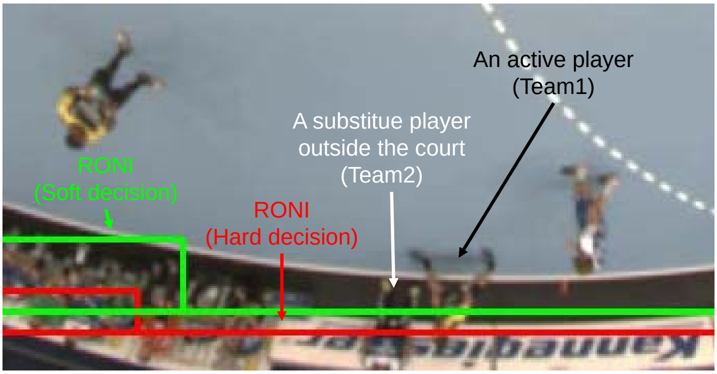


Figure 4.46: Soft and hard decision region of non-interest (RONI) [107]

A *Kalman filter* is used to predict the next position for each track. This prediction is used if no detection is assigned to an existed track as explained earlier. If an associated detection is found, the Kalman filter is updated with that detection. The last *N* positions of a player are stored in the *trace*. These positions are used for display, visualizing player tracking. In this system, *N* is set to 20 (a different number could be used to display a less or higher number of player last positions). *Consecutive_visible_score_count* is the number of consecutive frames when the track has detections in the soft decision

RONI region. Otherwise, this parameter is reset to zero. The tracks that correspond to false alarms or the tracks that did not have assigned detections for a long time could be deleted based on their parameters as shown in Algorithm 4.1 [107].

Algorithm 4.1 Track deletion [107]

```
for (all the tracks in each team) do
    if ((track's age < age_thr) AND (visibility < visibility_thr)) OR
    (consecutive_no_detection_count>consecutive_no_detection_count_max_thr) OR
    (consecutive_visible_score_count>consecutive_visible_score_count_max_thr) then
        Delete this track
```

For a particular sport, the number of players per team (P) is fixed (five players per team for basketball, and six players per team for handball excluding the goalkeeper). Since the number of tracks can be bigger than the number of players, a *player selector confidence* is used to select P tracks (i.e., five tracks for basketball and six tracks for handball) from the existing tracks. These selected tracks have the highest confidence values, and they are considered as the players of one team. The *player selector confidence* is equal to a weighted sum of the *P_score*, *detection_count*, and the *distance* as shown in Figure 4.45. In this system, the used weights of the player selector confidence A, B, and C (based on empirical tests) are 0.5, 0.25, and 0.25, respectively. This *player selector confidence* is also used to handle the player substitutions, where they are unlimited in basketball and handball games [107]. As a player leaves the court, he usually enters the soft decision RONI, where his *P_score* value does not increase further. After that, he enters the hard decision RONI, where his detections in that area is discarded, causing the *conesecutive_no_dectection_count* and *conesecutive_visible_score_count* to increase. As a result, the track that corresponds to this player is deleted using Algorithm 4.1. On the other hand, when a substitute player enters the court, a new track is created, and his *player selector confidence* value starts to increase during the subsequent frames. As a result, the track corresponding to this player is selected among the P tracks that represent the team.

The flowchart of the overall player tracking steps is shown in Figure 4.47. These steps are applied for the detections and tracks in every frame. The number of tracks in each frame is equal to the players' tracks and false positives tracks. New tracks are created as required for the unassigned detections in each frame. Figure 4.48 shows the final player tracking results. In this figure, the color used for the trajectories and the corresponding bounding boxes indicate the players that belong to the same team. Additionally, the tracks' ID and the current position are shown for each player [107].

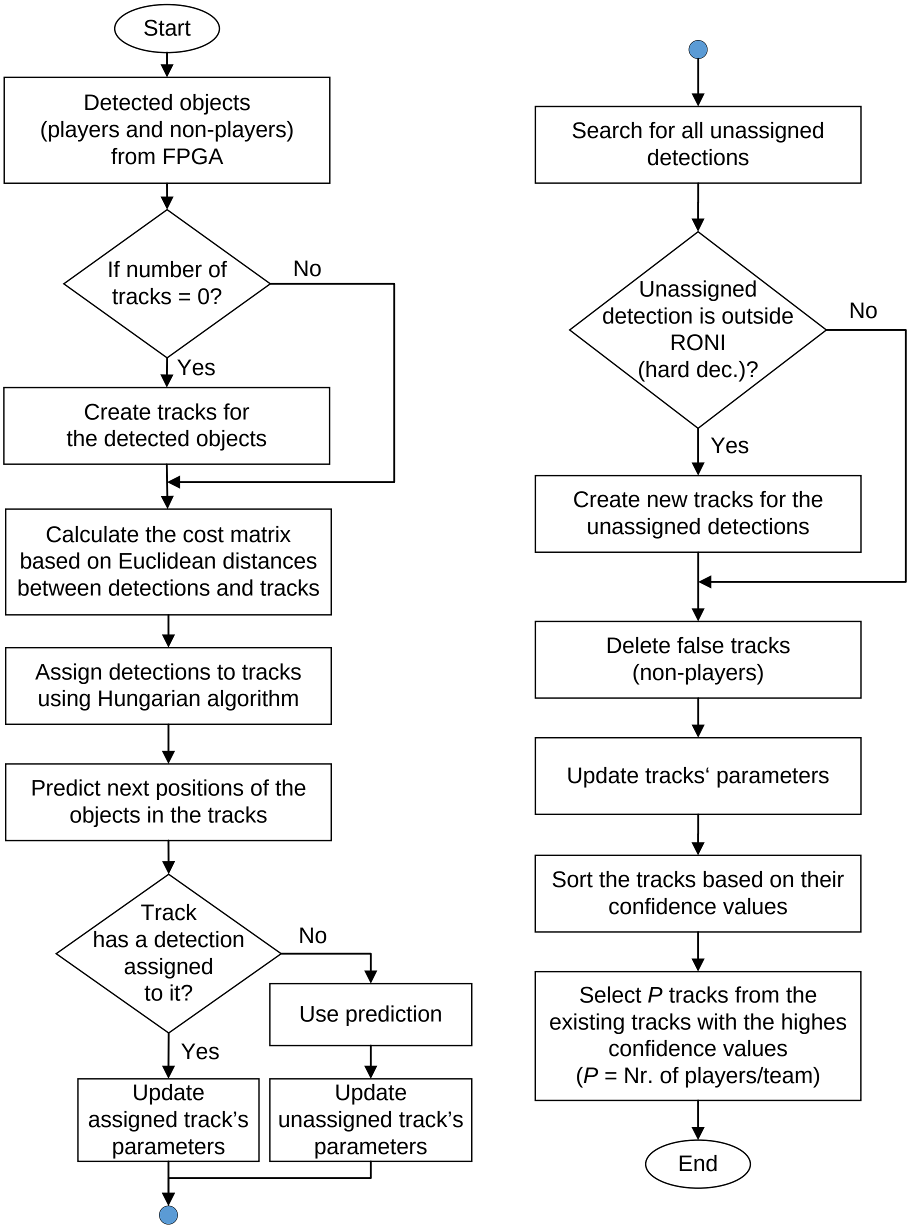

Start
Detected objects (players and non-players) from FPGA
If number of tracks = 0?
No
Yes
Create tracks for the detected objects
Calculate the cost matrix based on Euclidean distances between detections and tracks
Assign detections to tracks using Hungarian algorithm
Predict next positions of the objects in the tracks
Track has a detection assigned to it?
No
Use prediction
Yes
Update assigned track's parameters
Update unassigned track's parameters
Search for all unassigned detections
Unassigned detection is outside RONI (hard dec.)?
No
Yes
Create new tracks for the unassigned detections
Delete false tracks (non-players)
Update tracks' parameters
Sort the tracks based on their confidence values
Select *P* tracks from the existing tracks with the highes confidence values (*P* = Nr. of players/team)
End

Figure 4.47: Player tracking flowchart

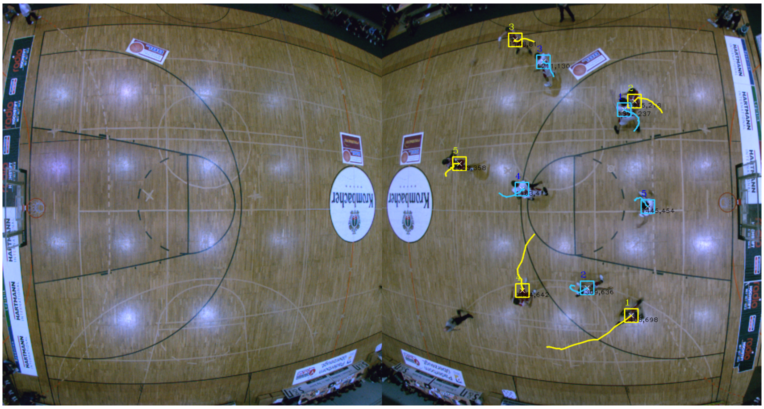


Figure 4.48: Visualization of player tracking results [107]

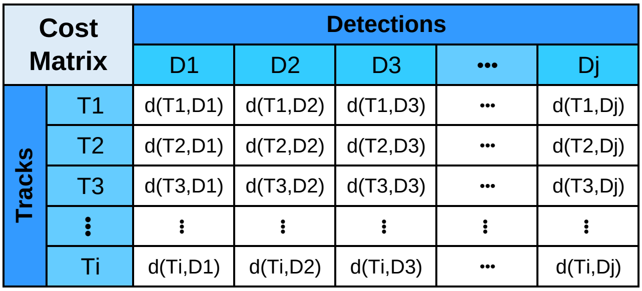
4.7.2 Detections Association to Tracks

In every frame, detections are assigned to tracks using the Munkres version of the Hungarian algorithm [68] as stated in the previous section. The cost matrix is calculated for every frame based on the Euclidean distance between the current position of each existing track (T) and each detection (D). Equation 4.13 shows the Euclidean distance $d(Ti, Dj)$ between the current position of a track i and a detection j. The cost matrix is shown in Figure 4.49. After the cost matrix is built, the Hungarian algorithm is applied to it, resulting in assigning one detection to each track using minimum cost.

$$d(Ti, Dj) = \sqrt{(Ti_x - Dj_x)^2 - (Ti_y - Dj_y)^2} \tag{4.13}$$

where:
d is the Euclidean distance between a track Ti and a detection Dj.
$Dj_{x,y}$ is the position of detection j.
$Ti_{x,y}$ is the current position of track i.

Figure 4.50a shows an example, by which there are three detections (D1, D2, and D3) and three tracks (T1, T2, and T3). T1 corresponds to an object who is moving from the right to the left direction, while T2 and T3 are objects moving from the left to the right. The cost matrix is calculated as depicted in Figure 4.50b. The Hungarian algorithm is applied to this cost matrix to find the minimum cost of assigning each detection to each track. The minimum cost is the lowest total distance of assigning all the detections to the tracks. After applying the Hungarian algorithm to this example, detection D1 is assigned to track T3, D2 is assigned to T1, and D3 is assigned to T2 as shown in Figure 4.50c. In this case, the total minimum cost of all assignments (shown



Cost Matrix		Detections				
		D1	D2	D3	•••	Dj
Tracks	T1	d(T1,D1)	d(T1,D2)	d(T1,D3)	•••	d(T1,Dj)
	T2	d(T2,D1)	d(T2,D2)	d(T2,D3)	•••	d(T2,Dj)
	T3	d(T3,D1)	d(T3,D2)	d(T3,D3)	•••	d(T3,Dj)
	⋮	⋮	⋮	⋮	⋮	⋮
	Ti	d(Ti,D1)	d(Ti,D2)	d(Ti,D3)	•••	d(Ti,Dj)

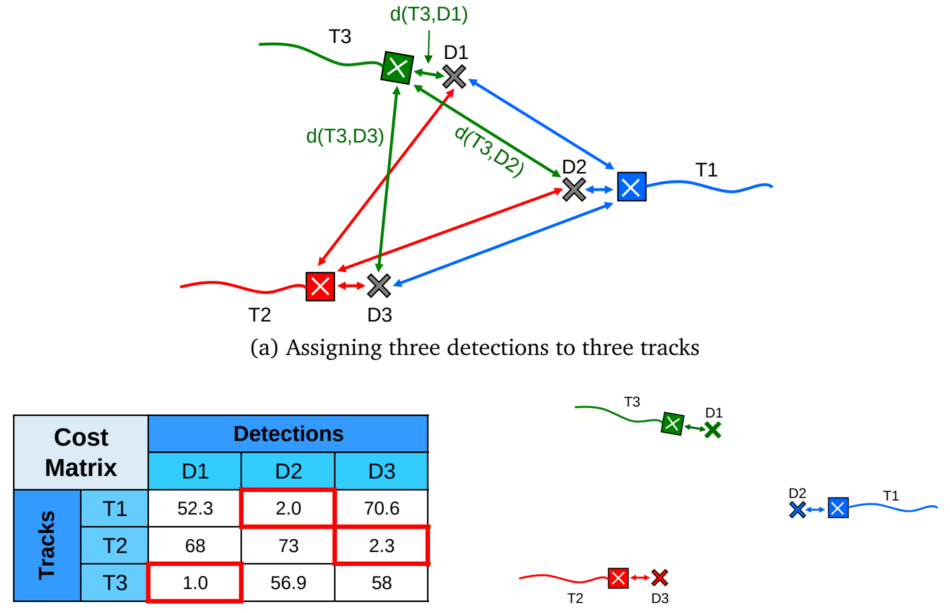
Figure 4.49: Cost matrix

using red rectangles in Figure 4.50b) is $2.0 + 2.3 + 1.0 = 5.3$.

However, if there is a detection which is relatively far from the existed tracks, applying the Hungarian algorithm to the cost matrix (shown in Figure 4.49) may give wrong assignment results. This is illustrated in example (2) as shown in Figure 4.51a. Here, the cost matrix is calculated as depicted in Figure 4.51b. Based on this matrix, the results of the assignment is shown in Figure 4.51c. In this case, the total minimum cost is $25 + 50 = 75$, and D1 is assigned to T1 while D2 is assigned to T2. However, for the correct assignment, D2 should be assigned to T1 instead of T2. Additionally, D1 should not be assigned to any of the existing tracks, and a new track (T3) must be created and associated with this detection.

To solve this problem, the cost matrix (shown in Figure 4.49) is padded with extra detection columns using a predefined distance threshold d_thr as shown in Figure 4.52. In this case, if the calculated distance values between the current position of a track and all the detections are greater than this d_thr value, this track is assigned this padded detection (d_thr). The empty rectangles in the padded cost matrix (shown in Figure 4.52) are filled with a very large number (10000 is used in this system) to ensure they are not assigned to any track, and only the padded detection with the d_thr value can be used for the assignment. The selected d_thr value in this work is based on empirical tests. The number of the padded detection columns is equal to the number of the existing tracks.

For the previous case (example (2) shown in Figure 4.51a), the padded cost matrix with a threshold value ($d\_thr = 5$) is used as shown in Figure 4.53a. In this case, two additional detection columns are padded (since there are two tracks). After applying the Hungarian algorithm to this matrix, D2 is assigned to T1 and D4 is assigned to T2 with the total minimum cost of $2 + 5 = 7$ as shown in Figure 4.53b. Since D4 is a padded detection and not a real detection, Track T2 is considered as an unassigned track.



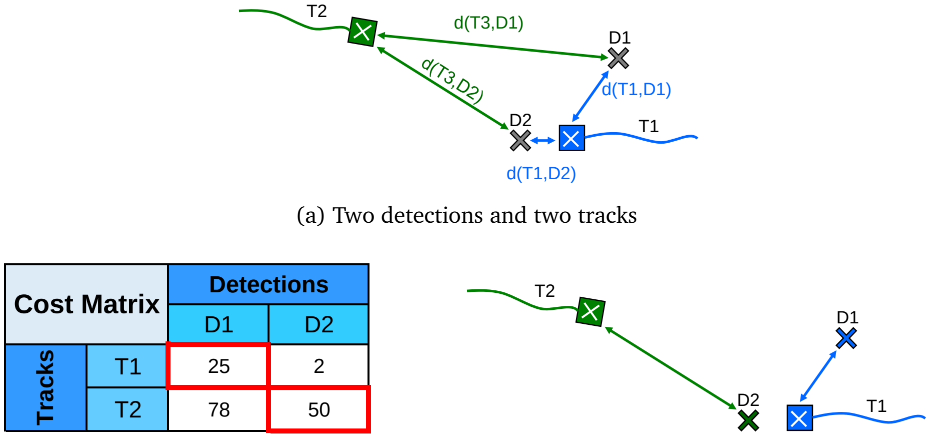
(a) Assigning three detections to three tracks

Cost Matrix		Detections		
		D1	D2	D3
Tracks	T1	52.3	2.0	70.6
	T2	68	73	2.3
	T3	1.0	56.9	58

(b) Cost matrix filled with Euclidean distance values

(c) Assiged detections to tracks results

Figure 4.50: Example (1) - Detections assignment to tracks



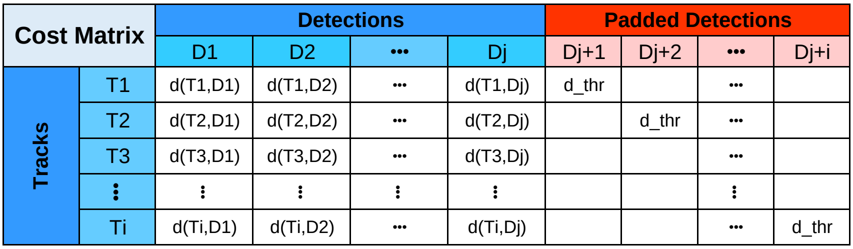
(a) Two detections and two tracks

Cost Matrix		Detections	
		D1	D2
Tracks	T1	25	2
	T2	78	50

(b) Cost matrix

(c) Detections are incorrectly assigned to tracks

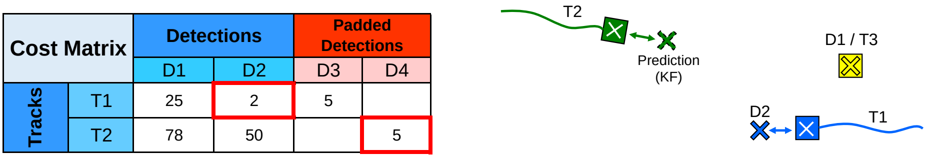
Figure 4.51: Example (2) - Detections are incorrectly assigned to tracks



Cost Matrix		Detections				Padded Detections			
		D1	D2	•••	Dj	Dj+1	Dj+2	•••	Dj+i
Tracks	T1	d(T1,D1)	d(T1,D2)	•••	d(T1,Dj)	d_thr		•••	
	T2	d(T2,D1)	d(T2,D2)	•••	d(T2,Dj)		d_thr	•••	
	T3	d(T3,D1)	d(T3,D2)	•••	d(T3,Dj)			•••	
	⋮	⋮	⋮	⋮	⋮			⋮	
	Ti	d(Ti,D1)	d(Ti,D2)	•••	d(Ti,Dj)			•••	d_thr

Figure 4.52: Cost matrix with padded columns (detections)

Therefore, the prediction value from Kalman filter is used for T2 in this frame. Furthermore, D1 is not assigned to any track as can be seen in Figure 4.53a. Since D1 is a real and not padded detection, it is used to create a new track (T3) as shown in Figure 4.53b.



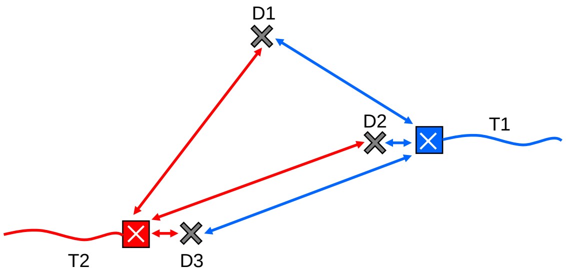
Cost Matrix		Detections		Padded Detections	
		D1	D2	D3	D4
Tracks	T1	25	2	5	
	T2	78	50		5

(a) Padded cost matrix with $d\_thr = 5$

(b) Correct result after the assignment

Figure 4.53: Solving assignment problem for example (2)

Example (3) shows another assignment scenario, by which the number of detections is higher than the number of tracks as depicted in Figure 4.54. In this example, there are three detections (D1, D2, and D3) and two tracks (T1 and T2), resulting in one unassigned detection as shown in Figure 4.54a. Figure 4.54b show the padded cost matrix and the results after applying the Hungarian algorithms. As can be seen, there is one unassigned detection (D1) which is used to create and initialize a new track (T3). On the other hand, if the number of detections is less than the number of tracks, the prediction value from Kalman filter is used for the track that has no detection assigned to it. This case is shown in example (4), depicted in Figure 4.55a. In this example, there are two detections (D1 and D2) and three tracks (T1, T2, and T3). The padded cost matrix is shown in Figure 4.55b. In this case, the two existed real detections are assigned to two tracks, while track (T3) gets the padded detection (D5) for the assignment with a total minimum cost of $2.0 + 2.3 + 5 = 9.3$. Since T3 is assigned a padded detection, the value of the predicted position from Kalman filter is used as the track's position for this frame.

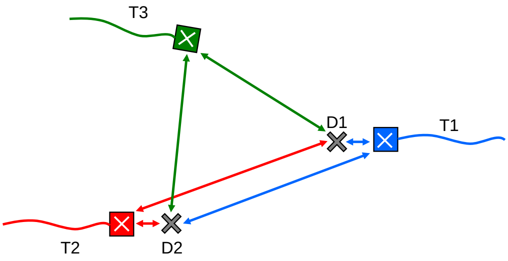


(a) Three detections and two tracks

Cost Matrix		Detections			Padded Detections	
		D1	D2	D3	D4	D5
Tracks	T1	52.3	2.0	70.6	5	1000
	T2	68	73	2.3	1000	5

(b) Cost matrix and the assignment results

Figure 4.54: Example (3) - An unassigned detection



(a) Two detections and three tracks

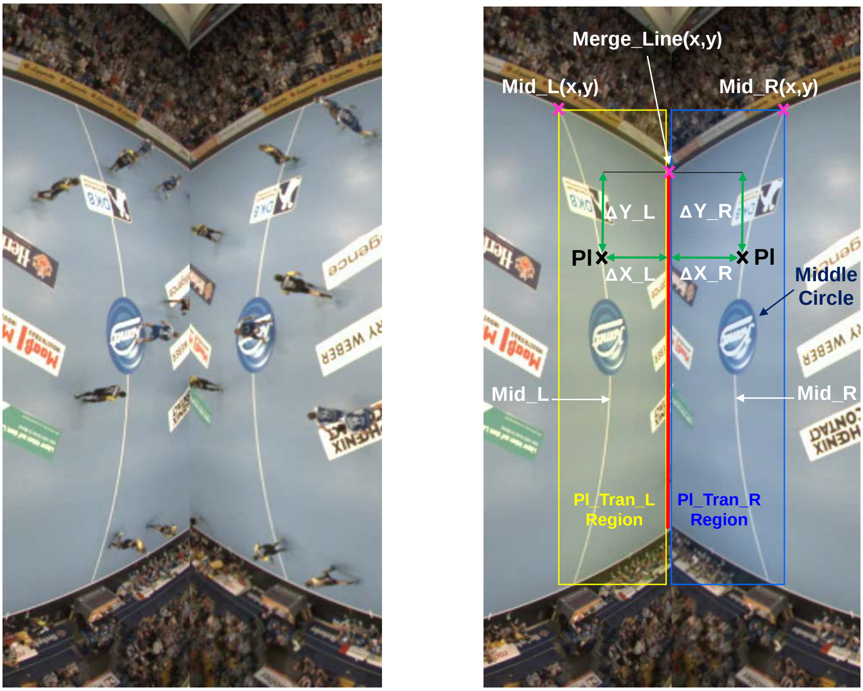
Cost Matrix		Detections		Padded Detections		
		D1	D2	D3	D4	D5
Tracks	T1	2.0	70.6	5	1000	1000
	T2	73	2.3	1000	5	1000
	T3	56.9	58	1000	1000	5

(b) Cost matrix and the assignment results

Figure 4.55: Example (4) - An unassigned track

4.7.3 Player Track Transfer Between the Two Cameras

Since two cameras (left and right) with an overlapping region are used in this work to cover the whole court as depicted in Figure 4.1, player tracking must be maintained when the players are moving from one camera view to the other. The proposed automatic player transfer method is shown in Figure 4.56. Here, an overlapping region is used where the players are visible in the two frames from the right and left camera as shown in Figure 4.56a. In this figure, the players are moving from the right to the left direction. Furthermore, two regions are defined where each player position is checked: player transfer left (Pl_Tran_L) and right (Pl_Tran_R) as shown in Figure 4.56b with the yellow and blue rectangles, respectively. These regions are defined by the court middle line which is visible in both camera views (Mid_L and Mid_R) and the merge line (the borderline where the two frames are merged). If a player is detected inside one of these regions, the player transfer algorithm is applied to transfer the tracker from one camera to the other depending on the movement direction of that player [107].



(a) Players are visible in the two cameras

(b) Main parameters used for player transfer [107]

Figure 4.56: Player transfer between the two cameras

The realized algorithms to transfer the player's track from the left to the right camera and from the right to the left camera are shown in Algorithm 4.2 and Algorithm 4.3, respectively. These algorithms are based on the fact that the distance between Mid_L and Merge_line is equal to the distance between Mid_R and Merge_line [66]. As an example, if a player (Pl) reaches the white middle line (Mid_L), and he is moving from the left to the right camera, his distance to the middle line (Δ X_L and Δ Y_L) in the left image is equal to his distance to the middle line in the right image (Δ X_R and Δ Y_R) as shown in Figure 4.56b. However, the two merged frames from the cameras are usually not perfectly aligned. Therefore, a predefined threshold is used when the player's position in one camera is compared to his position in the other camera. Additionally, the direction of player movement is extracted from the difference (using the x coordinate) of the player's position between the current frame and the previous n frame (e.g., 20). A positive difference value means that the player is moving from the left to the right direction, while a negative value means the player is moving from the right to the left direction.

Algorithm 4.2 Player transfer from the left to the right camera [107]

```
for i = 1 to number of players do
    if player(i).position is inside the Pl_Tran_L region then
        Calculate $\Delta X\_L$ & $\Delta Y\_L$ between player(i).position and Merge line
        for j=1 to number of players do
            if player(j).position is inside Pl_Tran_R region then
                Calculate $\Delta X\_R$ & $\Delta Y\_R$ between player(j).position and Merge line
                if $|\Delta X\_L - \Delta X\_R| < \Delta X\_Thr$ AND
                   $|\Delta Y\_L - \Delta Y\_R| < \Delta Y\_Thr$ then
                    if (Player(i).age > player(j).age) AND
                       (Player(i).trace(last) − Player(i).trace(first) > 0) then
                        Copy player(j) to player(i)
                        Delete player(j)
                        break
```

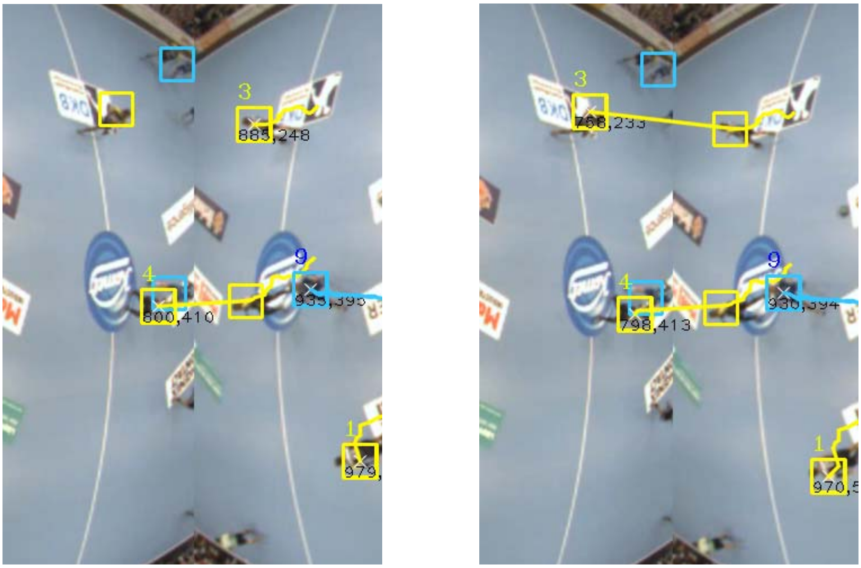
Therefore, a player's track is transferred from the left to the right camera (according to Algorithm 4.2) if three conditions are met. The first one is to verify that the player in left camera view (Pl_Tran_L region) is the same player in the right camera view (Pl_Tran_R region). The next step is to verify that the track that corresponds to the player in the (Pl_Tran_R) region is a newly created track (since the player has recently appeared in the right camera view). Finally, the direction of player's movement from the left to the right camera view is verified. If all these conditions are met, the parameters of the corresponding track in the (Pl_Tran_L) region are copied to the track in the (Pl_Tran_R) region. After that, the track in the (Pl_Tran_L) region is deleted [107].

The player transfer from the right to the left camera view is performed vice versa, and it is shown in Algorithm 4.3.

Algorithm 4.3 Player transfer from the right to the left camera

```
for i = 1 to number of players do
    if player(i).position is inside the Pl_Tran_R region then
        Calculate $\Delta X\_R$ & $\Delta Y\_R$ between player(i).position and Merge line
        for j=1 to number of players do
            if player(j).position is inside Pl_Tran_L region then
                Calculate $\Delta X\_L$ & $\Delta Y\_L$ between player(j).position and Merge line
                if $|\Delta X\_R - \Delta X\_L| < \Delta X\_Thr$ AND
                   $|\Delta Y\_R - \Delta Y\_L| < \Delta Y\_Thr$ then
                    if (Player(i).age > player(j).age) AND
                       (Player(i).trace(last) − Player(i).trace(first) < 0) then
                        Copy player(j) to player(i)
                        delete player(j)
                        break
```

Figure 4.57 shows an example of players transfer from the right to the left camera. In figure 4.57a, the player with ID 3 approaches the middle line in the right camera frame. At the same time, he is detected in the frame from the left camera. Using algorithm 4.3, the player's track is transferred from the right to the left camera view in the next frame as shown in Figure 4.57b [107].



(a) Frame n

(b) Frame n+1

Figure 4.57: Player transfer from the right to the left camera [107]

4.8 Summary

In this chapter, the proposed reconfigurable system for player tracking in indoor sports has been presented. It consists of the FPGA architecture and the CPU-based processing system. The FPGA architecture includes various modules, performing the compute-intensive vision processing tasks. These modules are video acquisition, video preprocessing, player segmentation, and team identification & player detection modules. Each module consists of different IP cores, by which the design and implementation of these cores on the FPGA are presented. Additionally, the resource utilization of these cores and the overall FPGA architecture is depicted. The complete FPGA system requires around 60% from the available resources of the Virtex-4 FPGA. The output of the FPGA is the detected objects, including the players that are sent to the host-PC for further processing.

On the host-PC, player tracking is achieved using the tracking-by-detection approach. The received detections from the FPGA are used to create tracks, consisting of different parameters. Player tracking is achieved by associating subsequent detections to these tracks. Finally, player tracks transfer between the two cameras is performed, achieving player tracking over the whole sports court.

In the next chapter, the proposed system is evaluated using benchmark datasets. The evaluation includes player detection and tracking as well as performance analysis of the FPGA architecture. Additionally, a comparison of the proposed system with the existing work shown earlier in Section 2.5 is depicted.

5 System Evaluation and Results

In this chapter, the proposed reconfigurable system is analyzed and evaluated. First, the system realization and the used datasets for the evaluation are shown, followed by the player detection and tracking evaluation. Subsequently, the performance evaluation of the FPGA architecture is presented. Finally, a comparison of the proposed system with the other systems that are presented in the related work section is depicted.

5.1 System Realization

The proposed reconfigurable system is evaluated in real hardware using the FPGA-based modular rapid prototyping RAPTOR platform [74]. This platform is connected to a host-PC which is equipped with an Intel i7-870 CPU (quad-core at 2.93 GHz). As depicted in Section 3.7, the RAPTOR platform supports up to six FPGA daughterboards. In this work, one daughterboard is used which consists of a Xilinx Virtex-4 (XC4VFX100) FPGA. Additionally, a display daughterboard and a Gigabit Ethernet extension board are used. This Ethernet board provides two Gigabit Ethernet interfaces as shown in Figure 5.1. The proposed design is targeted to use one FPGA for handling two cameras, and it is scalable to support more cameras if needed [107].

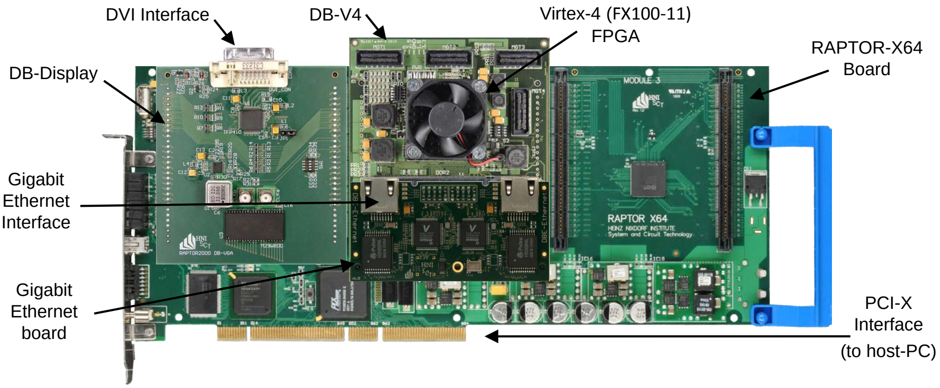


Figure 5.1: Realization of the proposed system using the RAPTOR-X64 platform

A multithreaded C++ program using the OpenCV library is implemented on the host-PC. In case of offline video processing, this program consists of the main thread which calls three other threads: the video files reader, processing, and video display threads as shown in Figure 5.2. The video files reader thread reads the two video files that correspond to the video streams from the left and right camera. The processing thread sends the video file data to the FPGA, waits until the data is processed, and then reads the output results (the detected objects for both teams) from the FPGA. Next, the processing thread creates, updates, and manages the tracks based on the detection results and performs the player's track transfer between the two cameras if required. Finally, the video display thread displays the detection and the tracking results on the host-PC [107]. Additional parameters for debugging and evaluation can be also displayed (e.g., number of detections and tracks of each team in every frame, current frame number, frame rate (fps), etc.) [108][1].

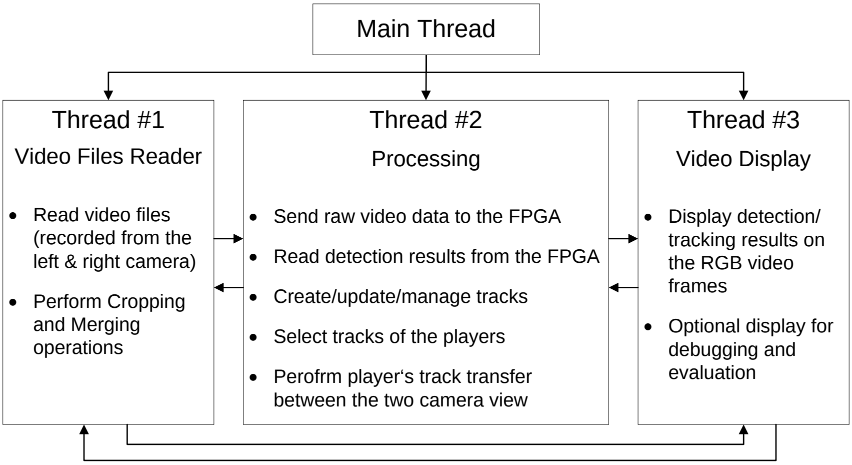

Main Thread
Thread #1
Video Files Reader
• Read video files (recorded from the left & right camera)
• Perform Cropping and Merging operations
Thread #2
Processing
• Send raw video data to the FPGA
• Read detection results from the FPGA
• Create/update/manage tracks
• Select tracks of the players
• Perofrm player's track transfer between the two camera view
Thread #3
Video Display
• Display detection/ tracking results on the RGB video frames
• Optional display for debugging and evaluation

Figure 5.2: C++/OpenCV multithread implementation on the host-PC, realizing the proposed system for player tracking using recorded video files

5.2 Datasets

For the evaluation of our system, three datasets (each consisting of two video files) are used. These datasets are captured using two GigE Vision cameras with a resolution of 1392x1040 pixels and a frame rate of 30 fps for each camera. The first dataset is a

[1] A live demo of the proposed system along with the paper [108] is presented at the DASIP 2017 conference in Dresden, Germany.

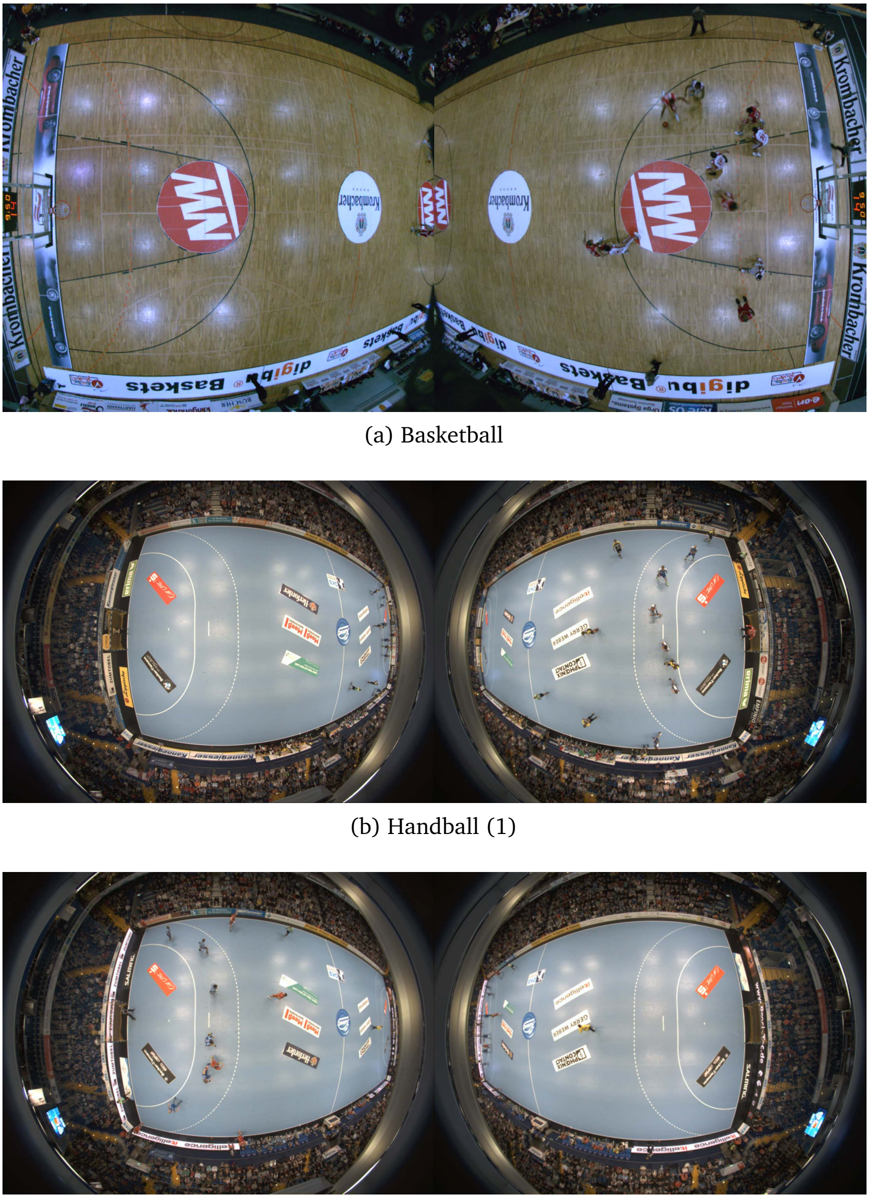
basketball game (as shown in Figure 5.3a) stored in a raw format where each pixel is represented by an 8 bit Bayer pattern. The second and third dataset are handball games (named as handball (1) and (2)), where the video data is stored in two formats. The first one is a raw data with Bayer pattern (8-bit for each pixel). The second format is a compressed RGB with 24 bits for each pixel after white balancing [107]. The handball (1) and (2) datasets are shown in Figures 5.3b and 5.3c, respectively. In addition to these three datasets, the APIDIS dataset [7] is used to compare the performance of the proposed system with another work presented in the literature as shown in the next section.

The annotated ground truth for these datasets was generated using the VitBAT tool [21] under human supervision. Since this annotation process requires a significant amount of interaction in order to achieve correct data for each player, we used 5000 frames for each dataset to evaluate the proposed system. During these frames, players are moving several times between the two camera views. Additionally, there are player substitutions. For the handball (1) dataset as an example, all the players moved three times between the two camera views and there are two substitutions for team 1 and three substitutions for team 2 during these 5000 frames [107].

To detect the players in the handball (1) dataset, the yellow color is used for the color mask of team 1, whereas, white and blue are used for team 2. Player detection in the handball (2) dataset is achieved based on the orange color for team 1 and blue for team 2. For the basketball dataset, red is used for team 1 and white is used for team 2. Player detection using the color of players' jerseys is challenging in these datasets since the colors that are used in the masks are shared by other objects in the sports hall. Some examples of such scenarios are: similar color in the advertisement panels as well as in the jersey of the opposing team as it is the case in the used basketball dataset where the jersey of team 1 is mostly in red with some white color and the opposite for team 2 (mostly in white with some red color) [107]. The evaluation of player detection for these datasets are shown in the next section.

5.3 Player Detection

In this work, precision and recall [37] are used as standard metrics to evaluate players detection as presented in Section 2.1. Precision is the ratio between the number of correctly detected players (TPs) and all the detections (TPs and FPs). Recall (also called detection rate) is the number of the players that are correctly detected (TPs) among the total number of players that should have been detected (TPs and FNs, i.e., the ground truth which represent the total number of players in a team). Based on Equation 2.1, the average precision and recall are calculated for the previously mentioned datasets using Equations 5.1 and 5.2., where N_{Frame} is equal to 5000. For illustration, Figure 5.4 shows a handball scene with a detection result using the proposed system. In this scene,



(a) Basketball

(b) Handball (1)

(c) Handball (2)

Figure 5.3: Example scenes from the three datasets used in the evaluation [107]

there are true positives (TPs) which are correctly detected players, false positives (FPs) which represent the incorrect detections (non-players), and players who should be detected but they are not, are the false negatives (FNs).

$$Precision = \frac{1}{N_{Frame}} \sum_{i=1}^{N_{Frame}} \frac{TP_i}{TP_i + FP_i} \times 100\% \tag{5.1}$$

$$Recall = \frac{1}{N_{Frame}} \sum_{i=1}^{N_{Frame}} \frac{TP_i}{TP_i + FN_i} \times 100\% \tag{5.2}$$

$$Recall = \frac{1}{N_{Frame}} \sum_{i=1}^{N_{Frame}} \frac{TP_i}{N_{Players}} \times 100\%$$

where:
TP_i is the number of true positives at frame i.
FP_i is the number of false positives at frame i.
FN_i is the number of false negatives at frame i.
N_{Frame} is the total number of frames.

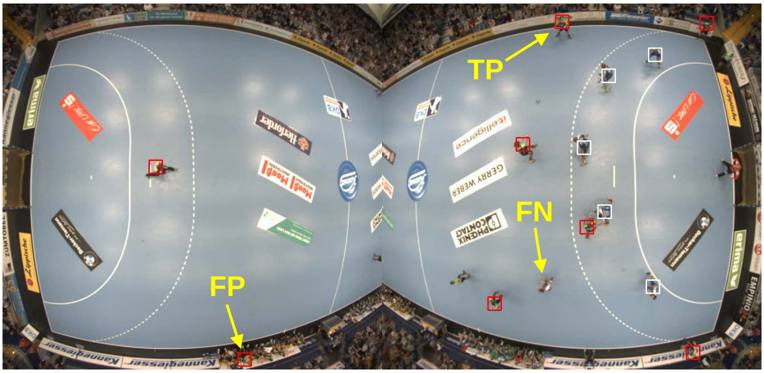


Figure 5.4: A handball scene with detection results showing TPs, FPs, and FNs

In this evaluation, a detection result is considered as a true positive if this detection is inside the bounding box of a player using the generated ground truth data. Furthermore, the Hungarian algorithm is used to match the detection results with the ground truth data. The matching cost is the Euclidean distance between the detection and the bounding box center of the ground truth. This will prevent assigning two detections to the same ground truth data (i.e., same player) in a frame. The precision and recall based on the detection output results from the FPGA are shown in Table 5.1 [107].

Table 5.1: Results of player detection for the used datasets [107]

Dataset	Team	Precision	Recall	Avg. Prec.	Avg. Rec.
Handball (1)	T1	80.9%	94.58%	84.02%	91.94%
	T2	87.14%	89.29%		
Handball (2)	T1	72.59%	98.93%	71.42%	96.6%
	T2	70.25%	94.27%		
Basketball	T1	71.73%	98.57%	67.49%	96.14%
	T2	63.24%	93.7%		

As shown in Table 5.1, the achieved average detection rate (recall) for the basketball dataset is 96.14%. For the handball (1) and (2) datasets, the achieved average recall values are 91.94% and 96.6%, respectively. However, the achieved average precision is 67.49% for the basketball as well as 84.02% and 71.42% for the handball (1) and (2) datasets, respectively. Since our approach to detect the players is based on the color, the recall and precision values may differ between the two teams. If an object (e.g., an advertising or sponsor panel) shares the same color with the jersey of one team, false positives are introduced which reduce the precision. One example of such a scenario is team 2 in the basketball dataset. In this case, the white color is used to detect the players of that team. However, this color is used by the players of team 1 as well as in the sponsor and advertisement panels as shown in Figure 5.3a. Additionally, the light reflections on the court's ground are also in white and could introduce additional false positives when a player crosses over them. Nevertheless and as discussed earlier in the tracking section in the previous chapter, the effect of these FPs is significantly reduced by the post-processing (tracking) in the host-PC as shown in the next section, where the player tracking evaluation is presented [107].

Figure 5.5 shows the precision and recall for both teams over the used 5000 frames for the handball (1) dataset (cf. Figure 5.3b). As can be seen in the upper diagram, the precision of both team 1 and 2 drops in three periods (shown using the black dashed lines): from around frame number 1280 to 1400, 2810 to 2990, and 3820 to 3920. In these periods, the players moved from one camera view to the another. In this case, the players are in the overlapping region where they appeared in both the left and right camera view, and hence they are detected twice (cf. Figure 4.57). Here, one of the two detections that belong to the same player is considered as a FP, reducing the precision value. Another period where the precision of team 1 decreases ranges from frame 2700 to 3800 as shown using the green dotted lines in Figure 5.5. Beside the player transfer effect, additional false positives are introduced from the moving substitute players of that team and from the digital advertisement panel which changed its content in that period. The effects of these false positives are reduced using two approaches. The first one is by the frequent update of the background. In this case, these substitute players and advertising panels are considered as parts of the background, and hence not detected after a certain time (e.g., after frame 3800). The second approach is through the tracking processing, where these false positives are used to create tracks with usually low player selector confidence value [107].

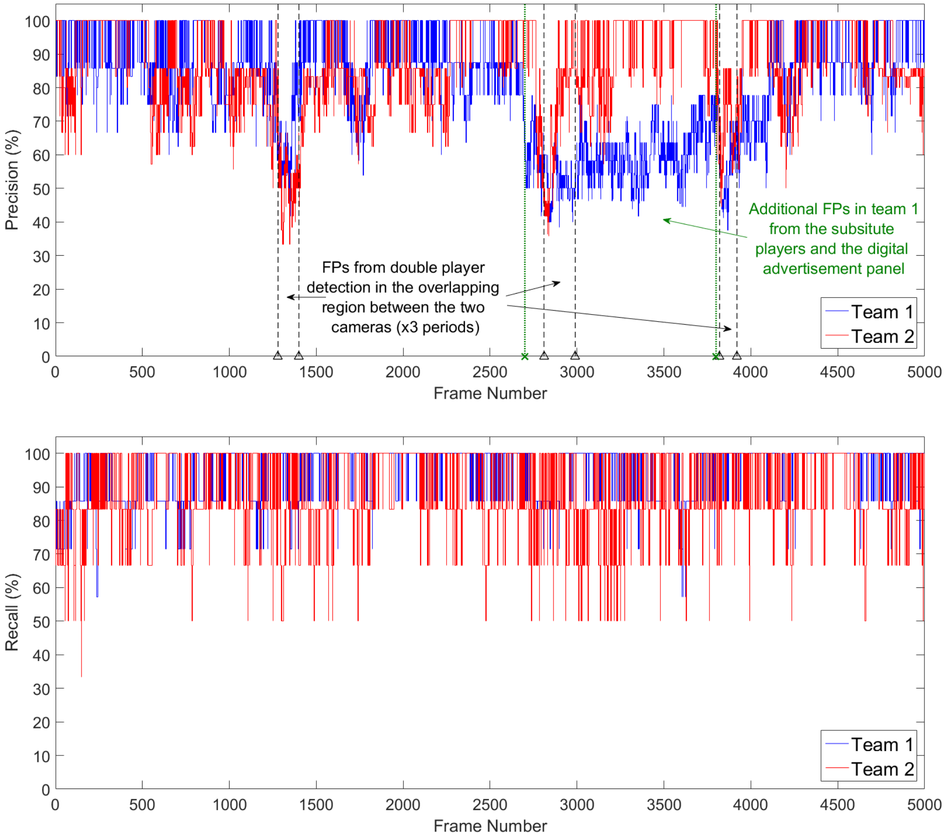


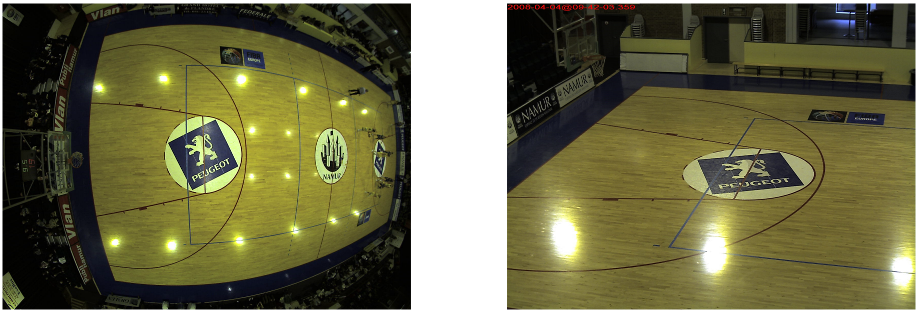
Figure 5.5: Precision and recall using the detection results for teams 1&2 (handball (1) dataset) [107]

For the recall values shown in Figure 5.5, team 1 achieved high recall while team 2 has lower values. A low recall means there are players who are not detected (FN). However, the recall is improved by using the prediction values from Kalman filter for the not detected player in the tracking step on the host-PC [107].

Table 5.2 shows comparison results of the proposed system with the work presented by Alahi et al. [4] based on the APIDIS basketball dataset (shown in Figure 5.6) [7]. As stated in the related work section, the authors downscaled all the images to a resolution of 320x240 pixels to reduce the computation cost. The results reported in [4] are for player detection over the left-half of the basketball court. To perform the detection measurements for the proposed system, the dataset from the fisheye camera (shown in Figure 5.6a) with the manually annotated player positions that are provided in [7] for one minute are used. Compared to the results reported in [4], the proposed system achieves better precision and recall. For team 1, a precision of 87.3% and a recall of 91.6% are achieved, whereas a 57.3% precision and a 94.4% recall are obtained for team 2. The average achieved precision is 72.3% and recall is 93%, compared to 72% and is 76% as reported in [4].

Table 5.2: Comparison with the work presented in [4] [108] [107]

System	Cameras	Resolution	Precision	Recall
Alahi et al. [4]	1xFisheye (Figure 5.6a)	downscaled to 320x240	55%	47%
Alahi et al. [4]	1xFisheye + 1xLinear (Figures 5.6a & 5.6b)	downscaled to 320x240	72%	76%
The proposed system	1xFisheye (Figure 5.6a)	Original (1600x1200)	72.3%	93%



(a) Fisheye camera (left)

(b) Linear camera (left)

Figure 5.6: APIDIS dataset [7] that are used for the comparison

If the overall movement is quite high (e.g., including movement from spectators), these movements will not have a significant impact on the detection results due to the following reasons. Firstly, since the dimensions of the sports court are fixed, any detections from frequently moving objects outside the court can be discarded without affecting the detections of the players inside the court. Secondly, the background is continuously updated, reducing the effect of moving non-player objects. Apart from the overall movement, shadows also do not have a significant influence on the system in general. One reason is the nature of an indoor sports hall, where the lighting conditions are usually very good and constant due to multiple well-distributed light sources. Furthermore, the player detection is based on the HSV color space, which is more robust to illumination change than the RGB color space. Additionally, if there are false positives caused by shadows, these FPs are significantly reduced by the post-processing in the host-PC [107].

5.4 Player Detection in Occlusion Scenarios

In handball and basketball games, player occlusions can generally be divided into two types: Occlusion between players of opposing teams and occlusions between players of the same team. The first type of occlusion is the most frequent in these indoor games, while occlusions between the same team's players are less common. In the used datasets (cf. Subsection 5.1) as an example, there were 41 occlusions between players of the opposing teams for the handball (1) dataset and 18 occlusions between players of the same team (10 occlusions for Team 1, and 8 for Team 2). For the basketball dataset, there were 67 occlusions between players of the opposing teams and 30 occlusions between players of the same team (17 and 13 occlusions for Team 1 and Team 2, respectively) [107].

Figure 5.7 shows the player detection using our system for the two types of occlusions between players. Our system shows promising results and robustness against occlusion between two players from opposing teams. As shown in the occlusion scenario (A) in Figure 5.7, the players from different teams are separated into two video frames (one for each team) using the color masking (thresholding). On each video frame, clustering is applied independently, and therefore the players are detected as shown in Figure 5.7 (scenario (A)). Successful player detections for various occlusions between players of opposing teams in handball and basketball are shown in Figures 5.8 and 5.9, respectively [107].

Player detection using the proposed system for occluded players of the same team is shown in Figure 5.7 (scenario (B)). As can be seen, two players of the same team can be detected as one player, resulting in one FN. This is because the vertices that are

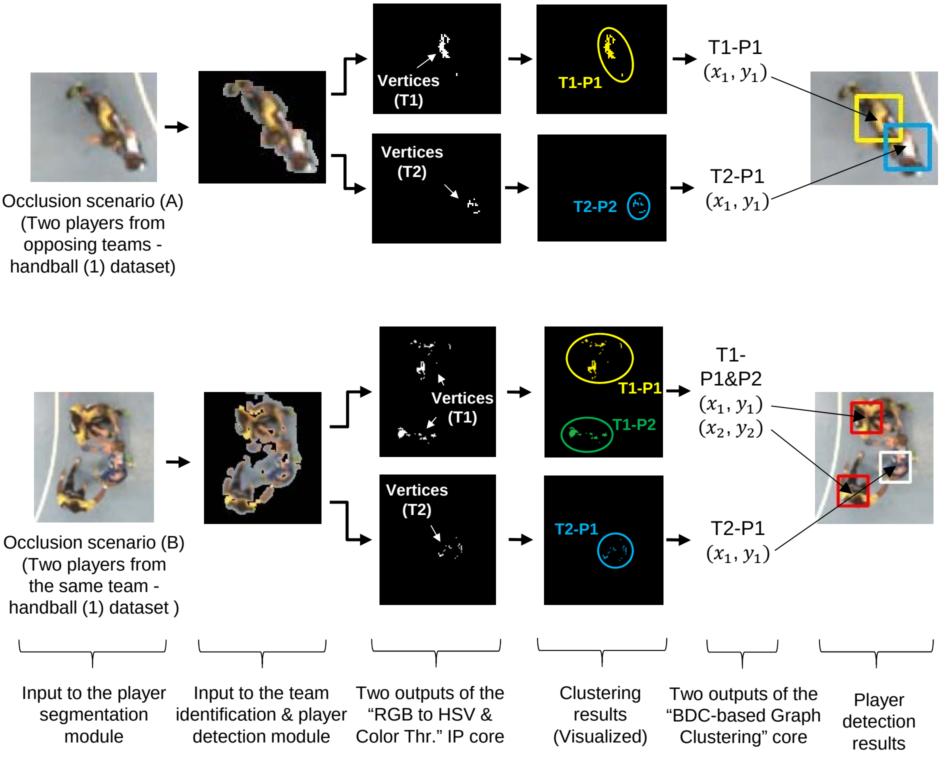

Vertices (T1)
T1-P1
T1-P1
(x_1, y_1)
Vertices (T2)
T2-P2
T2-P1
(x_1, y_1)
Occlusion scenario (A) (Two players from opposing teams - handball (1) dataset)
Vertices (T1)
T1-P1
T1-P2
T1-P1&P2
(x_1, y_1)
(x_2, y_2)
Vertices (T2)
T2-P1
T2-P1
(x_1, y_1)
Occlusion scenario (B) (Two players from the same team - handball (1) dataset)
Input to the player segmentation module
Input to the team identification & player detection module
Two outputs of the "RGB to HSV & Color Thr." IP core
Clustering results (Visualized)
Two outputs of the "BDC-based Graph Clustering" core
Player detection results

Figure 5.7: Player detection for occluded players (scenario A and B) [107]

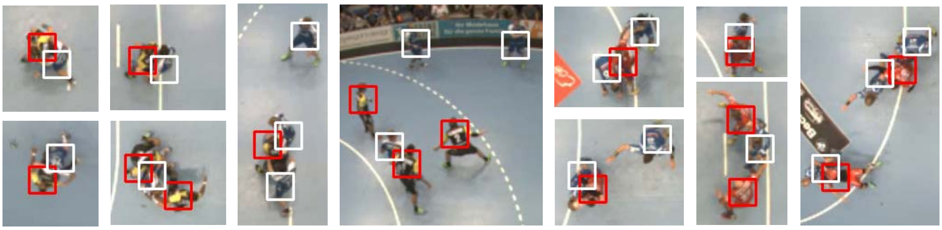


Figure 5.8: Successful detections for different occlusion scenarios between players of opposing teams in a handball game [107]

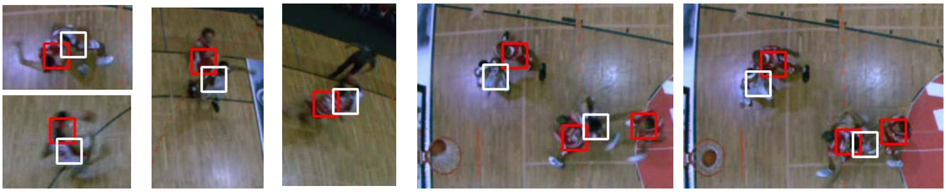


Figure 5.9: Successful detections for different occlusion scenarios between players of opposing teams in a basketball game [107]

generated from these two players are clustered as one player (T1-P1 in scenario (B)) since the binary distance between these vertices are smaller than the used threshold for maximum distance in the clustering. Additional examples of unsuccessful detections for such occlusions in handball and basketball games are shown in Figure 5.10 [107].

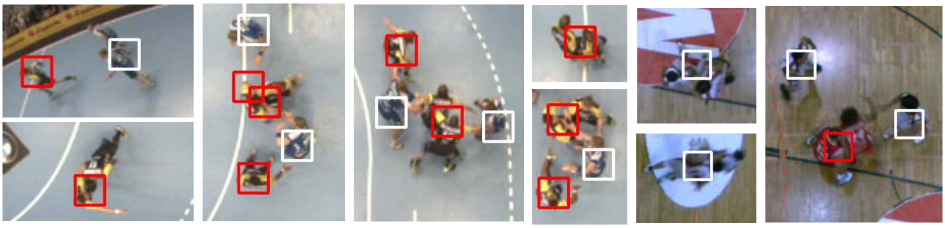


Figure 5.10: Unsuccessful detections for occlusion scenarios between players of the same team [107]

However, these unsuccessful detections due to occlusions between players of the same team have a low impact on the overall performance of player tracking, since when the two occluded players are detected as one (resulting in one not detected player), the prediction value from Kalman filter will be used for this not detected player as discussed in Subsection 4.7.1 [107].

5.5 Verification of the Player Detection Implementation

The hardware implementation of the team identification & player detection module is verified by comparing its output with the results obtained from a software implementation using C++ and OpenCV of the same module. The output of this software implementation is considered as the reference for the comparison. This verification process is illustrated in Figure 5.11, where the input are images with segmented players from the handball (1) dataset. First, player detection using the software implementation is applied to these images. The output consists of the detected clusters, including the number of vertices and the centroid of each cluster as shown in Figure 5.11. Later, the same input images are used for the hardware implementation of the player detection, and the results are recorded. Finally, the outputs of both implementations are compared to verify if the hardware implementation of the player detection module achieves the expected results (in comparison with the software implementation). The results of this comparison for one of the used images are shown in Tables 5.3 and 5.4 for team 1 and 2, respectively. In these comparisons, the same color threshold values (cf. Equations 4.6, 4.7, and 4.8) are used for both hardware and software implementations. Additionally, the same threshold values of the maximum distance (cf. Equation 4.11) and the minimum number of vertices (cf. Figure 4.41) are used for both implementations. These values are required for the clustering process.

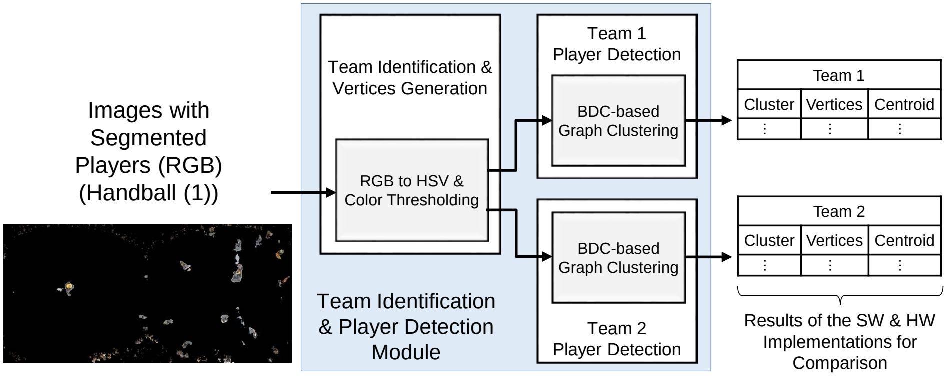

Images with Segmented Players (RGB) (Handball (1))
Team Identification & Vertices Generation
RGB to HSV & Color Thresholding
Team Identification & Player Detection Module
Team 1 Player Detection
BDC-based Graph Clustering
BDC-based Graph Clustering
Team 2 Player Detection
Team 1
Cluster | Vertices | Centroid
Team 2
Cluster | Vertices | Centroid
Results of the SW & HW Implementations for Comparison

Figure 5.11: Validation of the player detection implementation

As can be seen in Table 5.3, the total number of detected clusters in both implementations are equal (12 clusters are detected). These clusters represent TPs and FPs. However, the goal here is not to evaluate the detection results (e.g., precision and recall), but to compare the hardware results with their corresponding ones in software. Therefore, the first observation is that all twelve clusters are detected using the hardware implementation. Additionally, there are detected centroids (in hardware) which do not match the centroids from the software implementation. In this case, a maximum difference of 3 can be observed between the resulted centroids from the software and hardware implementations.

Cluster 2 is an example where the detected centroid from the hardware implementation does not exactly match the resulted detection using the software implementation.

Table 5.3: Comparison between the software and hardware implementations' results of the player detection module for team 1 using one image from the handball (1) dataset

Team1					
SW Implementation (C++)			HW Implementation (VHDL)		
Cluster	Vertices Nr	Centroid (x,y)	Cluster	Vertices Nr	Centroid (x,y)
1	7	(1193,28)	1	7	(1193,28)
2	23	(1586,185)	2	19	(1585,185)
3	28	(1032,233)	3	27	(1032,233)
4	80	(1361,278)	4	43	(1361,275)
5	133	(385,354)	5	64	(388,355)
6	5	(867,674)	6	5	(867,674)
7	12	(1039,696)	7	12	(1039,695)
8	12	(1581,736)	8	10	(1581,736)
9	22	(628,755)	9	22	(625,758)
10	11	(574,772)	10	11	(574,772)
11	8	(607,767)	11	6	(606,767)
12	4	(531,776)	12	4	(531,776)
12 (Total)	323 (Total)	-	12 (Total)	230 (Total)	-

As shown in Table 5.3, this cluster is detected in software with a centroid of (1586,185) using 23 vertices. As shown in Section 4.5.1, these vertices resulted from the color thresholding after HSV conversion. Using the hardware implementation, cluster 2 is detected with a centroid of (1585,185) using 19 vertices instead of 23. As can be seen, there is a difference of one in the X coordinates of the two centroids. This is because there are four vertices (and therefore four pixels) that are not included in the calculation of the centroid in hardware, which eventually effect the resulted centroid. The reason behind these four missing pixels is the modification that is applied to the RGB to HSV conversion for the hardware implementation as depicted in Section 4.5.1. As an example, one of these four pixels is located at (1598,190), and it has the RGB values of (198,164,99). Using Equations 3.12, 3.13, and 3.14 for the software implementation, the equivalent HSV values are (20,127,198). To detect the players in team 1, yellow is used for this dataset. The used HSV color thresholding range is shown in Equation 5.3. Therefore, Q_{T1} is equal to 1 (i.e., a vertex is generated) for this pixel using the software implementation.

$$Q_{T1} = \begin{cases} 1, & 20 \leq H \leq 30 \text{ AND} \\ & 100 \leq \ S \leq 150 \text{ AND} \\ & 160 \leq V \leq 255 \\ & \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

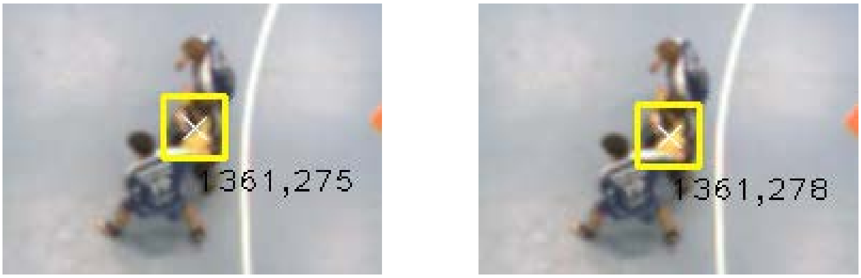
For the hardware implementation, Equations 4.3 and 4.5 are used to calculate the normalized $H \times \Delta$ and $S \times \max(RGB)$, respectively. Consequently, the pixel is considered as a vertex if the conditions in Equation 4.10 are fulfilled. For this pixel with the RGB values of (198,164,99), Δ is equal to 198-99=99, $H \times \Delta = 30 \times (G - B) = 1950$, and $S \times \max(RGB) = \Delta \times 256 = 25344$. After applying these values to Equation 4.10, the resulted threshold conditions are shown in Equation 5.4. As can be seen, the condition regarding the H component is not met (since 1950 is less than 1980), resulting in a Q_{T1} of 0 (i.e., no vertex is generated) using this hardware implementation. As a result, the pixel at position (1598,190) is included in the centroid calculation of cluster 2 using the software implementation, while it is not the case in the hardware implementation.

$$Q_{T1} = \begin{cases} 1, & 1980 \leq H \times \Delta \leq 2970 \text{ AND} \\ & 19800\times \leq S \times \max(RGB) \leq 29700 \text{ AND} \\ & 160 \leq V \leq 255 \\ & \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

Another pixel, which is not considered as a vertex in cluster 2, has RGB values of (185,166,76) and hence HSV values of (25,150,185). In this case, the condition, based on the $S \times \max(RGB)$ value, to generate a vertex is not fulfilled for the hardware implementation (using the modified HSV algorithm). Based on these results, a pixel may not be considered as a vertex if its H or S value is equal to the minimum or maximum value that are used in the color thresholding (e.g., H=20 and S=150 for the previously mentioned examples).

In cluster 7, the resulted centroid in software is (1039,696). However, the detected centroid has a coordinate of (1039,695) using the hardware implementation. In this case, the mismatch is due to a rounding issue in the used divider IP core. This rounding issue results in a maximum difference of one, compared to a rounded result based on a floating point division.

For these twelve detected clusters that are shown in Table 5.3, the maximum difference between the centroids of the two implementations is three. This is the case in the fourth cluster, by which the centroid (1361,275) is detected in hardware, while (1361,278) is the resulted centroid using the software implementation. Both centroids represent the player's position correctly as depicted in Figure 5.12. Here, a 30x30 bounding box is used for both Figure 5.12a and 5.12b. Therefore, the applied modifications to the RGB to HSV conversion (cf. Equations 4.3 and 4.5) do not have a significant influence on the detection results as shown in Table 5.3 and Figure 5.12, while realizing a resource-efficient and low latency hardware implementation of the player detection module.



(a) Using hardware implementaiton

(b) Using software implementation

Figure 5.12: The resulted centroid of cluster 4 in team 1

Table 5.4 shows a comparison between the resulted centroids of the software and hardware implementation for team 2. Here, white and blue are used to detect the players of team 2. In both implementations, there are seven detected clusters, and most of the centroids of these clusters are the same as shown in Table 5.4. The maximum difference value between the calculated centroids is two as the case with the fifth detected cluster shown in this table.

Table 5.4: Comparison between the software and hardware implementations' results of the player detection module for team 2 using one image from the handball (1) dataset

Team2					
SW Implementation (C++)			HW Implementation (VHDL)		
Cluster	Vertices Nr	Centroid (x,y)	Cluster	Vertices Nr	Centroid (x,y)
1	79	(1358,122)	1	78	(1357,122)
2	290	(1350,298)	2	286	(1350,298)
3	21	(1067,255)	3	21	(1067,255)
4	57	(1280,549)	4	55	(1280,549)
5	56	(1028,746)	5	51	(1026,745)
6	22	(1066,757)	6	25	(1065,757)
7	32	(1295,789)	7	31	(1295,789)
7 (Total)	557 (Total)	-	7 (Total)	547 (Total)	-

5.6 Player Tracking

While the precision and recall values presented previously are just based on the detection results, in this section the effect of tracking on these metrics is reported. When tracking is applied, true positives are increased since the prediction values from the Kalman filter are used when there are no detections. Additionally, false positives are reduced since these false detections are used to create tracks with low confidence values which are discarded in the final tracking results. However, both the precision and recall are decreased if a false track is selected instead of a player track based on the confidence value. The precision and the recall after player tracking are shown in Table 5.5. As can be seen, the average precision is almost equal to the average recall values for the individual teams. This is because the number of tracks for each team is fixed (equal to the number of players in that team) unless there are fewer tracks than the number of players (e.g., at the beginning of tracking when there could be fewer detections (and hence fewer tracks) than the players). For example, if there are seven tracks for a five players basketball team in a particular frame. Among these tracks, five tracks are selected (based on the tracks' confidence value) to represent the five players. If four tracks matched four players (4 TPs), one player would be left without being tracked (1 FN). Therefore, one of the five tracks corresponds to a non-player object (1 FP) in that frame. In this case, both the precision and recall are equal to 4/5*100% = 80%. Figure 5.13 shows the precision and recall using the tracking results for the handball (1) dataset. In this figure, the number of FPs and FNs (and hence precision and recall) are always equal starting from frame number 76 for team 1 and frame 84 for team 2 (after tracking has stabilized) [107].

Table 5.5: Results of player tracking for our datasets (detection results are presented in table 5.1) [107]

Dataset	Team	Precision	Recall	Avg. Prec.	Avg. Rec.
Handball (1)	T1	93.44%	93.25%	94.85%	94.72%
	T2	96.25%	96.19%		
Handball (2)	T1	94.65%	94.65%	93.93%	93.93%
	T2	93.2%	93.2%		
Basketball	T1	89.3%	89.3%	84.77%	84.68%
	T2	80.24%	80.06%		

The tracking performance is also evaluatd by using the metrics proposed in [56] and [19] for the evaluation of the Multiple Object Tracking (MOT) algorithms. The evaluation results are shown in Table 5.6 for the handball (1) dataset. The ground truth (GT) is the total number of players including the number of substitutions for a team. For team 1, GT is 9 which consists of 6 players, the goalkeeper (who wears a jersey with a similar color to his team), and 2 substitutions. For team 2, GT is 9 which consists of 6 players, and 3 substitutions (here, the goalkeeper is not tracked since his jersey differs in color as compared with his team). Mostly Tracked (MT) is the number of GT (players) trajectories which are covered by the output of the tracking system for more than 80% throughout the whole frames (5000), while Mostly Lost (ML) are GT trajectories which are covered by less than 20%. Partially Tracked (PT) equals to GT-MT-ML. The total number of times that a tracked player changes it's matched GT identity is the Identity Switch (ID Sw). Finally, Fragment (Frag) is the total number of times that a GT trajectory is interrupted in the tracking results [56] [107].

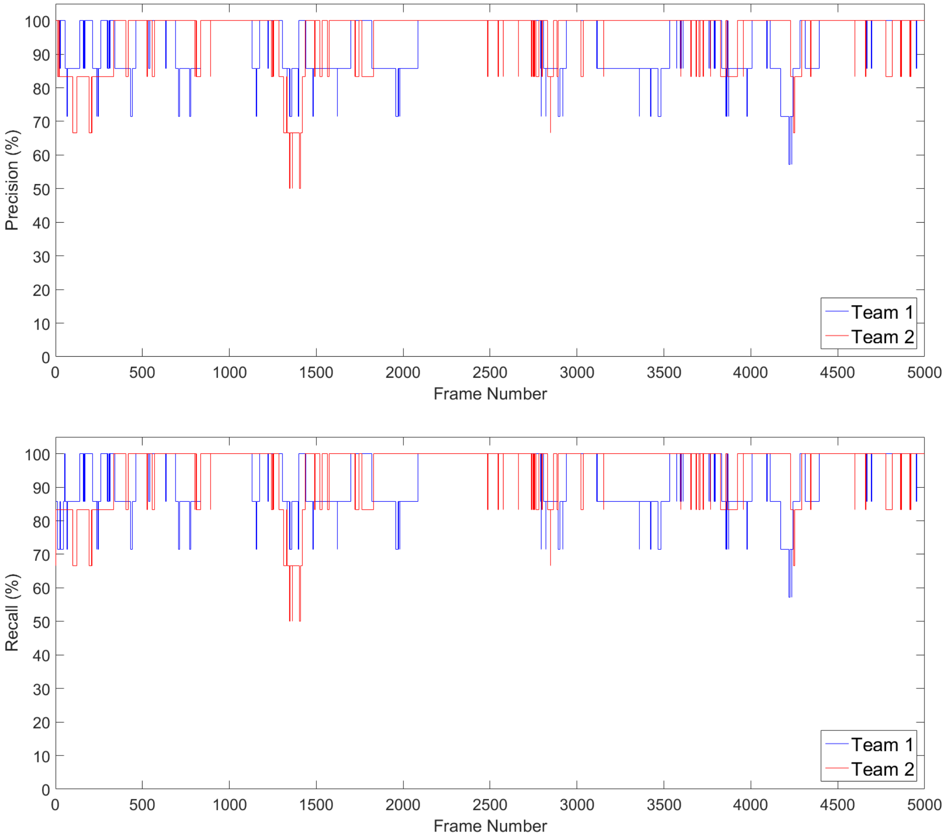


Figure 5.13: Precision and recall using the tracking results for teams 1&2 (handball (1) dataset) [107]

Table 5.6: Player tracking evaluation for the handball (1) dataset using the metrics presented in [19], [56] (The up arrow means the higher, the better; the down arrow indicates that the smaller, the better for the used metric) [107]

Team	GT	MT↑	ML↓	PT↓	ID Sw↓	Frag↓
T1	9	8	0	1	5	22
T2	9	9	0	0	9	30

As shown in Table 5.6, there are 8 players in team 1 and 9 players in team 2 (from a total of 9 players for each team) that are tracked for more than 80% (MT) by the proposed system. Whereas there is only one player which is partially tracked. Detailed information about the percentage of the covered tracked trajectory for each player is shown in Table 5.7 [107]. Additionally, a total of 5 ID switches between the players of team 1 is reported, while team 2 has 9. Moreover, there are 22 and 30 fragments (interruptions) during tracking the players of team 1 and team 2, respectively. These fragments include the ID switches as proposed by Li et al. as a more strict definition in comparison with the traditional metric, resulting in a higher number of fragments [56]. Once a tracked player is interrupted, he will be tracked again after some frames with a new ID. The number of these frames varies in each case. Despite these fragments, 14 players (out of 18) in both teams are tracked for more than 90% by the proposed system as shown in Table 5.7.

Table 5.7: Player tracking coverage for the handball (1) dataset [107]

Team	P1	P2	P3	P4	P5	P6	P7	P8	P9
T1	99.8%	83%	89.9%	97.4%	93.5%	96.3%	97.2%	90.4%	78.3%
T2	96.9%	98.2%	96.7%	83.1%	95.2%	99.2%	92.5%	99.5%	92.1%

The evaluation metrics that are used in this work are summarized as depicted in Table 5.8. Moreover, the player transfer algorithm between the two cameras is evaluated. For the handball (1) dataset, the total number of players who moved from one camera's view to the other while being tracked is counted, and it is equal to 37. The successful player transfer (using algorithms 4.2 and 4.3) are equal to 31. Some unsuccessful player transfer scenarios occurred when two players of the same team are close to each other (e.g., occluded) during the transfer between the two cameras, resulting in an ID switch or a fragmented track (i.e., a player who is not tracked for a certain number of frames). However, if player tracking is lost (i.e., fragmented) during the transfer between the two cameras, the player will be detected and hence tracked again (cf. Tables 5.6 and 5.7) with a new ID after some frames, based on his *player selector confidence* value.

An example scene for the player transfer using the handball (1) dataset is shown in Figure 5.14. In this scene, the players of both teams are moving from the right to the left direction. The tracks of all the players in team 1 are successfully transferred as depicted in Figure 5.14. Additionally, the player with the ID 5 in team 1 is going for a substitution (after a successful transfer of his track) with another player from his team. For team 2, players with ID 1 and ID 2 are occluded, and their IDs are switched. For the player with an ID 3, his track is successfully transferred as can be seen in Figure 5.14.

Table 5.8: Summery of the evaluation metrics [56] that are used in this work (The up arrow means the higher, the better; the down arrow indicates that the smaller, the better for the used metric)

Metric	Definition
GT	Number of groundtruth trajectories, i.e., number of players for one team plus the number of substitutions
TP ↑	True Positives: Number of objects that are detected or tracked by the system which have a matching ground truth (i.e., a player)
FP ↓	False Positives: Number of objects that are detected or tracked by the system which do not have a matching ground truth (i.e., a non player)
FN ↓	False Negatives: Number of objects (missed players) that are not detected or tracked by the system
Precision↑	(Frame-based) correctly matched players (TP)/total output objects (TP +FP)
Recall ↑	(Frame-based) correctly matched player (TP)/total groundtruth players (TP +FN)
MT ↑	Mostly tracked: Percentage of GT trajectories which are covered by tracker output for more than 80% in length.
ML ↓	Mostly lost: Percentage of GT trajectories which are covered by tracker output for less than 20% in length
PT ↓	Partially tracked: 1.0-MT-ML
ID Sw. ↓	ID switches: The total number of times that a tracked player changes its matched GT identity
Frag ↓	Fragments: The total of No. of times that a groundtruth trajectory is interrupted in tracking result

Finally, Figure 5.15 shows screenshots with tracking results using the three datasets. In the next section, the evaluation of the FPGA architecture is presented.

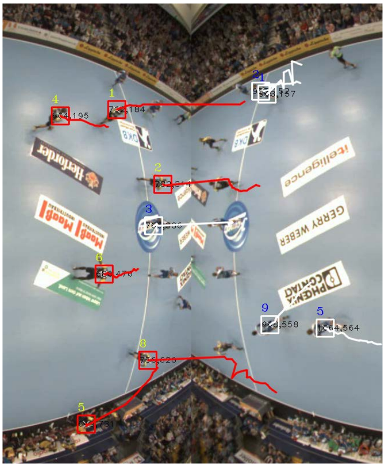


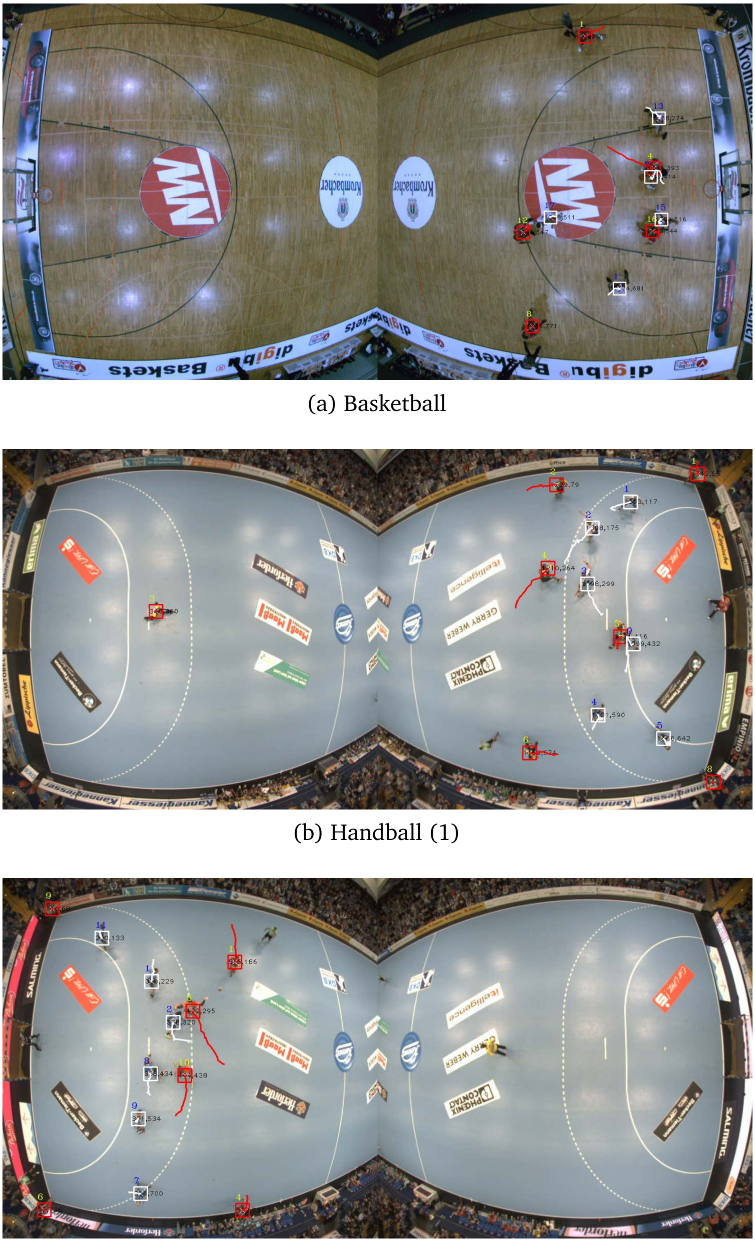
Figure 5.14: Example scene for the player tracks transfer (Handball (1) dataset)

5.7 FPGA Architecture

The FPGA architecture is evaluated in this section. This evaluation includes the multiple GigE Vision cameras support, maximum performance of the individual IP cores in the system, overall latency of the FPGA architecture, and power consumption analysis of the reconfigurable vision system. Additionally, the acceleration factor and the overall system performance are presented in this section.

5.7.1 Multiple GigE Vision Cameras

Interfacing to multiple cameras is realized using the developed MC_GigEV IP core. Additional cameras can be used if needed to support more features in the system (e.g., player identification using the digits on the players' jerseys) [107]. The MC_GigEV IP core reconstructs the video frames from multiple GigE Vision cameras as discussed in Section 4.2.1. Using one Gigabit Ethernet interface and one MC_GigEV core, different frame rates can be realized with a single or multiple cameras at different image resolutions. Figure 5.16 shows the performance of the MC_GigEV core for different resolutions and frame rates using 1, 2, 3 and 4 GigE Vision cameras with Bayer pattern



(a) Basketball

(b) Handball (1)

(c) Handball (2)

Figure 5.15: Example scenes from the three datasets showing the tracking results

output and one Byte/pixel [106].

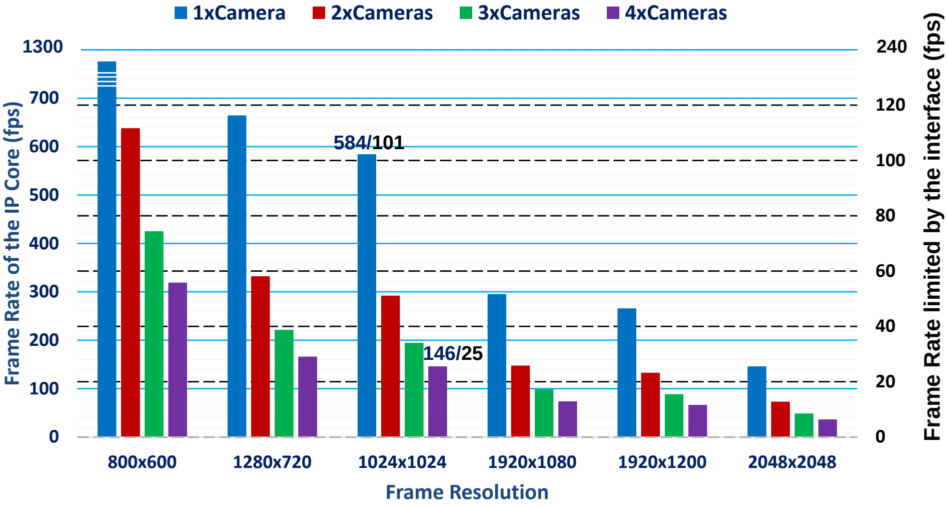


Figure 5.16: MC_GigEV Core performance for different resolutions and frame rates using configurations with 1, 2, 3 and 4 cameras [106]

The limitation for the number of cameras in the proposed system is the theoretically available bandwidth of 1 Gbps. All cameras that are connected to the same Ethernet port will share this bandwidth; hence, this will limit the number of cameras and the total amount of data received from them [106]. As shown in Figure 5.16, the MC_GigEV core can achieve a frame rate up to 584 fps for one camera with a resolution of 1024x1024 pixels. This frame rate is reduced to 101 fps due to the limitation imposed by the Gigabit interface. On the other hand, if four cameras are used, the achieved frame rate by the MC_GigEV core is 146 fps, and it is limited to 25 fps using the 1 Gbps bandwidth.

Table 5.9 shows a comparison of the MC_GigEV IP core to a commercial single camera GigE Vision IP core, the GigEV core [81], developed by a Xilinx partner company. The comparison is based on the used resources and maximum clock frequency. The GigEV core resources are given (based on logic synthesis) for the Xilinx Artix-7 FPGA. For better comparison, the MC_GigEV core logic resources are generated for this Artix-7 FPGA. In this comparison, both cores can handle only one camera per core instance. As shown in Table 5.9, the proposed MC_GigEV core requires significantly less FPGA resources than the GigEV core, while achieving almost the same maximum clock frequency. In the GigEV core, the complete control part of the GigE Vision protocol is handled by an embedded CPU and the stream channel is implemented in hardware to achieve maximum throughput. Additionally, the core supports bidirectional streaming, i.e., it can be used for receiving GigE Vision packets from a transmitting device as well as for sending video data via the GigE Vision protocol [106].

Table 5.9: Comparison with other GigE Vision IP core [106]

	GigEVCore1.2	MC_GigEV IP Core	
FPGA	Artix-7	Virtex-4	Artix-7
REGs	3441	1317	1191
LUT	3800	1618	1237
BRAM	9	13	7
FMAX	172 MHz	155 MHz	170 MHz

Using the developed MC_GigEV core, memory storage space is saved by extracting the raw video data directly from the GVSP packets when they are received, as compared to storing the complete packets in memory first and then extracting the video data. Furthermore, the MC_GigEV core supports the video data extraction from the standard Ethernet as well as Jumbo packets. The efficiency of the GVSP protocol can be calculated for the standard Ethernet and Jumbo packets using Equation 5.5. A standard Ethernet packet has a maximum packet size of 1500 bytes. Therefore, the maximum size of the GVSP packet is 1514 bytes including the Ethernet header. For the jumbo packets, the GVSP maximum packet size is 9014 bytes. Since the total overhead is 50 bytes resulted from the packet's headers (cf. Figure 2.11), the payload size is 1464 and 8964 bytes for the standard and jumbo packets, respectively. Therefore, the GVSP protocol efficiency for jumbo packets is higher than standard Ethernet packets as shown in Equations 5.6 and 5.7.

$$\text{Protocol efficiency} = \frac{\text{Payload size}}{\text{Packet size}} \tag{5.5}$$

$$\text{Protocol efficiency (standard packet)} = \frac{1464}{1514} = 96.7\% \tag{5.6}$$

$$\text{Protocol efficiency (jumbo packet)} = \frac{8964}{9014} = 99.45\% \tag{5.7}$$

As stated in Chapter 4, two GigE Vision cameras are used for the player tracking system. Each camera is operating with a maximum resolution of 1392x1040 pixels at

30 fps with 8 bits/pixel. As a result, the total Ethernet bandwidth (BW) that is needed for streaming the GigE Vision stream packets from the two cameras can be calculated using Equation 5.8 [107].

$$\begin{aligned} \text{BW} = \text{Camera Resolution} \times \text{Frame Rate} \times \text{Nr. of bits/pixel} \times \text{Nr. of Cameras} \\ + \text{Packets Overhead} \end{aligned} \tag{5.8}$$

$$\text{BW} = 1392 \times 1040 \times 30 \times 8 \times 2 + 23.74 \text{ Mbps}$$

$$\text{BW} = 718.62 \text{ Mbps}$$

The packet overhead is the total number of bytes of the packet's headers for the GigE Vision streaming protocol. As can be seen from Equation 5.8, the needed bandwidth is 718.62 Mbps, which is less than the maximum 1 Gbps bandwidth of the Gigabit Ethernet [107]. Therefore, these two GigE Vision cameras can be operated with their maximum resolution and frame rate, sharing the same one Gigabit Ethernet bandwidth.

5.7.2 Performance and Throughput

The maximum performance (frame rate) of each IP core of the video processing modules is calculated using Equation 5.9. As can be seen, the frame rate depends on the maximum frequency (Fmax) of the IP cores and the input frame resolution to the respective core. Using a Xilinx Virtex-4 FPGA, the maximum frame rates of the individual IP cores in the proposed system are shown in Figure 5.17 for the handball and basketball datasets, while Figure 5.18 shows the performance for a Xilinx Virtex-7 FPGA.

$$\text{Maximum Frame Rate} = \frac{\text{Fmax}}{\text{Frame Resolution}} \tag{5.9}$$

The input frame resolution is 1392x1040 pixels from each of the two cameras. However, the two frames from the two cameras are cropped and merged, forming a larger frame as depicted in Figures 4.2 and 4.6. The frame resolution after cropping and merging is 1664x800 pixels for the handball dataset. While for the basketball dataset, the frame size is 1792x900 pixels. Since the merged frame resolution of the basketball dataset is larger than the handball frame, the respective frame rates are lower as shown in Figure 5.17. The maximum achieved frame rate of the proposed architecture using a Xilinx Virtex-4 FPGA is 96.7 fps and it can be increased to 136.4 fps using Virtex-7 FPGA. However, this frame rate is limited by the Gigabit Ethernet bandwidth. If one Gigabit Ethernet interface is shared for the two cameras, the maximum frame rate is 41.7 fps (42.9 fps using jumbo packets). This frame rate can be increased to 83.4 fps

(86.3 fps using jumbo packets) if two Gigabit Ethernet interfaces are used, one interface for each GigE Vision camera allowing for a total bandwidth of 2 Gbps [107].

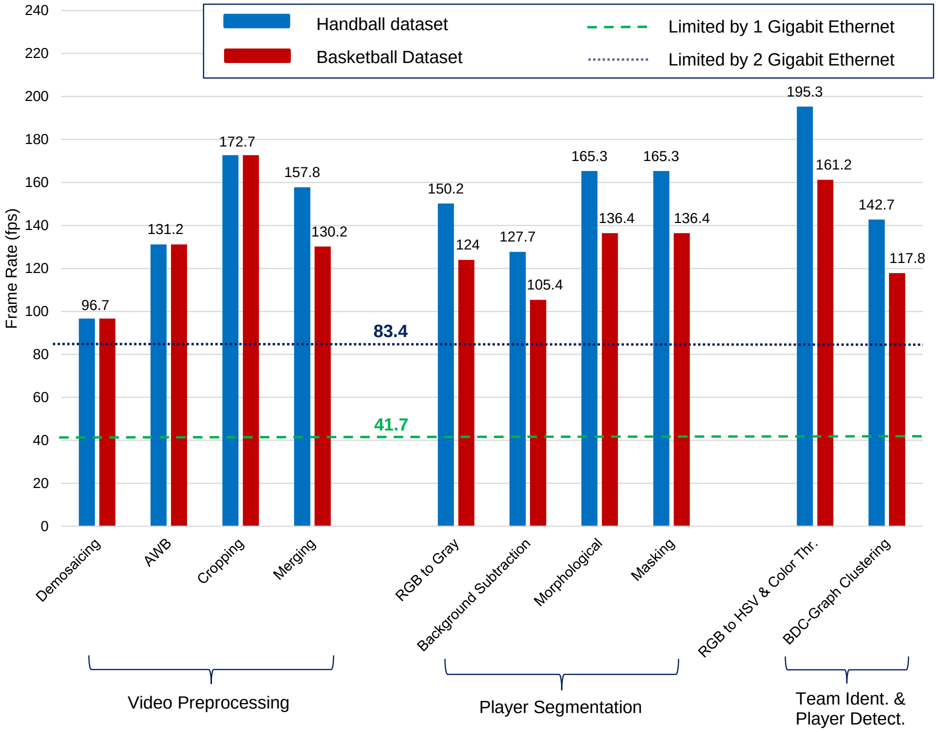


Figure 5.17: Performance of the FPGA architecture with a Virtex-4 FPGA using two cameras, each has a frame resolution of 1392x1040 pixels [107]

Throughput can be defined as the number of pixels at a certain frame resolution that can be processed per second. The minimum required throughput can be calculated using Equation 5.10. Here, the required frame rate for calculation is 30 fps, corresponding to the maximum frame rate of the utilized cameras, and it is considered as a sufficient frame rate for this player tracking application using the camera setup shown in Figure 4.1 [66] [65]. Therefore, the required throughput by the IP cores (e.g., demosaicing) for processing the video data from one GigE Vision camera is 43.43 MPixels/second as shown in Table 5.10. After cropping and merging the video streams from both cameras, the required throughput values by the IP cores (e.g., background subtraction) are 39.94 and 48.38 MPixels/second for the handball and basketball datasets, respectively. If a standard width resolution (e.g., 1920) is used (this is required by the DVI display controller for intermediate results display (useful for debugging)), the throughput values are increased to 46.08 and 51.84 MPixels/second for the handball and basketball datasets, respectively.

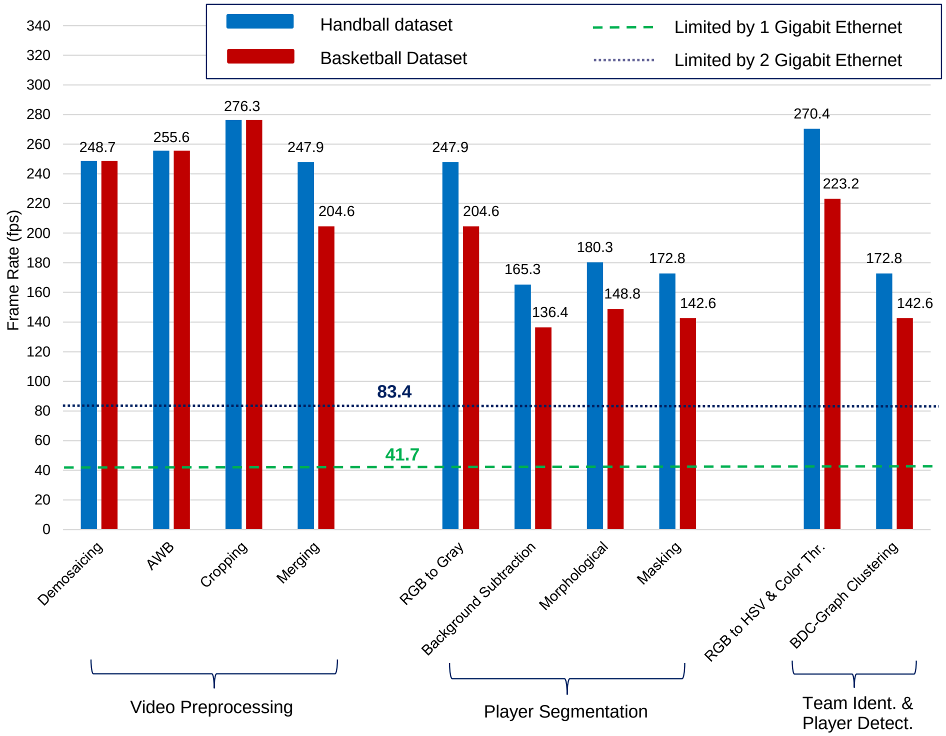


Figure 5.18: Performance of the FPGA architecture with a Virtex-7 FPGA using two cameras, each has a frame resolution of 1392x1040 pixels

$$\text{Throughput} = \text{Frame Resolution} \times \text{Required Frame Rate} \tag{5.10}$$

Table 5.10: Throughput requirements (MPixels/second)

GigE Vision Camera (x1) (1392x1040)	Handball dataset (1664x800)	Basketball dataset (1792x900)	Handball dataset (Disp.) (1920x800)	Basketball dataset (Disp.) (1920x800)
43.43	39.94	48.38	46.08	51.84

In the FPGA architecture, the video processing IP cores produce one pixel every clock cycle. Therefore and based on the achieved maximum frequency of these cores, the maximum throughput for each IP core using a Xilinx Virtex-4 FPGA is shown in Figure 5.19. As can be seen, all the processing cores achieve significantly higher throughput than the required values (cf. Table 5.10). In particular, the demosaicing, AWB, and cropping cores achieve more than 43.43 MPixels/seconds (the required throughput from one camera), while the other cores have a throughput higher than 51.84 MPixels/second (for the merged video stream). The BDC-graph cluster IP core is not shown in Figure 5.19, since it does not output a processed pixel in every clock but it produces centroids of the detected objects after a frame is received.

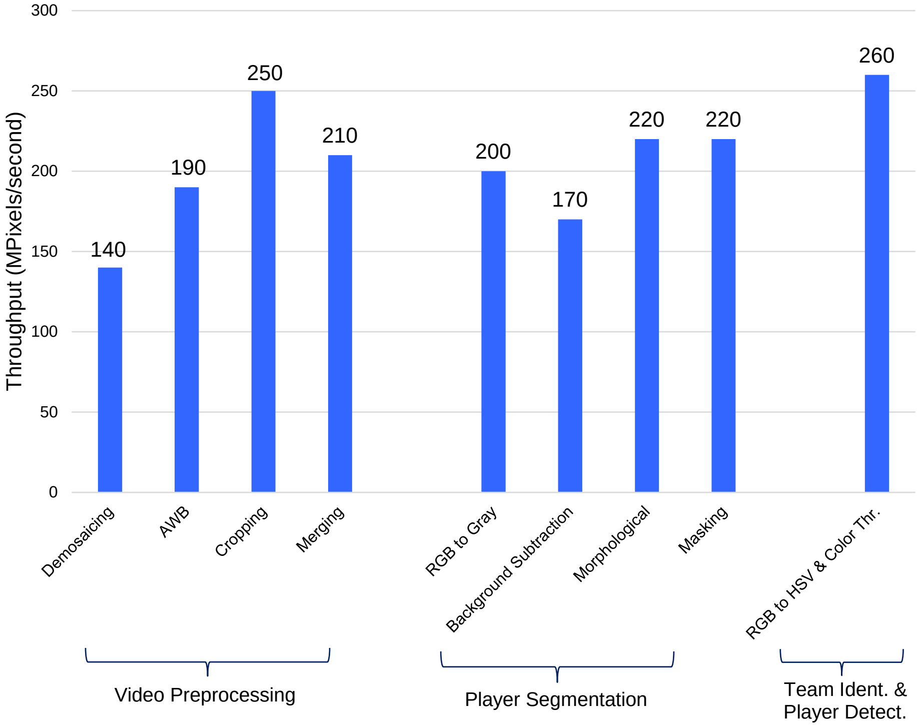


Figure 5.19: Maximum throughput (MPixels/second) of the IP cores for a Virtex-4 FPGA

Figure 5.20 shows the maximum throughput of the IP cores in MByte/second for a Xilinx Virtex-4 FPGA. As can be seen, several cores (e.g., demosaicing, AWB, etc.) have a high number of MBytes/second, since their outputs consist of 4 Bytes for each pixel.

While for other IP cores (e.g., RGB to Gray, etc.), the resulted pixel consists of one Byte.

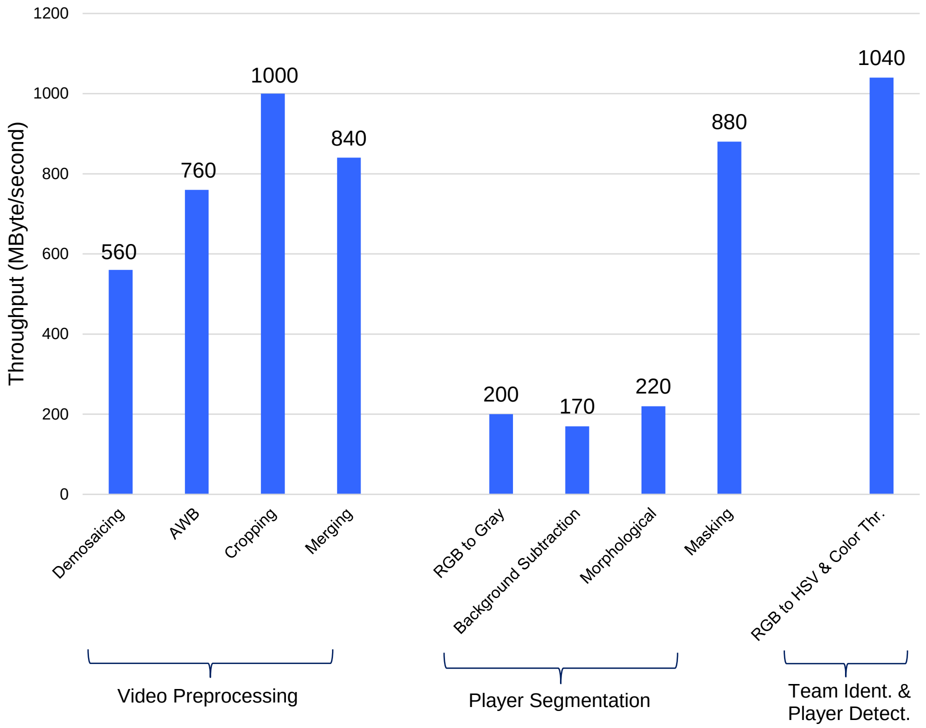


Figure 5.20: Maximum Throughput (MByte/second) of the IP cores for a Virtex-4 FPGA

5.7.3 Acceleration Factor and Overall System Performance

For comparing the performance (in terms of processing time and frame rate) of the proposed FPGA architecture with a pure software-based system, a software implemented in C++ using the OpenCV library has been realized. The execution time for each module is measured as shown in Table 5.11. These measurements are based on the handball (1) dataset using the software implementation, executed on a host-PC equipped with an Intel i7 CPU (870 at 2.93 GHz). 5000 frames are used to calculate the average time for these measurements. For comparison, the performance of the hardware implementation using a Xilinx Virtex-4 FPGA is also included in the table. As can be seen, the FPGA implementation achieves a speedup of 15.5 times in comparison to the software implementation on the PC (96.7 fps vs. 6.23 fps) [107].

Table 5.11: Performance comparison with a software implementation [107]

SW - Intel i7 CPU (2.93 GHz)	Average Time	Frame Rate	Throughput
Video Preprocessing	69.35 ms	14.42 fps	41.75 MPixel/s
Player Segmentation	17.3 ms	57.8 fps	76.94 MPixel/s
Team Ident. & Player Detect.	73.99 ms	13.52 fps	18 MPixel/s
Total SW Implementation	160.64 ms	6.23 fps	18.04 MPixel/s
HW - Virtex-4 FPGA	10.34 ms	96.7 fps	279.98 MPixel/s

Furthermore, the performance of the proposed reconfigurable system is detailed in Table 5.12. It consists of: the processing time for the FPGA modules, the time required for reading the FPGA output results by the host-PC, and the required processing time for player tracking on the CPU. In the host-PC, the average time was measured using 5000 frames of the handball (1) dataset. As can be seen in Table 5.12, the total average processing time on the host-PC is 0.103+2.43 =2.533 ms. As a result, the overall performance of the reconfigurable system is 77.6 fps for an input frame resolution of 1392x1040 pixels from each of the two GigE Vision cameras [107].

Table 5.12: The overall system performance [107]

Operation	Technology	Average Time	Frame Rate
Video Acquisition, Video Preprocessing, Player Segmentation, Team Identification & Player Detection	Virtex-4 FPGA	10.34 ms	96.7 fps
Reading detection results from the FPGA	Intel i7-870 CPU (4 at 2.93 GHz)	0.103 ms	N/A
Player Tracking	Intel i7-870 CPU (4 at 2.93 GHz)	2.43 ms	N/A
Overall Performance	FPGA & CPU	12.873 ms	77.6 fps

5.7.4 Overall Latency

The latency of an operation is the time between when data is first input to this operation, and the corresponding output is available [15]. The total latency of the FPGA architecture (using video files as the input source) is measured as shown in Figure 5.21. First, the time of sending one video frame (with a resolution of 1920x800 pixels, corresponding to 1.536 MB) from the host-PC to the FPGA through the PCI-X interface is measured, and it is equal to 17.67 ms. However, the FPGA IP cores start processing the input data as soon as the first pixel of a frame is received. Therefore, this video frame transmission and the FPGA processing are pipelined as depicted in Figure 5.21.

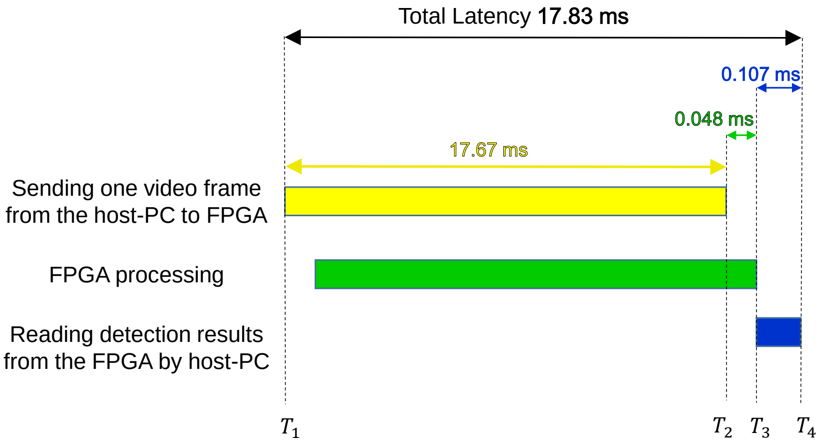

Total Latency 17.83 ms
0.107 ms
0.048 ms
17.67 ms
Sending one video frame from the host-PC to FPGA
FPGA processing
Reading detection results from the FPGA by host-PC
T_1
T_2
T_3
T_4

Figure 5.21: Total latency of the FPGA architecture (Video file input)

The required time to read the detection results from the FPGA by the host-PC (i.e., T_3 to T_4 as shown in Figure 5.21) is measured, and it is equal to 0.107 ms. This time corresponds to reading the detected player positions (including FPs) for both teams. Furthermore, the total latency is measured, starting from sending the video frame to the FPGA (T_1) until all the detection results are received (T_4). As shown in Figure 5.21, this total latency is equal to 17.83 ms. Therefore, the period between T_2 and T_3, which corresponds to a part of the FPGA processing latency as depicted in Figure 5.21, can be calculated and it is equal to 17.83 - 17.67 - 0.107 = 0.048 ms. This small latency is expected since for the morphological operation (dilation) IP core as an example, a latency of one row (corresponding to 19.2 μs for a 100 MHz clock and a row width of 1920 pixels) is required. Additionally, the BDC-based graph clustering core requires additional clock cycles to transfer the computed centroids from the intermediate registers to the output FIFO of the core after the frame is processed. For the example mentioned in Section 5.5, a latency of 45 clock cycles (corresponding to 0.45 μs for a 100 MHz clock input) is required for the centroids to be ready before the transfer to the host-PC can be started. The measured time values (shown in Figure 5.21) are obtained by averaging the corresponding time measurements for 5000 frames using the handball (1) dataset.

The total latency of the FPGA architecture is shown in Figure 5.22 using the GigE Vision cameras as the input video source. As depicted in Subsection 4.2.1, the transmitted video data from the cameras are captured by the FPGA using the MC_GigE Vision IP core. The time for streaming out the pixels of one video frame using this core can be calculated using Equation 5.11. For the used cameras in this work, the image width and height are 1392 and 1040, respectively, while the operating clock frequency for the MC_GigE Vision IP core is 100 MHz. Therefore, the time to stream out a video frame from one camera is 14.48 ms. This amount of time remains the same for streaming out video frames from two cameras since the MC_GigE Vision core uses two AXI4-Stream outputs to stream video frames simultaneously from the two cameras.

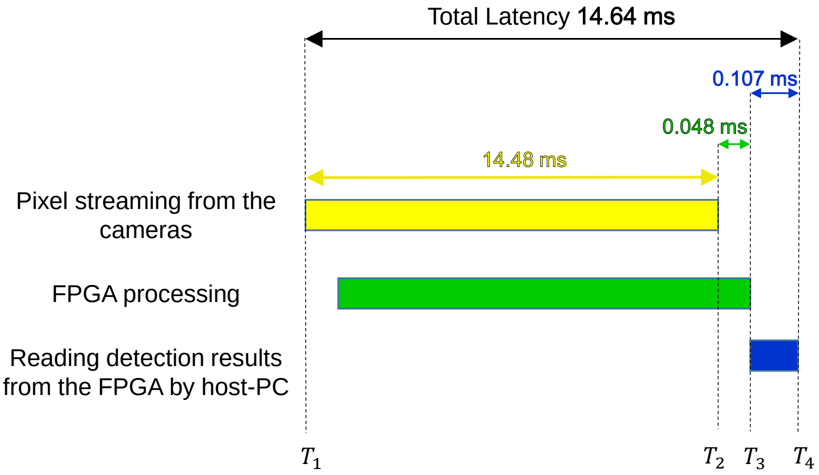

Total Latency 14.64 ms
0.107 ms
0.048 ms
14.48 ms
Pixel streaming from the cameras
FPGA processing
Reading detection results from the FPGA by host-PC
T_1
T_2
T_3
T_4

Figure 5.22: Total latency of the FPGA architecture (Camera input)

$$\text{Video frame time} = \text{Image Width} \times \text{Image Height} \times T_{clk} \tag{5.11}$$

5.7.5 Power Consumption

For the proposed player tracking system, power consumption measurements are performed on three levels as shown in Figure 5.23. The first level targets the power consumed by the Daughter Board (DB), including the Virtex-4 FPGA (DB-V4), while the second one is to measure the total power used by the RAPTOR board including the DB-V4, DB-display, and Gigabit Ethernet boards. The third level is the complete reconfigurable vision system implementation on a host-PC (here, an open frame PC equipped with an Intel Xeon CPU E3-1226 with 4 cores at 3.30 GHz is used), including the RAPTOR board and its components.

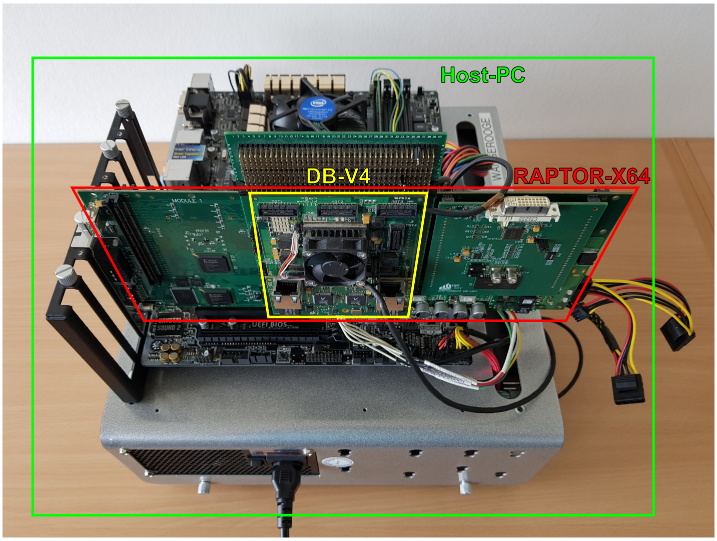


Figure 5.23: Power consumption measurements

Table 5.13 shows the power consumption of the DB-V4 board. This daughter board includes the Virtex-4 FPGA, fan, external memory (DDR2-Synchronous Dynamic RAM (SDRAM)), and DC to DC converters. In this DB-V4, the used voltages are 5, 3.3, 2.5, and 1.8 V. Here, the power measurements are performed when the FPGA is in the idle state (i.e., without bitstream), and when the FPGA is programmed. In both cases, the measurements are performed for two configurations. The first one is excluding the DDR2-SDRAM and fan (configuration (A)). While in configuration (B), the measurements are performed with the fan and the external memory installed on the DB-V4 board. Finally, the consumed power is measured when the FPGA is in the active state, i.e., during the processing of the incoming input video stream using the Handball (1) dataset. As can be seen in Table 5.13, the total amount of the consumed power by the

DB-V4 board is 8.41 Watt with the FPGA being programmed and all the used clocks are activated. This amount of power is required to have a functioning FPGA-based board including the required components (e.g., external memory). This value is increased to 8.94 Watt during processing, since the signals in the design toggle based on the input data as well as there are memory access (reading and writing from/to the external memory) which consume additional power. Furthermore, the amount of the power consumed by the FPGA implementation after the bitstream is downloaded to the FPGA can be extracted from this table, and it is equal to 8.41 - 2.14 = 6.27 Watt which is used by the various components in the system including logic, IOs, BRAMS, PowerPC, DSPs, etc.

Table 5.13: Power consumption of the DB-V4 board (including the Virtex-4 FPGA)

	Idle FPGA (w/o Bitstream)		Programmed FPGA (with Bitstream)		Active
	Conf. (A)	Conf. (B)	Conf. (A)	Conf. (B)	(Processing)
Power (Watt)	1.84	2.14	7.32	8.41	8.94

Since the clock frequencies of the individual IP cores influence the consumed power by the FPGA design, the used clock frequencies in these cores (while measuring the power consumption) are reported in Table 5.14. As can be seen, the video processing IP cores are operated with a 100 MHz clock, resulting in a throughput of 100 MPixels/second. This clock frequency is adequate to process the video streams from the used cameras in this work with their maximum frame rate (i.e., 30 fps).

Table 5.14: Operating clock frequencies used by the IP cores in the FPGA architecture

Clock Frequency	Video Proc. IP Cores	PPC405	TEMAC	MPMC	AXI4-S to NPI	LB-Slave to AXI4-S/NPI
25 MHz						X
100 MHz	X				X	X
125 MHz			X			
200 MHz				X	X	
300 MHz		X				

The power consumption of the RAPTOR-X64 board is shown in Table 5.15. The measured input voltage to this board is 12.31 V. Firstly, the consumed power by the main board is measured without any additional boards. Secondly, the consumed power by the RAPTOR-X64 board including the DB-V4, Gigabit Ethernet, and display boards are reported in this table without and with the bitstream being downloaded to the FPGA. Finally, the used power is measured when the complete RAPTOR-X64 is running and processing the video stream from the handball (1) dataset. As shown in Table 5.15, the total power consumed by the RAPTOR system including its additional boards is 12.53 Watt (without the bitstream). This power is increased to 19.94 Watt when processing the input data for the player tracking system.

Table 5.15: Power consumption of the RAPTOR-X64 board

	Main Board Only	With Daughter Boards	With Daughter Boards & Bitstream	Active (Processing)
Power (Watt)	9.84	12.53	19.06	19.94

The last level of the performed power measurement is for the complete host-PC. The results of these measurements are shown in Table 5.16. The used power by the host-PC (excluding and including the RAPTOR board) is reported in this table. Additionally, the consumed power is measured when the FPGA is programmed, and when the complete reconfigurable player tracking system is running (active). As can be seen in Table 5.16, the total consumed power by the host-PC increases from 46.5 Watt to 89 Watt while processing the video data, realizing the player tracking system as shown in Figure 5.2. This power increase is due to the additional power that is required when the host-PC sends the video frames to the RAPTOR board and reads its output. Furthermore, the CPU consumes additional power for post-processing, and displaying the final tracking results.

Table 5.16: Power consumption of the host-PC

	Idle - Without RAPTOR Board	Idle - With RAPTOR Board	With RAPTOR Board & Bitstream	Active (Processing)
Power (Watt)	27	41	46.5	89

Figure 5.24 summarizes the power consumption of the DB-V4, RAPTOR system, and host-PC for the proposed player tracking system based on the previously mentioned

measurements. The values in the idle state are measured when the bitstream is not downloaded to the FPGA. The measurements in the active state represent the consumed power when the system is running and processing the input data to track the handball players. Based on these results, the performance per Watt (fps/Watt) is calculated for the DB-V4, RAPTOR, and host-PC while they are in the active state and processing the handball video with 30 fps. These performance per Watt values are shown in Figure 5.25 (the higher, the better). Additionally, the proposed player tracking system is implemented in software (i.e., without the FPGA support) on the host-PC, and the performance per Watt is reported as shown using the yellow line in Figure 5.25 (here, the RAPTOR board is removed during the power measurement). As can be seen, the proposed reconfigurable system achieves a better performance per Watt (0.337 fps/Watt) as compared with the software implementation (0.139 fps/Watt) (i.e., 2.4 times higher performance per Watt is realized using the reconfigurable system), since the achieved frame rate of the software-based system is low without the FPGA acceleration. For the reconfigurable vision system, the power measurements are performed while the system processes the recorded video data (handball (1) dataset) at 30 fps. For higher frame rates (e.g., up to 77.6 fps in this system), an increase in the performance per Watt can be achieved using the proposed reconfigurable system.

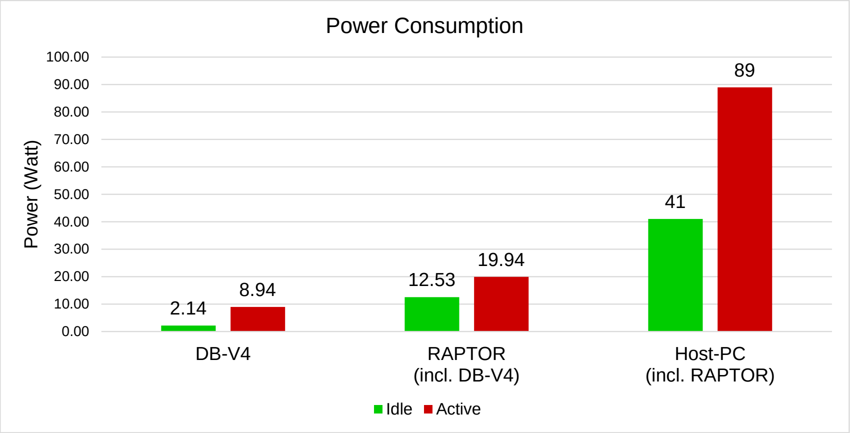


Figure 5.24: Power consumption of the proposed reconfigurable system for player tracking using the handball (1) dataset

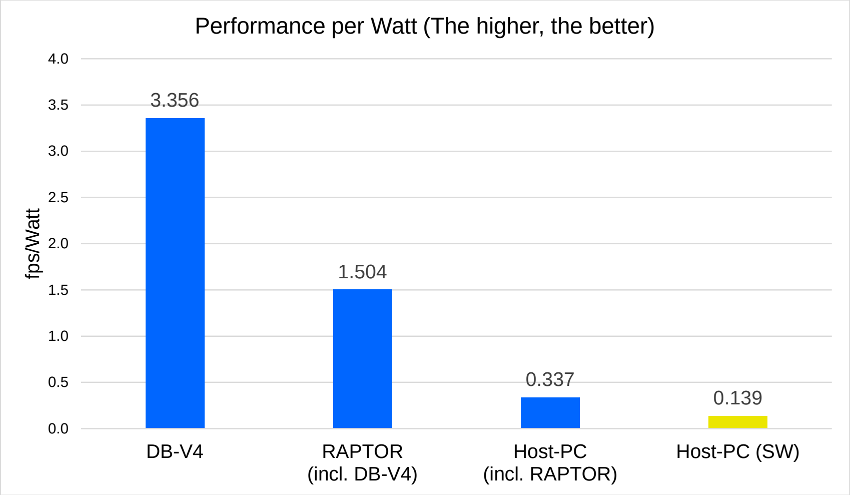


Figure 5.25: Performance per Watt of the proposed reconfigurable system and a comparison with a software implementation of the player tracking system

5.8 Comparison with Existing Systems

For evaluation of the results, the proposed system is compared with existing work discussed in the related work section. The chosen parameters for comparison in Table 5.17 include the targeted sports type, the utilized method for detecting and tracking the players, and the source type of the input video. This source can be a broadcast video stream or a dedicated camera. Furthermore, the resolution and the frame rate of the input video source, as well as the processing architectures on which the systems have been implemented, the achieved frame rate and the required processing time are depicted in this table together with the reported detection and tracking results [107].

As can be seen in Table 5.17, there are several means to categorize these systems. One approach for categorization is using the type of the input video source. For broadcast video-based systems, the amount of pixel data that needs to be processed to track the players is usually less than those in the dedicated camera-based systems. However, the sports courts are not completely covered in every frame of the broadcast video streams. Additionally, the dimensions of the courts need to be extracted in each frame, since these systems usually use moving cameras with a zoom-in and out feature. This extraction process imposes additional processing. On the other hand, systems based on dedicated cameras cover the whole court. As shown in Table 5.17, one to six stationary cameras are used in different systems. Furthermore, multiple cameras are usually required to achieve better tracking results [66] [78] [4] [85]. Consequently, the total number of pixel data from these cameras that needs to be processed is increased, slowing down the overall system. Therefore, some systems use GPUs to accelerate the tracking process as in [78], while FPGA technology is used in this work [107].

Using the camera setup shown in this work, players have different shapes (structures) based on their positions on the sports court as can be seen in Figure 2.4. Compared to more generic object detection techniques based on structure information (e.g., Deformable Part Model used to detect the basketball players by Lu et al. [58]) with high computational requirements, the proposed approach for player detection based on the color of the players' jerseys is simple, yet effective. Additionally, the presented approach in this thesis takes advantage of the specific environment of an indoor sports hall combined with stationary cameras, by which the court has fixed dimensions in the captured video frames. Furthermore, this approach does not require training using annotated datasets, and only the colors of the jerseys should be given in advance, in contrast to many existing systems that require such training datasets (e.g., Lu et al. [58] and Acuna [1]) [107].

As can be seen in Table 5.17, some existing systems do not support full coverage of the sports halls (e.g., broadcast video systems [41], [27], [58], and [1]), while other systems do not track all the players in the sport games (e.g., player detection only on the left half of a sports court in Parisot et at. [73], and player tracking for one volleyball team only in Li et al. [55]). For player tracking, the achieved precision and recall in some systems are around 90% (e.g., Hu et al. [41], Chen et al. [41], Acuna et al. [1], and Morimitsu et al. [67]). However, a precision of 98% is achieved using the system presented by Lu et al. [58], while the achieved recall is only 82%. Furthermore, some systems use many cameras (e.g., up to 5 cameras in Alahi et al. [4], and 6 cameras in the STATS SportVU system [85]). Other systems do not achieve high frame rate which is required for realizing real-time player tracking (e.g., 1 fps in Lu et al. [58], 19.6 fps in Monier et al. [65], 16.02 fps in Santiago et al. [78], and 4 fps in Morimitsu et al. [67]). As compared with the state-of-the-art work, the proposed system uses dedicated cameras, covering the whole sports court using only two cameras. It tracks all the players of both teams. For player tracking, the presented system in this work achieves high precision and recall of 94.85% and 94.72%, respectively. Additionally, the system supports a high frame rate up to 77.6 fps using FPGA technology, realizing an online and real-time player tracking system.

Table 5.17: Comparison with existing systems (Pr.= Precision, Re.= Recall, D= Detection, T= Tracking, Fr.= Frame, Res.= Resolution, P.= Processing, - = Not mentioned) [107]

System	Sports/ Method Type	Video Src./ Res.(pixels)/ Fr. Rate	Processing Archit- ecture	Achieved Fr. Rate P. Time	Results (Best Reported Values)
Hu et al. [41]	Basketball CamShift	Broadcast 720x480 29.97 fps	-	-	Pr.(T): 91.38% Re.(T): 91.34%
Chen et al. [27]	Basketball k-means clustering+ Kalman F.	Broadcast 640x352 29.97 fps	-	-	Pr.(T): 89.71% Re.(T): 89.20%
Lu et al. [58][59]	Basketball Deformable Part Model (DPM)	Broadcast 1280x720 -	CPU 2.8 GHz + GPU	0.5-1 fps 1-2 sec	Pr.(D): 97% Re.(D): 74% Pr.(T): 98% Re.(T): 82%
Acuna [1]	Basketball Deep Neural Net	Broadcast NCAA Dataset	-	-	Avg. Pr.: 89%
Monier et al. [66][65]	Handball Basketball Template Matching[1] Particle F.[2]	2 Cameras 1392x1040 30 fps	CPU i7-950 4 cores Multicore Implement.	19.6 fps[1] 51 ms[1] 9.7 fps[2] 103 ms[2]	Avg. Correction Rate: 0.00677 Cor./Fr./Pl.[1] Error Rate: 5.08 error/min.[2]
Santiago et al. [78][79]	Handball Basketball BG Sub. +Fuzzy Logic	2 Cameras 1024x768 30 fps	CPU i7-640M 4@2.8 GHz + GPU NVidia NVS2100M 16@0.5 GHz	16.02 fps 62.5 ms	Average Tracking Rate: 98.79% Offline T.
Alahi et al. [4]	Basketball BG Subt. + Sparsity constrained occupancy map	5 Cameras APIDIS 1600x1200 downscaled to 320x240 22 fps	-	-	2 Cameras Pr.(D): 72% Re.(D): 76% 5 Cameras Pr.(D): 83% Re.(D): 74%

Table 5.17: Comparison with existing systems (Pr.= Precision, Re.= Recall, D= Detection, T= Tracking, Fr.= Frame, Res.= Resolution, P.= Processing, - = Not mentioned) [107] - (*Continued*)

System	Sports/ Method Type	Video Src./ Res.(pixels)/ Fr. Rate	Processing Architecture	Achieved Fr. Rate P. Time	Results (Best Reported Values)
H. C. de Padua et al. [72]	Futsal BG Sub.+ Blob A.+ Particle F.	1 Camera 752x480 30 fps	CPU Intel i7 8 cores 3.4 GHz	40 fps 25 ms	Pr.(D): 90.9% Re.(D): 76.4% Pr.(T): 89.3% Re.(T): 80%
Parisot et al. [73]	Basketball FG D.+ Bayesian Classifier	1 Camera 1600x1200 30 fps	i7-4790 CPU 4@3.60 GHz Hyper-threaded	30 fps	Re(D): 90% FPs rejection rate: 80% Left court's half
Morimitsu et al. [67]	TableTennis Badminton Volleyball Online Graph +Particle F.	1 Camera up to 1280x720 30 fps	CPU Intel i5	1.5-4 fps 250 ms - 666 ms	Re.(T): 89.3% FPs rate(T): 9.6%(↓) ID Sw: 85
Butt et al. [25]	Generic (Pedestrian) Lagrangian Relaxation	1 Camera 640x480 14 - 25 fps	CPU (MATLAB)	1.43 sec- 200 Fr. 59 sec- 1000 Fr.	Mismatches Nr: 14 - 23 Detections Nr: 819 - 1514
STATS SportVU (NBA) [85]	Basketball -	6 Cameras 25 fps	Commercial Product -	Real-Time Tracking	-
TRACAB [29]	Indoor/ Outdoor Sports	2 Cam units Super-HD 25 fps	Commercial Product -	Real-Time Tracking	-
Li et al. [55]	Volleyball BG Subt. +Template Matching	1 Camera 800x600 30 fps	FPGA (Spartan-6) LUTs: 14571 (53.4%)	100 fps 10 ms	Recognition Accuracy: 72.2% Only 1 Team T.
The Proposed System	Handball Basketball BG Sub. + Graph Clustering	2 Cameras 1392x1040 30 fps	FPGA (Virtex-4) LUTs: 51875 +CPU i7-870 4@2.93 GHz	77.6 fps 12.87 ms	Pr.(D): 84.02% Re.(D): 96.6% Pr.(T): 94.85% Re.(T): 94.72%

5.9 Summary

The proposed vision-based reconfigurable system for player tracking has been evaluated in this chapter. First, the used hardware environment and the implemented multithreaded program to realize the proposed system is presented. Then, the datasets that are used for evaluation are shown. After that, player detection is analyzed and evaluated using standard metrics, followed by further analysis for different player occlusion scenarios. For player detection, the experimental results show that the achieved average precision and recall are up to 84.02% and 96.6%, respectively. Additionally, the hardware implementation of the player detection module is verified in this chapter. After player detection performance analysis is presented, the evaluation of the player tracking is depicted. The achieved average precision and recall for player tracking are up to 94.85% and 94.72% respectively.

The evaluation of the FPGA architecture is presented in this chapter, including the multiple camera support, performance and throughput of the individual IP cores, and overall latency of the architecture. Furthermore, the acceleration factor that is gained using the FPGA implementation, and the overall system performance are presented. Using the proposed FPGA architecture, an acceleration factor of 15.5 is achieved compared to an OpenCV-based software implementation on a host-PC. The proposed reconfigurable system achieves a maximum frame rate of 77.6 fps using two GigE Vision cameras with a resolution of 1392x1040 pixels each. Moreover, power consumption measurements and analysis as well as performance per Watt are presented on different levels of the proposed system. The results show that the proposed reconfigurable system achieves 0.337 fps/Watt, while an equivalent software implementation (without FPGA support) achieves 0.139 fps/Watt (i.e., 2.4 times higher performance per Watt is realized using the reconfigurable system). Finally, this chapter ends with a comprehensive comparison of the proposed system in this thesis with the other systems that are presented in the related work section in Chapter 2.

6 Conclusions and Future Work

This chapter concludes the work presented in this thesis, and proposes some suggestions for the future development.

6.1 Conclusions

In this thesis, a complete reconfigurable vision processing system for automatic and online player tracking in indoor sports is presented. The proposed system can process live video data streams from multiple cameras as well as offline video data, targeting player tracking for basketball and handball games. Two GigE Vision cameras are used with a resolution of 1392x1040 pixels and a frame rate of 30 fps, covering the complete sports court. Player tracking systems have high computational demands resulting from the video processing algorithms as well as from the huge amount of video data to be processed from multiple cameras. Therefore, FPGA technology is used to handle the compute-intensive vision processing tasks, achieving real-time player tracking, while the less compute-intensive operations are performed on a CPU. Dedicated hardware modules have been implemented for video acquisition, video preprocessing, player segmentation, and team identification & player detection, targeting Xilinx Virtex-4 to 7 Series FPGAs [107].

In the proposed system, the two teams are identified and the positions of the players are detected based on the colors of the players' jerseys. More precisely, player detection is achieved using background subtraction, color thresholding, and graph clustering techniques. Furthermore, the tracking-by-detection approach is used to achieve player tracking. Moreover, player transfer between the two camera views is implemented, realizing player tracking on the complete sports court.

Player detection and tracking are evaluated using basketball and handball datasets. High precision and recall for player tracking are achieved compared with existing systems. For the proposed system, the achieved average precision and recall for player detection are up to 84.02% and 96.6%, respectively. For player tracking, the maximum achieved average precision and recall are 94.85% and 94.72%, respectively.

In the proposed system, the compute-intensive vision processing tasks are implemented on the FPGA, achieving a maximum frame rate of 96.7 fps using a Xilinx Virtex-4 FPGA and 136.4 fps using a Virtex-7 FPGA. The less compute-intensive tracking processing operations are implemented on an Intel i7-870 CPU (4 cores at 2.93 GHz) of the host-PC, requiring an average processing time of only 2.5 ms. As a result, the proposed system can achieve real-time player tracking with a maximum frame rate of 77.6 fps for an input frame resolution of 1392x1040 pixels from each of the two GigE Vision cameras. The results presented in this thesis show that FPGA technology significantly enhances the performance of the player tracking system, and off-loads the CPU from the compute-intensive vision processing tasks. The proposed reconfigurable system achieves a significantly higher computing performance than a software-based implementation. Utilizing a Xilinx Virtex-4 FPGA, a speedup by a factor of 15.5 is achieved in comparison to an OpenCV-based software implementation on a PC equipped with a 2.93 GHz i7-870 Intel CPU [107].

Logic resources and performance evaluations are measured for each implemented module, the overall FPGA utilization of the Virtex-4 FPGA is around 60%. Power consumption measurements are performed on the proposed system, including the consumed power by the FPGA-based daughterboard (DB-V4), RAPTOR board, and the host-PC. Moreover, the performance per Watt are calculated based on these power measurements. The results show that the proposed reconfigurable system achieves a 2.4 times higher performance per Watt than a software-based implementation (without FPGA support) on the host-PC. As compared with the existing systems in literature, the realized system in this work performs online and real-time player tracking using two dedicated cameras. Players of the two teams in handball and basketball games are tracked automatically with high precision and recall values. Additionally, the system supports a high frame rate up to 77.6 fps using FPGA technology.

6.2 Future Work

For the proposed system in this work, possible improvements to can be achieved in two levels: algorithmic and hardware. In the algorithmic level, the player detection can be improved in scenarios where two players of the same team are occluded for a long time. In this case, a possible ID switch could result between these players. Additionally, these players could be detected as one player. To solve these issues, the use of additional features (e.g., based on shapes) for player detection (in addition to the color of the jersey) can be investigated. In this case, the detection results from the proposed system (based on the color information) can be fused with a detector that uses these additional features, enhancing the player detection results.

In the hardware level, the currently used physical interface in the Gigabit Ethernet board supports up to 1 Gbps. This interface limits the number of the used GigE Vision cameras operating with their maximum resolution and frame rate, sharing this one Gigabit bandwidth. Therefore, an improvement can be achieved by using a 10 Gigabit Ethernet interface. In this case, more cameras can be supported, sharing the same 10 Gbps bandwidth. Another improvement can be made in the communication between the FPGA in the RAPTOR board and the CPU in the host-PC, which is currently achieved through the PCI-X interface. Therefore, an upgrade to the PCI Express (PCIe) as a high-speed interface for data transfer is recommended.

Additionally, the mapping of the less compute-intensive player tracking processing from the CPU on a host-PC to an embedded processor (e.g., an ARM processor) can be investigated, using a System on Chip (SoC) like the Xilinx Zynq SoC. In this case, the FPGA implementation presented in this work can be mapped to the programmable logic (PL) of the Zynq SoC, and the complete reconfigurable player tracking system can be implemented on a single chip, targeting a vision-based embedded system for player tracking. Figure 6.1 shows this concept, by which a player tracking vision box acquires the video data directly from the cameras through its 1 or 10 Gigabit Ethernet interfaces. Additionally, it performs the player tracking processing on its embedded SoC chip (e.g., Zynq SoC). Finally, it transfers the tracking results through a wireless interface to a trainer's tablet for a live real-time interaction. Moreover, the tracking results can be stored in the host-PC (e.g., for later evaluation) as shown in Figure 6.1.

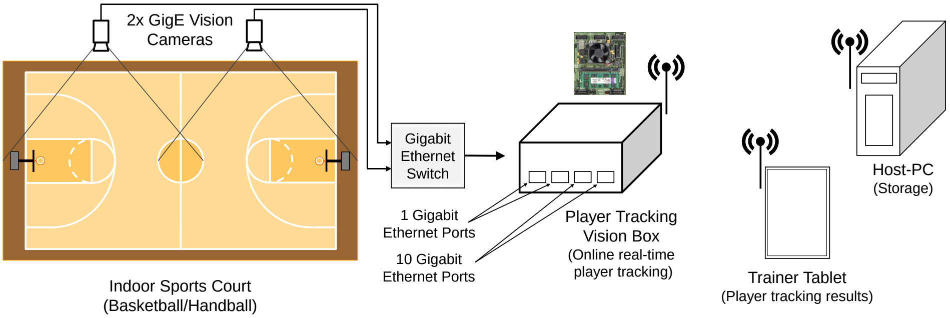


Figure 6.1: Player tracking vision box

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
AWB	Automatic White Balancing
AXI	Advanced Exensible Interface
BDC	Binary Distance Calculation
BRAM	Block RAM
CFA	Color Filter Array
CLB	Configurable Logic Block
CPU	Central Processing Unit
DB	Daughter Board
DDR	Double Data Rate
DMA	Direct Memory Access
DSP	Digital Signal Processor
EDK	Embedded Development Kit
EOL	End of Line
FF	Flip-Flop
FIFO	First-In First-Out
FN	False Negative
FP	False Positive
FPGA	Field Programmable Gate Array
fps	frame per second
Frag	Fragment
FSM	Finite State Machine
GMII	Gigabit Media Independent Interface
GPU	Graphics Processing Unit
GVCP	GigE Vision Control Protocol
GVSP	GigE Vision Streaming Protocol

GWA	Gray World Assumption
HDL	Hardware Description Language
HSV	Hue Saturation Value
ID Sw	Identity Switch
IP	Intellectual Property
ISE	Integerated Synthesis Enviroment
LL	Local Link
LUT	LookUp Table
MAC	Medium Access Control
MC_GigEV	Multi-Camera GigE Vision
ML	Mostly Lost
MOT	Multiple Object Tracking
MPMC	Multi-Port Memory Controller
MPSoC	Multiprocessor System on a Chip
MT	Mostly Tracked
NBA	National Basketball Association
NPI	Native Port Interface
PCI-X	Peripheral Component Interconnect eXtended
PCIe	PCI Express
PHY	Physical Layer
PLB	Processor Local Bus
PRA	Perfect Reflector Assumption
PT	Partially Tracked
RFSoC	Radio Frequency SoC
RGB	Red Green Blue
RONI	Region of Non-Interest
SDRAM	Synchronous Dynamic RAM
SoC	System on a Chip
SOF	Start of Frame
SRAM	Static Random Access Memory
TEMAC	Tri-Mode Ethernet Media Access Controller
TP	True Positive

UDP	User Datagram Protocol
USB	Universal Serial Bus
VFBC	Video Frame Buffer Controller
VHDL	Very High Speed Integrated Circuit Hardware Description Language

References

[1] D. Acuna. “Towards Real-Time Detection and Tracking of Basketball Players using Deep Neural Networks”. In: 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA. 2017.

[2] Adimek. *Vision connectivity interfaces, Choosing between Camera Link, CoaXPress, GigE Vision, Camera Link HS, 10 GigE Vision, and USB3 Vision*. Tech. rep. 2012. URL: `http://info.adimec.com`.

[3] AIA, EMVA, and JIIA. *Global machine vision interface standards: understanding today’s digital camera interface options*. 2014.

[4] A. Alahi, Y. Boursier, L. Jacques, and P. Vandergheynst. “Sport players detection and tracking with a mixed network of planar and omnidirectional cameras”. In: *3rd ACM/IEEE International Conference on Distributed Smart Cameras, ICDSC 2009*. 2009, pp. 1–8. DOI: `10.1109/ICDSC.2009.52893406`.

[5] E. S. Albuquerque, A. P. A. Ferreira, G. M. Silva, R. L. M. Carlos, D. S. Albuquerque, and E. N. S. Barros. “An FPGA-based accelerator for multiple real-time template matching”. In: *29th Symposium on Integrated Circuits and Systems Design (SBCCI)*. 2016, pp. 1–6. DOI: `10.1109/SBCCI.2016.7724071`.

[6] K. Amma, Y. Yaguchi, Y. Niitsuma, T. Matsuzaki, and R. Oka. “A comparative study of gesture recognition between RGB and HSV colors using time-space continuous dynamic programming”. In: *2013 International Joint Conference on Awareness Science and Technology & Ubi-Media Computing (iCAST 2013 & UMEDIA 2013)*. 2013, pp. 185–191. DOI: `10.1109/ICAwST.2013.6765431`.

[7] *APIDIS Basketball dataset*. URL: `http://sites.uclouvain.be/ispgroup/index.php/Softwares/APIDIS`.

[8] ARM. *AMBA 4 AXI4-Stream Protocol Specification v1.0*. Tech. rep. 2010.

[9] ARM. *AMBA AXI Protocol Specification, v2.0*. Tech. rep. 2010.

[10] Automated Imaging Association (AIA). *Camera Link HS - The Machine Vision Protocol Moving Forward*. 2016. URL: `http://www.visiononline.org/`.

[11] Automated Imaging Association (AIA). *Camera Link – The Only Real-Time Machine Vision Protocol*. URL: `http://www.visiononline.org/`.

[12] Automated Imaging Association (AIA). *GigE Vision - True Plug and Play Connectivity*. 2016. URL: `http://www.visiononline.org`.

[13] Automated Imaging Association (AIA). *GigE Vision, Video Streaming and Device Control over Ethernet Standard v2.0*. 2012.

[14] Automated Imaging Association (AIA). *USB3 Vision v1.0*. 2013.

[15] D. G. Bailey. *Design for embedded image processing on FPGAs*. John Wiley & Sons, 2011, p. 496. ISBN: 9780470828496.

[16] D. Bailey, S. Randhawa, and J. S. J. Li. "Advanced Bayer demosaicing on FPGAs". In: *2015 International Conference on Field Programmable Technology, FPT 2015*. 2016, pp. 216–220. DOI: `10.1109/FPT.2015.7393154`.

[17] B. E. Bayer. *Color imaging array*. 1975. URL: `https://www.google.com/patents/US3971065`.

[18] S. Benton. "Background subtraction, MATLAB models". In: *EETimes* (2008).

[19] K. Bernardin and R. Stiefelhagen. "Evaluating multiple object tracking performance: The CLEAR MOT metrics". In: *Eurasip Journal on Image and Video Processing* (2008). DOI: `10.1155/2008/246309`.

[20] BERTEN DSP. *GPU vs FPGA Performance Comparison (White Paper)*. 2016. URL: `http://www.bertendsp.com`.

[21] T. A. Biresaw, T. Nawaz, J. Ferryman, and A. I. Dell. "ViTBAT: Video tracking and behavior annotation tool". In: *2016 13th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS 2016*. Vol. 1. August. 2016, pp. 295–301. DOI: `10.1109/AVSS.2016.7738055`.

[22] D. D. Bloisi, A. Pennisi, and L. Iocchi. "Background modeling in the maritime domain". In: *Machine Vision and Applications* 25.5 (Dec. 2013), pp. 1257–1269. ISSN: 0932-8092.

[23] M. Bredereck, Xiaoyan Jiang, M. Korner, and J. Denzler. "Data association for multi-object Tracking-by-Detection in multi-camera networks". In: *Sixth International Conference on Distributed Smart Cameras (ICDSC), Hong Kong*. 2012, pp. 1–6. ISBN: 978-1-4503-1772-6.

[24] W. Brooks. *Intel Acquires Replay Technologies for Immersive Sports*. URL: `https://newsroom.intel.com/editorials/intel-acquires-replay-technologies/`.

[25] A. A. Butt and R. T. Collins. "Multi-target tracking by lagrangian relaxation to min-cost network flow". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2013, pp. 1846–1853. DOI: `10.1109/CVPR.2013.241`.

[26] C.-H. Chen, S.-Y. Tan, and W.-T. Huang. "A novel hardware-software co-design for automatic white balance". In: *Proceedings of the 7th WSEAS International Conference on Multimedia, Internet & Video Technologies, Beijing*. 2007, pp. 203–212.

[27] H. T. Chen, C. L. Chou, T. S. Fu, S. Y. Lee, and B. S. P. Lin. “Recognizing tactic patterns in broadcast basketball video using player trajectory”. In: *Journal of Visual Communication and Image Representation* 23.6 (2012), pp. 932–947. DOI: 10.1016/j.jvcir.2012.06.003.

[28] S.-C. Cheung and C. Kamath. “Robust techniques for background subtraction in urban traffic video”. In: *Proceedings of Video Communications and Image Processing, SPIE Electronic Imaging* (2004), pp. 881–892. DOI: 10.1117/12.526886.

[29] ChyronHego. *TRACAB Optical TRacking, Optical Sports Performance Tracking*. URL: https://chyronhego.com/products/sports-tracking/tracab-optical-tracking/.

[30] B. Cope, P. Cheung, W. Luk, and L. Howes. “Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study”. In: *IEEE Transactions on Computers* 59.4 (2010), pp. 433–448. DOI: 10.1109/TC.2009.179.

[31] A. Criminisi and J. Shotton. *Advances in Computer Vision and Pattern Recognition*. 2013, p. 368. DOI: 10.1007/978-1-4471-4929-3.

[32] P. H. De Padua, F. L. Padua, M. T. Sousa, and M. D. A. Pereira. “Particle Filter-Based Predictive Tracking of Futsal Players from a Single Stationary Camera”. In: *Brazilian Symposium of Computer Graphic and Image Processing*. Vol. 2015-Octob. 2015, pp. 134–141. DOI: 10.1109/SIBGRAPI.2015.10.

[33] Embedded Vision Alliance. *Processors for Embedded Vision*. URL: https://www.embedded-vision.com/technology/programmable-devices (visited on 01/02/2017).

[34] R. Faragher. “Understanding the basis of the kalman filter via a simple and intuitive derivation [Lecture Notes]”. In: *IEEE Signal Processing Magazine* 29.5 (2012), pp. 128–132. DOI: 10.1109/MSP.2012.2203621.

[35] D. Farin, P. H. N. de With, and W. Effelsberg. “Video-object segmentation using multi-sprite background subtraction”. In: *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763), Taipei*. 2004, pp. 343–346. DOI: 10.1109/ICME.2004.1394199.

[36] J. D. Foley, A. van Dam, F. Steven K, and J. F. Hughes. *Computer graphics: principles and practice*. Addison-Wesley, Boston, MA, USA, 1996. ISBN: 0-201-12110-7.

[37] A. Godil, R. Bostelman, M. Shneier, and W. Shackleford. “Performance Metrics for Evaluating Object and Human Detection and Tracking Systems”. In: 2014, pp. 1–13. DOI: 10.6028/NIST.IR.7972.

[38] A. Güneş, H. Kalkan, and E. Durmuş. “Optimizing the color-to-grayscale conversion for image classification”. In: *Signal, Image and Video Processing* 10.5 (2016), pp. 853–860. DOI: 10.1007/s11760-015-0828-7.

[39] M. C. Hanumantharaju, G. R. Vishalakshi, S. Halvi, and S. B. Satish. “Global Trends in Information Systems and Software Applications”. In: vol. 270. July. 2012. DOI: 10.1007/978-3-642-29216-3.

[40] Q. He, J. Wu, G. Yu, and C. Zhang. “SOT for MOT”. In: *arXiv:1712.01059v1* (2017). arXiv: 1712.01059. URL: http://arxiv.org/abs/1712.01059.

[41] M. C. Hu, M. H. Chang, J. L. Wu, and L. Chi. “Robust camera calibration and player tracking in broadcast basketball video”. In: *IEEE Transactions on Multimedia* 13.2 (2011), pp. 266–279. DOI: 10.1109/TMM.2010.2100373.

[42] C. Huang, B. Wu, and R. Nevatia. “Robust Object Tracking by Hierarchical Association of Detection Responses”. In: *Forsyth D., Torr P., Zisserman A. (eds) Computer Vision – ECCV 2008. Lecture Notes in Computer Science, vol 5303. Springer, Berlin, Heidelberg* (2008), pp. 788–801. DOI: 10.1007/978-3-540-88688-4_58.

[43] R. Hunt. *The Reproduction of Colour*. Wiley, 2004, p. 724. ISBN: 978-0-470-02425-6.

[44] Intel. *Intel redefines the fan experience for NBA All-Star weekend 2016*. 2016. URL: https://newsroom.intel.com/news-releases/intel-redefines-the-fan-experience-for-nba-all-star-weekend-2016/.

[45] A. Irwansyah. “Heterogeneous Computing Systems for Vision-based Multi-Robot Tracking”. PhD thesis. Bielefeld University, Germany, 2017.

[46] M. Jacobsen, S. Sampangi, Y. Freund, and R. Kastner. “Improving FPGA accelerated tracking with multiple online trained classifiers”. In: *24th International Conference on Field Programmable Logic and Applications, FPL 2014*. 2014. DOI: 10.1109/FPL.2014.6927505.

[47] JAI. *BM-141GE GigE Vision Camera (Datasheet)*. URL: http://www.jai.com/en/products/bb-141ge.

[48] S. Jeeva and M. Sivabalakrishnan. “Survey on background modeling and foreground detection for real time video surveillance”. In: *Procedia Computer Science* 50 (2015), pp. 566–571. DOI: 10.1016/j.procs.2015.04.085.

[49] D. Jurić. *Object Tracking: Kalman Filter with Ease*. 2015. URL: https://www.codeproject.com.

[50] R. E. Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Journal of Basic Engineering* 82.1 (1960), p. 35. DOI: 10.1115/1.3662552.

[51] E. Y. Lam and G. S. K. Fung. “Automatic White Balancing in Digital Photography”. In: *Single-Sensor Imaging: Methods and Applications for Digital Cameras*. Ed. by R. Lukac. CRC Press, 2009. Chap. 10, pp. 267–294. DOI: 10.1201/9781420054538.ch10.

[52] E. Lam. “Combining Gray World and Retinex Theory for Automatic White Balance in Digital Photography”. In: *Proceedings of the Ninth International Symposium on Consumer Electronics, (ISCE 2005)*. 2005, pp. 1–6. ISBN: 0780389204.

[53] E. Land. “The Retinex”. In: *American Scientist* 52 (1964), pp. 247–264.

[54] E. H. Land and J. J. McCann. “Lightness and Retinex theory”. In: *Journal of the Optical Society of America* 61.1 (1971), pp. 1–11.

[55] C. Li, L. Y. Yee, H. Maruyama, and Y. Yamaguchi. “FPGA-based Volleyball Player Tracker”. In: *ACM SIGARCH Computer Architecture News*. Vol. 44. 4. 2017, pp. 80–86. DOI: 10.1145/3039902.3039917.

[56] Y. Li, C. Huang, and R. Nevatia. “Learning to associate: Hybridboosted multi-target tracker for crowded scene”. In: *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2009* (2009), pp. 2953–2960. DOI: 10.1109/CVPRW.2009.5206735.

[57] O. Losson, L. Macaire, and Y. Yang. “Comparison of color demosaicing methods”. In: *Advances in Imaging and Electron Physics*. Vol. 162. C. 2010, pp. 173–265. DOI: 10.1016/S1076-5670(10)62005-8.

[58] W. L. Lu, J. A. Ting, J. J. Little, and K. P. Murphy. “Learning to track and identify players from broadcast sports videos”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.7 (2013), pp. 1704–1716. DOI: 10.1109/TPAMI.2012.242.

[59] W.-l. Lu. “Learning to Track and Identify Players from Broadcast Sports Videos”. PhD thesis. The University Of British Columbia, 2011.

[60] W. Luo, J. Xing, A. Milan, X. Zhang, W. Liu, X. Zhao, and T.-K. Kim. “Multiple Object Tracking: A Literature Review”. In: *arXiv:1409.7618v4* (2017), pp. 1–18. arXiv: 1409.7618.

[61] MathWorks. *Machine Learning with MATLAB (ebook)*. 2016, p. 12.

[62] C. Maxfield. *The Design Warrior’s Guide to FPGAs*. Newnes, 2004, p. 560. ISBN: 0750676043.

[63] P. L. Mazzeo, L. Giove, G. M. Moramarco, P. Spagnolo, and M. Leo. “HSV and RGB color histograms comparing for objects tracking among non overlapping FOVs, using CBTF”. In: *8th IEEE International Conference on Advanced Video and Signal Based Surveillance, AVSS*. 2011, pp. 498–503. DOI: 10.1109/AVSS.2011.6027383.

[64] N. J. B. McFarlane and C. P. Schofield. “Segmentation and tracking of piglets in images”. In: *In Machine Vision and Applications* 8.3 (1995), pp. 187–193. DOI: 10.1007/BF01215814.

[65] E. Monier. “Vision Based Tracking in Team Sports”. PhD thesis. Paderborn University, Germany, 2011.

[66] E. Monier, P. Wilhelm, and U. Ruckert. “Template matching based tracking of players in indoor team sports”. In: *In Third ACM/IEEE International Conference on Distributed Smart Cameras, ICDSC 2009.* 2009, pp. 1–6. DOI: 10.1109/ICDSC.2009.5289408.

[67] H. Morimitsu, I. Bloch, and R. M. Cesar-Jr. “Exploring structure for long-term tracking of multiple objects in sports videos”. In: *Computer Vision and Image Understanding* 159 (2017), pp. 89–104. DOI: 10.1016/j.cviu.2016.12.003. arXiv: 1612.06454.

[68] J. Munkres. “Algorithms for the Assignment and Transportation Problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 5.1 (1957), pp. 32–38. DOI: 10.1137/0105003.

[69] National Instruments. *Introduction to FPGA Technology: Top 5 Benefits*. 2012. URL: http://www.ni.com/white-paper/6984/en/.

[70] NVidia. *GPU vs CPU? What is GPU Computing?* 2016. URL: http://www.nvidia.com/object/what-is-gpu-computing.html.

[71] N. Otterness, M. Yang, S. Rust, E. Park, J. H. Anderson, F. D. Smith, A. Berg, and S. Wang. “An evaluation of the NVIDIA TX1 for supporting real-time computer-vision workloads”. In: *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*. 2017, pp. 353–363. DOI: 10.1109/RTAS.2017.3.

[72] P. H. de Pádua, F. L. Pádua, M. de A. Pereira, M. T. Sousa, M. B. de Oliveira, and E. F. Wanner. “A vision-based system to support tactical and physical analyses in futsal”. In: *Machine Vision and Applications* 28.5-6 (2017), pp. 475–496. DOI: 10.1007/s00138-017-0849-z.

[73] P. Parisot and C. De Vleeschouwer. “Scene-specific classifier for effective and efficient team sport players detection from a single calibrated camera”. In: *Computer Vision and Image Understanding* 159 (2017), pp. 74–88. DOI: 10.1016/j.cviu.2017.01.001.

[74] M. Porrmann, J. Hagemeyer, J. Romoth, M. Strugholtz, and C. Pohl. “RAPTOR-A scalable platform for rapid prototyping and FPGA-based cluster computing”. In: *Advances in Parallel Computing* 19 (2010), pp. 592–599. DOI: 10.3233/978-1-60750-530-3-592.

[75] *Replay Technologies*. URL: http://replay-technologies.com/.

[76] J. Romoth, J. Romoth, M. Porrmann, and R. Ulrich. *Survey of FPGA applications in the period 2000 – Survey of FPGA applications in the period 2000 – 2015*. Tech. rep. March. Bielefeld University, Germany, 2017. DOI: 10.13140/RG.2.2.16364.56960.

[77] C. B. Santiago, A. Sousa, M. L. Estriga, L. P. Reis, and M. Lames. "Survey on team tracking techniques applied to sports". In: *IEEE 2010 International Conference on Autonomous and Intelligent Systems, AIS 2010* (2010). DOI: 10.1109/AIS.2010.5547021.

[78] C. B. Santiago, A. Sousa, and L. P. Reis. "Vision system for tracking handball players using fuzzy color processing". In: *Machine Vision and Applications* 24 (2013), pp. 1055–1074. DOI: 10.1007/s00138-012-0471-z.

[79] C. Santiago, L. Gomes, A. Sousa, L. Reis, and M. Estriga. "Tracking Players in Indoor Sports Using a Vision System Inspired in Fuzzy and Parallel Processing". In: *Cutting Edge Research in New Technologies*. Ed. by P. C. Volosencu. 2012. DOI: 10.5772/2431.

[80] S. E. Schaeffer. "Graph clustering". In: *Computer Science Review* 1 (2007), pp. 27–64. DOI: 10.1016/j.cosrev.2007.05.001.

[81] Sensor to Image GmbH. *GigE Vision IP Specification (Document Revision X-1.5.2)*. Tech. rep. 2013.

[82] T. Sledevi. *FPGA-based Selected Object Tracking Using LBP, HOG and Motion Detection*. 2016. DOI: 10.13140/RG.2.1.4157.8967.

[83] A. Sobral and A. Vacavant. "A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos". In: *Computer Vision and Image Understanding* 122 (2014), pp. 4–21. DOI: 10.1016/j.cviu.2013.12.005.

[84] Songhwai Oh, S. Russell, and S. Sastry. "Markov chain Monte Carlo data association for general multiple-target tracking problems". In: *2004 43rd IEEE Conference on Decision and Control (CDC) (IEEE Cat. No.04CH37601)*. 2004, 735–742 Vol.1. DOI: 10.1109/CDC.2004.1428740.

[85] *STATS Website*. URL: www.stats.com.

[86] Stemmer Imaging. *Introduction to FPGA acceleration*. URL: https://www.stemmer-imaging.co.uk/en/technical-tips/introduction-to-fpga-acceleration/.

[87] Tom Catalino and Asheesh Bhardwaj. *An architecture for compute-intensive, custom machine vision (white paper)*. Tech. rep. Texas Instruments, 2013, p. 10.

[88] F. Torres. "A survey on FPGA-based sensor systems: towards intelligent and reconfigurable low-power sensors for computer vision, control and signal processing". In: *Sensors (Basel, Switzerland)* 14.4 (2014), pp. 6247–6278. DOI: 10.3390/s140406247.

[89] B. Treece. *CPU or FPGA for image processing: Which is best?* 2017. URL: http://www.vision-systems.com/.

[90] A. Trost and A. Žemva. “Rapid Prototyping of Embedded Video Processing Systems in FPGA Devices”. In: *Cutting Edge Research in Technologies*. Ed. by C. Volosencu. InTech, 2015. Chap. 3. DOI: 10.5772/61136.

[91] G. Welch and G. Bishop. *An Introduction to the Kalman Filter (TR 95-041)*. Tech. rep. Department of Computer Science, University of North Carolina at Chapel Hill, July 2006.

[92] Y. Xiang, A. Alahi, and S. Savarese. “Learning to Track: Online Multi- Object Tracking by Decision Making Multi-Object Tracking”. In: *2015 IEEE International Conference on Computer Vision (ICCV)*. 2015, pp. 4705–4713. DOI: 10.1109/ICCV.2015.534.

[93] Xilinx. *7 Series FPGAs Data Sheet (Product Specification, DS180-v2.4)*. 2017.

[94] Xilinx. *AXI Reference Guide (User Guide UG761-v13.2)*. 2011.

[95] Xilinx. *Color Filter Array Interpolation v7.0 (LogiCORE IP Product Guide-PG002)*. 2015.

[96] Xilinx. *Introduction to FPGA Design with Vivado High-Level Synthesis (UG998-v1.0)*. Tech. rep. 2013, pp. 1–89.

[97] Xilinx. *LocalLink Interface Specification (SP006-v2.0)*. 2005.

[98] Xilinx. *LogiCORE IP XPS LL TEMAC (Product Specification, DS537-v2.03a)*. 2010.

[99] Xilinx. *Multi-Port Memory Controller (DS643-v6.05.a)*. 2011.

[100] Xilinx. *UltraRAM : Breakthrough Embedded Memory Integration on UltraScale+ Devices (White Paper, WP477 (v1.0))*. Tech. rep. 2016, p. 11.

[101] Xilinx. *UltraScale Architecture and Product Data Sheet (DS890-v3.2)*. 2018.

[102] Xilinx. *Virtex-4 Family Overview (Product Specification, DS112-v3.1)*. 2010.

[103] Xilinx. *Zynq-7000 All Programmable SoC Data Sheet (DS190-v1.11)*. 2017.

[104] *Xilinx All Programmable*. URL: https://www.xilinx.com.

[105] G. Zapryanov, D. Ivanova, and I. Nikolova. “Automatic White Balance Algorithms for Digital Still Cameras – a Comparative Study”. In: *Journal of Information Technologies and Control* 1 (2012), pp. 16–22.

Publications

[106] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert. “A Resource-Efficient Multi-Camera GigE Vision IP Core for Embedded Vision Processing Platforms”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–6. DOI: 10.1109/ReConFig.2015.7393282.

[107] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert. “FPGA-Based Vision Processing System for Automatic Online Player Tracking in Indoor Sports”. In: *Journal of Signal Processing Systems* (2018), pp. 1–27. DOI: 10.1007/s11265-018-1381-8.

[108] O. W. Ibraheem, A. Irwansyah, J. Hagemeyer, M. Porrmann, and U. Rueckert. “Reconfigurable Vision Processing System for Player Tracking in Indoor Sports”. In: *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2017, pp. 1–6. DOI: 10.1109/DASIP.2017.8122114.

[109] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rueckert. “FPGA-based Circular Hough Transform with Graph Clustering for Vision-based Multi-Robot Tracking”. In: *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015, pp. 1–8. DOI: 10.1109/ReConFig.2015.7393313.

[110] A. Irwansyah, O. W. Ibraheem, J. Hagemeyer, M. Porrmann, and U. Rueckert. “FPGA-based multi-robot tracking”. In: *Journal of Parallel and Distributed Computing* 107 (2017), pp. 146–161. DOI: 10.1016/j.jpdc.2017.03.008.

[111] A. Irwansyah, O. W. Ibraheem, D. Klimeck, M. Porrmann, and U. Rueckert. “FPGA-based Generic Architecture for Rapid Prototyping of Video Hardware Accelerators using NoC AXI4-Stream Interconnect and GigE Vision Camera Interfaces”. In: *Bildverarbeitung in der Automation (BVAu)*. 2014, pp. 1–12.