

University of Minho
School of Engineering

ESRG Class 2020/2021

Hybrid Computer Vision:
Hardware-Accelerated Programmable
Computer Vision System

ESRG Report
Master's in Industrial Electronics and
Computers Engineering
Embedded Systems

Carried out under guidance of
PhD Adriano Tavares
Prof. Eng. Vítor Silva

Acronyms

AGAST Adaptive and Generic Accelerated Segment Test.

AMBA Advanced Microcontroller Bus Architecture.

AMP Asynchronous Multi-Processing.

API Application Programming Interface.

ASIC Application Specific Integrated Circuits.

AXI Advanced Extensible Interface.

BRAM Block Random-Access Memory.

BRIEF Binary Robust Independent Elementary Features.

BRISK Binary Robust Invariant Scalable Keypoints.

CAN Controller Area Network.

CLB Configurable Logic Block.

CPU Central Processing Unit.

CV Computer Vision.

DDR Double Data Rate.

DMA Direct Memory Access.

DRAM Dynamic Random-Access Memory.

DSL Domain-Specific Language.

DSP Digital signal processing.

DUT Device Under Test.

ESRG Embedded Systems Research Group.

FAST Features from Accelerated Segment Test.

FLANN Fast Library for Approximate Nearest Neighbour.

FPGA Field-Programmable Gate Array.

GPL General-Purpose Language.

HDL Hardware Description Language.

HLS High Level Synthesis.

I2C Inter-Integrated Circuit.

IC Integrated Circuit.

ICC Intel C++ Compiler.

IDE Integrated Development Environment.

IP Intellectual property.

ISR Interrupt Service Routine.

JDK Java Development Kit.

JSON JavaScript Object Notation.

JVM Java Virtual Machine.

KNN K nearest neighbours.

LAB Logic Array Blocks.

LSH Locality-Sensitive Hashing.

LUT Look-Up Table.

ML Machine Learning.

MMCM Mixed-Mode Clock Manager.

MMU Memory Management Unit.

MSVC Microsoft Visual C++.

NMS Non-Maximum Suppression.

ORB Oriented FAST and Rotated BRIEF.

OS Operating System.

OTP one-time programmable.

PISO Parallel In Serial Out.

PL Programmable Logic.

PLL Phase-Locked Loop.

PS Processing System.

RAM Random-Access Memory.

ROM Read-Only Memory.

RTL Register-Transfer Level.

SD Secure Digital.

SDIO Secure Digital Input Output.

SIFT Scale-Invariant Feature Transform.

SISO Serial In Serial Out.

SMP Synchronous Multi-Processing.

SoC System-on-Chip.

SPI Serial Peripheral Interface.

SRAM Static Random Access Memory.

SSR Score and Score Request.

SW Software.

UART Universal Asynchronous Receiver-Transmitter.

USB Universal Serial Bus.

UUT Unit Under Test.

VHDL Very High-Speed Integrated-Circuit Hardware Description Language.

VP Video Processing.

ZYBO ZYNQ Board.

Abstract

HWAPCVS: Hardware-Accelerated Programmable Computer Vision System

Computer vision systems usually rely on software to implement their algorithms as it is pretty flexible. Although, in some cases where the algorithms introduce exhaustive computations, the trade-offs might compensate switching to a hardware implementation to avoid system bottlenecks.

These hardware accelerators can not only boost the system's speed but also improve power efficiency if managed properly. To do this, one will resort to a FPGA accelerators deployed on a ZYNQ-7010 SoC.

To address the connection between the Processing System (PS) and the Programmable Logic (PL) within the ZYNQ Board (ZYBO), an AXI-based interface will be developed as well, allowing for both high throughput and low latency data exchange. The hardware accelerators' implementation will be extensively compared with their software counterpart to further assess the metrics that determine a successful algorithm offload.

In order to customise the variability points present in the annotated files of this project and generate an executable code from the files present in the repository a DSL will be used.

Keywords: keypoint detection, computer vision, algorithm hardware acceleration, generative programming

Contents

	Page
Acronyms	ii
List of Figures	ix
List of Tables	xvii
List of Listings	xix
1 Introduction	1
1.1 Problem Statement	1
1.2 Context and Motivation	1
1.3 Project Goals	2
1.3.1 To reach the first goal	2
1.3.2 To reach the second goal	2
1.4 Report Outline	2
2 Analysis	4
2.1 Background and State of the Art	4
2.1.1 Hardware Accelerators	4
2.1.2 Floating-Points	9
2.1.3 Detection and Description Algorithms	10
2.1.4 Matching algorithms	21
2.1.5 Outlier Removal	27
2.1.6 DSL	29
2.1.7 Benchmarking	32
2.1.8 Profiling	32
2.1.9 Profiler	33
2.1.10 Hardware Description Languages (HDL)	35
2.1.11 Field Programmable Gate Array (FPGA)	36
2.1.12 Advanced eXtensible Interface (AXI)	37
2.2 Requirements and Constraints	42
2.2.1 Functional Requirements	42
2.2.2 Non-Functional Requirements	42
2.2.3 Technical Constraints	42
2.2.4 Non-Technical Constraints	42
2.3 System Overview	42
2.4 Hardware Specification	43
2.5 System Architecture	44
2.6 Interface	44

2.7	Detection	45
2.8	ORB Detection	45
2.8.1	Overview	45
2.8.2	Rotation Invariance	46
2.8.3	Hardware acceleration	49
2.8.4	Software refactoring	51
2.9	BRISK Detection	53
2.10	Description	54
2.10.1	State Chart	54
2.10.2	Hardware Acceleration	55
2.10.3	Software Refactoring	57
2.11	Matching	58
2.11.1	Software Refactored	59
2.11.2	Hardware	60
2.12	DSL	62
2.12.1	Objectives	62
2.12.2	DSL constitution	62
2.12.3	DSL compilation states	63
2.12.4	Code Generation	63
2.12.5	Use Cases	64
2.12.6	Grammar	65
2.13	Benchmarking/Profiling Tools and Compilers	66
2.13.1	OProfile	66
2.13.2	VTune	68
2.13.3	Microsoft Visual C++ Compiler	68
2.13.4	Intel C++ Compiler	68
3	Design	69
3.1	Interface	69
3.2	ORB Detection	70
3.2.1	Patcher	70
3.2.2	Dark/Bright classifier	71
3.2.3	Contiguity Counter	72
3.2.4	Scoring	73
3.2.5	Contiguity Test	73
3.2.6	Non-maximum Suppression	75
3.2.7	Software Refactored	79
3.3	BRISK Detection	82
3.3.1	Image Slicing	82
3.3.2	Dynamic Mapping	85
3.3.3	Controller	86
3.3.4	Bright/Dark Classifier	89
3.3.5	Contiguity Test	90

3.3.6	Non-maximum Suppression	92
3.3.7	Software Refactoring	95
3.4	Description	97
3.4.1	Memory Access	97
3.4.2	Pair Generation	100
3.4.3	Binary Test	101
3.4.4	Software Refactoring	103
3.5	Matching	106
3.5.1	Matching Hardware Specification	106
3.5.2	Software and Algorithm Specification	113
3.5.3	Class Diagram	115
3.6	DSL Design	116
3.6.1	Syntax	116
3.6.2	Validation Rules	117
3.6.3	Code generation approaches	118
3.7	Profiling	120
3.7.1	Tool choice	120
3.7.2	Test Design	120
3.8	Test Cases	123
3.8.1	DSL Unit and Integration Tests	123
3.8.2	ORB Detection Unit and Integration Tests	124
3.8.3	BRISK Detection Unit and Integration Tests	124
3.8.4	BRISK Scoring and Contiguity Unit and Integration Tests	125
3.8.5	BRISK NMS Unit and Integration Tests	125
3.8.6	BRIEF	125
3.8.7	Matching	126
3.8.8	Hardware	126
3.8.9	Software	127
4	Implementation	128
4.1	ORB	128
4.1.1	Corner Score Computation	128
4.1.2	Modules implementation	129
4.1.3	Modules integration	131
4.1.4	NMS	132
4.1.5	Hardware Approaches	133
4.1.6	Software Refactoring	137
4.2	BRISK	140
4.2.1	Image Slicing and BRAM	140
4.2.2	Bright/Dark Classifier	142
4.2.3	Detection Controller and Register File	143
4.2.4	Detection Integration and IP Packaging	145
4.2.5	Score Computation	147

4.2.6	Contiguity Test	150
4.2.7	2D Non-maximum Suppression	151
4.2.8	3D NMS	155
4.2.9	Detection integration	156
4.2.10	Software Refactoring	157
4.3	BRIEF	160
4.3.1	FIFO Controller	161
4.3.2	Patch Computation	163
4.3.3	Register File	166
4.3.4	Binary Test	169
4.3.5	Systolic Architecture	170
4.3.6	Software Refactoring	175
4.4	Matching	180
4.4.1	Hardware	180
4.4.2	BRAM	186
4.4.3	Controller	187
4.4.4	Cross Checking	191
4.4.5	Wrapper Module	192
4.4.6	Software	192
4.4.7	Software Refactored: Matching	193
4.5	DSL	195
4.5.1	Grammar Rules	195
4.5.2	Validation Rules	197
4.5.3	Code generation	197
4.6	Profiling	206
4.6.1	ORB Detector	207
4.6.2	ORB Descriptor	207
4.6.3	Open-CV BRISK Detector	208
4.7	Linux Image Creation and Memory Reservation	210
4.7.1	Linux Image Creation	210
4.7.2	Memory Allocation	211
4.8	Register-based Configuration	212
5	Testing and Result Assessment	215
5.1	Generation	215
5.2	Execution	218
5.2.1	ORB Hardware	218
5.2.2	ORB Software	229
5.2.3	BRISK	230
5.2.4	BRIEF	239
5.2.5	Matching	246
5.3	Software Refactoring Integration	252
5.4	Profiling and Benchmarking	256

5.4.1	VTune Profiling of the ORB Descriptor	256
5.4.2	MSVC/ICC Results for BRISK Detection	259
5.4.3	OProfile Profiling	262
5.4.4	ORB Detection	262
5.4.5	ORB Descriptor	264
5.4.6	BRIEF Refactored	267
5.4.7	BRISK Detection	268
5.5	BRISK MSVC vs. ICC vs. OProfile	268
5.6	Verification	269
5.6.1	DSL Unit and Integration Tests	269
5.6.2	ORB Detection Unit and Integration Tests	269
5.6.3	ORB- Software Refactoring	270
5.6.4	BRISK Detection Unit and Integration Tests	270
5.6.5	BRISK Scoring and Contiguity Test Unit and Integration Tests	270
5.6.6	BRISK NMS Unit and Integration Tests	270
5.6.7	BRISK Software Refactoring	271
5.6.8	BRIEF	271
5.6.9	Matching	272
6	Conclusion	274
6.1	Future Works	274
Bibliography		275
Appendices		277
A	Testbenches	278
B	Scripts	286
C	Project Planning	288
D	Schematics	289

List of Figures

	Page
Chapter 1	
1.1.1 Expected market growth of Computer Vision Technologies	1
1.4.1 Waterfall Methodology	3
Chapter 2	
2.1.1 Overview of blocking and non-blocking hardware calls	4
2.1.2 Overview of shared and dedicated accelerators	5
2.1.3 Common data copy between Processor and Accelerator	5
2.1.4 <i>Zero-copy</i> approach	6
2.1.5 Data copy between Processor and Accelerator through DMA Controller	6
2.1.6 Hardware Register Interface	6
2.1.7 Software Interface	7
2.1.8 Memory Write handled through ISR	7
2.1.9 Overview of Cache Coherency handling when writing directly to memory	8
2.1.10 Independent Accelerator stages	8
2.1.11 Chained Accelerator stages	9
2.1.12 Single-precision (32-bit) form (bias = 127)	9
2.1.13 FAST-n: Detection Overview	10
2.1.14 FAST-n: Bresenham Circle Possibilities	11
2.1.16 Scale-invariance in BRISK	12
2.1.17 Octave processing	13
2.1.18 Inputs general format.	14
2.1.19 Assignment example.	14
2.1.20 Conditions verification.	15
2.1.21 Stage outcome.	15
2.1.22 Processing layers.	16
2.1.23 ORB detection stage overview	16
2.1.24 7x7 pixel patch	17
2.1.25 Performance of BRIEF against other descriptors	18
2.1.26 Different approaches for pair sampling	19
2.1.27 Rotation invariance between BRIEF and other descriptors	20
2.1.28 Example of sampling pairs	21
2.1.29 Clustering Example	22
2.1.30 Brute-force algorithm	23

2.1.31	Comparison of brute-force matching with and without cross-checking	25
2.1.32	Brute force matching using kNN	25
2.1.33	Brute-force matching using $k = 2$ and outlier removal	26
2.1.34	Example of Point correspondences with outliers	28
2.1.35	Hierarchical Organisation of an HDL Design	35
2.1.36	HDL Testbench	36
2.1.37	Underlying FPGA Fabric	36
2.1.38	AXI Read and Write Channels	38
2.1.39	Example of an AXI Transfer	38
2.1.40	Example of an AXI Read Transaction (Memory Mapped)	39
2.1.41	Example of an AXI Write Transaction (Memory Mapped)	39
2.1.42	Example of an AXI4-Stream Transaction	40
2.1.43	Example of an AXI4-Stream Handshake (1)	41
2.1.44	Example of an AXI4-Stream Handshake (2)	41
2.1.45	Example of an AXI4-Stream Handshake (3)	41
2.3.1	System Big Picture	42
2.4.1	Zybo Z7-10 Development Board	43
2.5.1	System Architecture: Overview	44
2.6.1	Hardware Architecture: Processing System Interface	44
2.7.1	ORB and BRISK block diagram	45
2.8.1	Overview of ORB detector architecture	46
2.8.2	Module Overview	46
2.8.3	Module Overview Flowchart	47
2.8.4	Hybrid Approach Diagram	48
2.8.5	Hardware approach diagram	48
2.8.6	ORB module state chart	49
2.8.7	ORB Use Case diagram	50
2.8.8	ORB Sequence Diagram	51
2.8.9	Use Case Diagram (Software Refactored)	52
2.8.10	Use Case Diagram (Software Refactored)	52
2.9.1	Hardware Architecture: Detector Overview	53
2.10.1	BRIEF block diagram	54
2.10.2	State Chart of BRIEF module	54
2.10.3	BRIEF Use Case Diagram (Hardware)	55
2.10.4	BRIEF Sequence Diagram (Hardware)	56
2.10.5	BRIEF Use Case Diagram (Software)	57
2.10.6	BRIEF Sequence Diagram (Software)	58
2.11.1	Matching Software Overview	59
2.11.2	Matching Software Use Cases	59
2.11.3	Contextualization of the hardware implementation with the project	60
2.11.4	Matching Hardware Use Cases	60
2.11.5	Matching State Machine	61
2.12.1	DSL constitution	62

2.12.2	DSL compilation states	63
2.12.3	Code Generation Overview	64
2.12.4	DSL use cases	65

Chapter 3

3.1.1	Hardware Architecture: Processing System Interface	69
3.2.1	Bresenham circle mask for FAST-12, FAST-9 and FAST-5, respectively	70
3.2.2	Patcher design overview	71
3.2.3	Pipelining stages of the dark/bright classifier stage	71
3.2.4	Contiguity Counter Algorithm.	72
3.2.5	Scoring Module.	73
3.2.6	Contiguity Test Module.	73
3.2.7	Corner Score Computation Module.	74
3.2.8	Corner Score Computation Module.	74
3.2.9	Flowchart of execution time measurement	75
3.2.10	- First step of any hardware approach	75
3.2.11	Floating-point (7.1) IP core customisation window	76
3.2.12	Resource usage of Floating-point IP core configured for division operation	77
3.2.13	Diagram of the Hardware with Floating-Point Numbers Approach	77
3.2.14	Diagram of the Hardware with Fixed-Point Numbers Approach	78
3.2.15	CORDIC IP Customisation Window	78
3.2.16	ORB - Software Refactoring State Machine	80
3.2.17	Flowchart detect function	80
3.2.18	Flowchart detect function	81
3.3.1	Image Slicing: Overview (No Limitation)	82
3.3.2	Image Slicing: Overview (Limited)	83
3.3.3	Memory Read Operation	84
3.3.4	Memory Write Operation	84
3.3.5	Dynamic Mapping Overview	85
3.3.6	Buffer Controller Overview	86
3.3.7	Simplified view of the data flow between the Controller and the Bright/Dark Classifier - 12 columns and 6 cycles of delay	87
3.3.8	Simplified view of the data flow between the Controller and the Bright/Dark Classifier - 12 columns and 10 cycles of delay	87
3.3.9	Controller State Machine	88
3.3.10	Hardware Architecture: Bright/Dark Classifier	89
3.3.11	16 Possible States for FAST-5, FAST-9 and FAST-12	90
3.3.12	Parallel Hardware Architecture For Contiguity Test Where N Can Be 7,11 or 15	90
3.3.13	Deconcatenation of the Outputs from Bright/Dark Classifier in case its FAST-12	91
3.3.14	Score Computation for the Dark Group	91
3.3.15	Comparison Between the Scores	92

3.3.16	NMS Filter	92
3.3.17	2D Processing Logic	93
3.3.18	Horizontal NMS - State machine.	93
3.3.19	Data rearrangement.	94
3.3.21	State Chart Diagram	95
3.3.22	Class Diagram	96
3.4.1	Keypoint location retrieval and binary descriptor string storage flowchart	97
3.4.2	Patch creation module diagram	97
3.4.3	Register File diagram	98
3.4.4	Example image representation	98
3.4.5	Pair Generation Overview	100
3.4.6	Recognition rate of the five different sampling geometries	100
3.4.7	Pair generation function flowchart	101
3.4.8	Binary Vector Computation Module Diagram	101
3.4.9	Flowchart for the binary test	102
3.4.10	Software Refactoring State Machine	104
3.4.11	Software Refactoring State Machine	104
3.4.12	Software Refactoring Class Diagram	105
3.4.13	Sampling pattern generated by Matlab with the descriptor size as 64 and the patch size as 11	105
3.4.14	Flowchart for the descriptor generation in software	106
3.5.1	BRAM design for storing and getting the descriptors	107
3.5.2	Matching Hardware Overview diagram	108
3.5.3	Matching Core Diagram	110
3.5.4	Pipelining for each matching core	111
3.5.5	Parallelism and Pipelining in Matching	111
3.5.6	Cross Checking diagram	112
3.5.7	Cross-Checking Flowchart part 1	113
3.5.8	Cross-Checking Flowchart part 2	114
3.5.9	Cross-Checking Flowchart part 3	114
3.5.10	Slope-based Rejection Algorithm	115
3.5.11	Matching Class Diagram	116
3.6.1	Second approach	118
3.6.2	Flowchart of the generic function to replace annotations	119
3.6.3	Flowchart of how the code inside false condition is deleted	120
3.7.1	OProfile CPU-CLK-UNHALTED	121
3.7.2	VTune Profiling 1	122
3.7.4	VTune: Parameter Configuration	123

Chapter 4

4.1.1	RTL Analysis Schematic.	130
4.1.2	Linker Settings Configuration.	132

4.1.3	Hardware with Floating-Point Numbers Approach RTL	133
4.1.4	Utilisation report for Hardware with Floating-Point Numbers Approach	135
4.1.5	Hardware with fixed-point numbers approach module's declaration.	135
4.1.6	Utilisation report for Hardware with Fixed-Point Numbers Approach	137
4.2.1	BRAM Output Forwarding	141
4.2.2	BRISK Image Slice RTL Schematic	142
4.2.3	Classifier Module: Schematic (FAST-12)	143
4.2.4	BRISK Classification RTL Schematic	145
4.2.5	IP Packaging and Usage Flow	146
4.2.6	BRISK Classification Block Design	146
4.2.7	Score Schematic for FAST-5 Module (Augmented in fig.D.3)	147
4.2.8	Score Schematic for FAST-9 Module (Augmented in fig.D.4)	148
4.2.9	Score Schematic for FAST-12 Module (Augmented in fig.D.5)	148
4.2.10	Example of a Contiguity Test(1)	150
4.2.11	Example of a Contiguity Test(2)	150
4.2.12	Generic schematic.	152
4.2.13	Horizontal data handling schematic.	153
4.2.14	Filter's schematic.	155
4.2.15	Octave schematic.	156
4.2.16	Schematic: Integration of BRISK detection stages.	156
4.3.1	BRIEF RTL Diagram	161
4.3.2	BRIEF with systolic architecture RTL Diagram	161
4.3.3	Patch Computation RTL diagram	165
4.3.4	Register File workflow	167
4.3.5	Register File RTL diagram	168
4.3.6	Execution time for BRIEF without and with parallelism	170
4.3.7	Block diagram for the implementation of the BRIEF parallelism	171
4.3.8	RTL design of the register file controller	172
4.3.9	RTL design of the output controller	174
4.4.1	Matching Core Diagram Modules	181
4.4.2	Matching Core Pipeline Registers and Stages	181
4.4.3	Matching Core Sums	182
4.4.4	Matching Core RTL schematic	182
4.4.5	Matching Core LUT and FF	183
4.4.6	Matching Core Nine stages Schematic	183
4.4.7	Matching Core Last Stages	184
4.4.8	Matching Core Counter RTL	184
4.4.9	Wrapper Module	192
4.4.10	Wrapper Module Resources	192
4.4.11	Overall Matching Implementation Diagram	193
4.7.1	Steps to build an image linux for Zybo z7-10	210
4.7.2	Boot Components for linux image	211
4.7.3	Root Components for linux image	211

4.7.4	Memory reservation through changing kernel base address	212
4.8.1	Register Map	214
4.8.2	Block Design of the Integration of the Register-based Control System with the Processing System	214

Chapter 5

5.1.1	Error: Wrong object name (should be "System")	215
5.1.2	Error: Wrong Operation mode (should be "Sw")	215
5.1.3	Error: Wrong detector name (should be BRISK or ORB)	215
5.1.4	Error: Wrong descriptor name (should be BRIEF)	215
5.1.5	Error: Wrong parameter (should be N-FASTN)	216
5.1.6	Error: Wrong type (should be integer)	216
5.1.7	Error: Wrong type (should be integer)	216
5.1.8	Error: Wrong StringSize value (should be 64, 128 or 256)	216
5.1.9	Error: Wrong PatchSize value (should be 11, 15 or 19)	216
5.1.10	Error: Wrong N-FASTN value (should be 5, 9 or 12)	216
5.1.11	Command line result	217
5.1.12	Folders for all the algorithms	217
5.1.13	Project Repository for BRISK	217
5.1.14	Final Result	218
5.2.1	Conversion of the image to hexadecimal values	218
5.2.2	Patch Valid for the first time for a Fast of order 12	219
5.2.3	Patch Valid for the first time for a Fast of order 9	219
5.2.4	Patch Valid for the first time for a Fast of order 5	220
5.2.5	Patch Valid to invalid transition for the first time for a Fast of order 12	220
5.2.6	Patch Valid to invalid transition for the first time for a Fast of order 9	221
5.2.7	Patch Valid to invalid transition for the first time for a Fast of order 5	221
5.2.8	Patch invalid to valid second transition for the first time for a Fast of order 12	222
5.2.9	Patch invalid to valid second transition for the first time for a Fast of order 9	222
5.2.10	Patch invalid to valid second transition for the first time for a Fast of order 5	223
5.2.11	Corner Score Computation TestBench.	223
5.2.12	Resource Usage Table.	224
5.2.13	Power Consumption Summary.	224
5.2.15	Hybrid approach validation.	226
5.2.16	Hardware with floating-point numbers approach validation	226
5.2.17	Results compilation	226
5.2.18	ORB Block Diagram	227
5.2.19	ORB Simulation - Detector, Scorer, NMS	228
5.2.20	ORB - NMS and Rotation Invariant	228
5.2.21	Final Result	229
5.2.22	Image Slice Module Simulation: Details	230
5.2.23	Image Slice Module Simulation: Dynamic Mapping	231

5.2.24	Simulation: Image slices read from BRAM	231
5.2.25	Simulation: Register File	232
5.2.26	Simulation: Bresenham Circle FAST-12	232
5.2.27	Simulation: Classifier	233
5.2.28	Contiguity Test Simulation for FAST-5,FAST-9 and FAST-12	234
5.2.29	Horizontal filter suppression.	234
5.2.30	Vertical filter suppression.	235
5.2.31	3D flow control.	235
5.2.32	3D comparison.	235
5.2.33	Simulation of FAST-12 Score	236
5.2.34	Integration Between Detection e Scoring Stages	237
5.2.35	Output and BRIEF connection.	237
5.2.36	Final result	238
5.2.37	Simulation: FIFO Controller	239
5.2.38	Simulation: Patch Computation	239
5.2.39	Simulation: Register File	240
5.2.40	Simulation: Register File output pairs	240
5.2.41	Simulation: Binary Test	241
5.2.42	Simulation: Patch Computation with the systolic architecture	241
5.2.43	Simulation: BRIEF with the default architecture	242
5.2.44	Simulation: BRIEF with the systolic architecture	242
5.2.45	Resource Utilisation: N=11, 64 bit description string	242
5.2.46	Resource Utilisation: N=15, 128 bit description string	243
5.2.47	Resource Utilisation: N=19, 256 bit description string	243
5.2.48	BRIEF software implementation validation of the sampling pattern	244
5.2.49	BRIEF software implementation validation of the descriptors	244
5.2.50	Matching Core Simulation	246
5.2.51	Matching Core Simulation	246
5.2.52	Matching Core Simulation with index of the best selection	247
5.2.53	BRAM Simulation	248
5.2.54	BRAM Simulation	248
5.2.55	Controller Simulation	249
5.2.56	Controller Simulation	249
5.2.57	Cross Checking simulation	250
5.2.58	Parallelism simulation	250
5.2.59	Parallelism simulation	251
5.3.1	Test 1	253
5.3.2	Test 2	253
5.3.3	Test 3	253
5.3.4	Test 4	254
5.3.5	Test 5	254
5.3.6	Test 6	254
5.4.1	Test: Fewer Keypoints	256

5.4.2	Test: Additional Keypoints	256
5.4.3	Test: Execution with Additional Keypoints	257
5.4.4	Results: Execution with Additional Keypoints vs Fewer Keypoints	257
5.4.6	Brief CPU Load	258
5.4.7	ORB Hotspots	259
5.4.8	BRISK Detection Time Analysis	259
5.4.9	BRISK Detection Top Hotspots	260
5.4.10	BRISK Detection Top Tasks	260
5.4.11	BRISK Detection Time Analysis	261
5.4.12	BRISK Detection Top Hotspots	261
5.4.13	BRISK Detection Top Tasks	261
5.4.14	Three images used to profile the ORB Detection phase	262
5.4.15	Results of oProfile in the Eagle image.	263
5.4.16	Results of oProfile in the Castle image.	263
5.4.17	Results of oProfile in the Lion image.	264
5.4.18	Results: Test 1	265
5.4.19	Results: Test 2	265
5.4.20	Results: Test 3	266
5.4.21	Results: Test 4	266
5.4.22	Profile Refactored	267
5.5.1	MSVC vs. ICC vs. OProfile Graph	268

Chapter C

C.1	ESRG Project Plan	288
-----	-----------------------------	-----

Chapter D

D.1	Classifier Schematic (FAST-12)	289
D.2	Schematic: Bit Concatenation	290
D.3	Schematic: Score FAST-5	291
D.4	Schematic: Score FAST-9	292
D.5	Schematic: Score FAST-12	293
D.6	Schematic: Integration of BRISK detection stages	294

List of Tables

	Page
Chapter 2	
2.1 HDL Candidates Gap Analysis	36
2.2 Grammar Entities used	66
2.3 Keywords used	66
Chapter 3	
3.1 Validation Rules	117
3.2 Gap analysis table between VTune and OProfile	120
3.3 DSL Unit Tests	123
3.4 DSL Integration Tests	124
3.5 ORB Unit and Integration Tests	124
3.6 ORB - Test Cases for Software Implementation	124
3.7 BRISK Detection Unit and Integration Tests	125
3.8 BRISK Scoring and Contiguity Test Unit and Integration Tests	125
3.9 BRISK NMS Unit and Integration Tests	125
3.10 BRIEF Unit and Integration Test Cases for Hardware Implementation	126
3.11 BRIEF Unit and Integration Test Cases for Software Implementation	126
3.12 Matching Unit and Integration Test Cases for Hardware Implementation	126
3.13 Matching Unit and Integration Test Cases for Software Implementation	127
Chapter 4	
4.1 Comparison between implementation with and without boost	160
Chapter 5	
5.1 HW vs SW test.	225
5.2 Hardware with fixed-point numbers approach validation	227
5.3 OpenCV vs Software Refactored.	229
5.4 Average Execution Time	238
5.5 Average Execution Time of BRIEF implementations	245
5.6 Smallest Difference and Index of best Matches	249
5.7 MSVC Time Table	259

5.8	ICC Time Table	260
5.9	oProfile results - ORB Detection Phase	264
5.10	Results: Test 1 Timing	265
5.11	Results: Test 2 Timing	266
5.12	Results: Test 3 Timing	266
5.13	Results: Test 4 Timing	267
5.14	Results: Test 5 Timing	267
5.15	OProfile Time Table	268
5.16	MSVC vs. ICC vs. OProfile	268
5.17	DSL Unit Tests	269
5.18	DSL Integrated Tests	269
5.19	ORB Unit and Integration Tests	269
5.20	ORB - Software Implementation	270
5.21	Unit and Integrated Tests	270
5.22	BRISK Scoring and Contiguity Test Unit and Integration Tests	271
5.23	BRISK NMS Unit and Integration Tests	271
5.24	BRISK Unit and Integration Test Cases for Software Implementation	271
5.25	BRIEF Unit and Integration Test Cases for Hardware Implementation	272
5.26	BRIEF Unit and Integration Test Cases for Software Implementation	272
5.27	Matching Unit and Integration Test Cases for Hardware Implementation	272
5.28	Matching Unit and Integration Test Cases for Software Implementation	273

List of Listings

3.1	Instantiation template of the Xilinx CORDIC IP with the desired customisation	79
3.2	create() Method	79
3.3	detect() Method	79
3.4	create() Method	103
3.5	compute() Method	103
3.6	DSL Syntax	116
3.7	Execution of the operf command	121
3.8	Execution of the oreport command	121
4.1	Scorer Inputs and Outputs.	128
4.2	Contiguity Counter Inputs and Outputs.	129
4.3	Scorer implementation.	130
4.4	Contiguity Counter implementation	130
4.5	Contiguity Counter instantiation.	131
4.6	Usage of the global timer counter register for execution time calculation	132
4.7	Intermediary Pipeline Register Module	133
4.8	Sum of the squares of both patch's moments	133
4.9	Output of the stage 1	134
4.10	Output of stage 2	134
4.11	Output of stage 3	134
4.12	Output of stage 4	135
4.13	Hardware with fixed-point numbers approach module's declaration	136
4.14	Sum of the patch's moments' squares computation	136
4.15	Sin and cos of the rotation angle computation	136
4.16	Oriented keypoint's coordinates computation	136
4.17	ORB class	137
4.18	Bresenham circle for all fasts	137
4.19	detect function	138
4.20	score function	139
4.21	Image Slice Module: Output Forwarding and BRAM Generation	140
4.22	Image Slice Module: Output Forwarding and BRAM Generation	141
4.23	Image Slice Module: Dynamic Mapping	142
4.24	Bright/Dark Classifier Interface	142
4.25	Classifier Module: Generated Block	143
4.26	Detection Controller Interface	143
4.27	Detection Controller - Counters	144
4.28	Detection Controller - Register File	144
4.29	Detection Controller - Non-Max Suppression interface	145
4.30	FAST-5 Module	147
4.31	FAST-9 Module	147

4.32	FAST-12 Module	147
4.33	Score Adders for FAST-9 Module	149
4.34	Score Assignment and Transition Between Registers	149
4.35	FAST-5 Module	150
4.36	FAST-9 Module	150
4.37	FAST-12 Module	150
4.38	Contiguity Test for FAST-5 Module	151
4.39	Transition Between Registers	151
4.40	Defines.	152
4.41	Module declaration.	152
4.42	Horizontal inst.	152
4.43	Transition inst.	152
4.44	Vertical inst.	152
4.45	State machine implementation - part I.	153
4.46	State machine implementation - part II.	153
4.47	Transition module essential code.	154
4.48	Defines	155
4.49	3D Module - Control, interpolation and suppression	155
4.50	Transformation arrays for all the fasts	157
4.51	BRISK structs using boost library	158
4.52	Function to convert image to boost library's matrix	159
4.53	BRISK compute_threads function	160
4.54	Create and Joint Threads functions	160
4.55	Fifo Controller module declaration	161
4.56	Fifo Controller module's outputs assignment	162
4.57	Fifo Controller module control	162
4.58	Patch Computation module declaration	163
4.59	Patch Computation mem_rdy flag	163
4.60	Patch Computation mem_valid flag	163
4.61	Patch Computation line_iterator and mem_has_data flag	164
4.62	Patch Computation extended_address	165
4.63	Patch Computation patchline conditional output	165
4.64	Register File module declaration	166
4.65	Register File's control variables	167
4.66	Register File's output pairs	168
4.67	Register File's control variables	168
4.68	Binary Test module declaration	169
4.69	Computation of patch pair addresses	170
4.70	Computation of descriptor	170
4.71	Register File Controller module declaration	171
4.72	Computation of the register file controller	172
4.73	Output Controller module declaration	173
4.74	Computation of the output controller	173

4.75 Matlab sampling pattern generation code	175
4.76 Initial Software Implementation of BRIEF	176
4.77 BRIEF software implementation using boost library	176
4.78 Software implementation of the general BRIEF compute function	177
4.79 Software implementation of main function for BRIEF with boost optimizations	177
4.80 Software implementation of BRIEF binary test without OpenCV dependencies	178
4.81 Software implementation of BRIEF constructor for Multi-threading	178
4.82 Software implementation of BRIEF Create and Joint Threads functions	179
4.83 Software implementation of BRIEF compute function for Multi-threading	179
4.84 Software implementation of main function for BRIEF with Multi-threading	179
4.85 Matching Core Code	184
4.86 Matching Core Code	185
4.87 Matching Core Code	185
4.88 Matching Core Code	186
4.89 Matching Core Code	186
4.90 BRAM Code	187
4.91 Controller Core Inputs and Outputs	187
4.92 Controller module Store Patterns	188
4.93 Controller module Store Patterns - continuation	188
4.94 Controller States	189
4.95 Controller State Machine Status	189
4.96 Controller descriptors address	189
4.97 Controller State Machine code	190
4.98 Cross Checking Array	191
4.99 Brute Force Method	194
4.100 Hamming Distance Method	194
4.101 Cross Checking Method	194
4.102 Clear False Matches Method	195
4.103 Grammar rules	196
4.104 Validation Rules	197
4.105 Generation code of OpenCV implementation for the ORB algorithm	198
4.106 Annotations used in Sw-Refactored script for orb	198
4.107 Generation code of OpenCV implementation	198
4.108 Generation code of OpenCV implementation	199
4.109 ORB SW Refactored code generation	199
4.110 ORB SW Refactored annotated code	199
4.111 Generation code of Software Refactored header file	200
4.112 Annotations used in Sw-Refactored script	200
4.113 Annotations used in Sw-Refactored script	200
4.114 Annotations used in Sw-Refactored script	201
4.115 Generation code of Software Refactored header file	201
4.116 Example of annotations used in SW-Refactored Code	201
4.117 Matching sw-Refactored code generation	202

4.118 ORB Hybrid code generation	202
4.119 Generation code of Hybrid implementation	203
4.120 How the depth is obtained	204
4.121 Variable Depth annotation usage	204
4.122 How the circle size is obtained	204
4.123 Variable CircleSize annotation usage	204
4.124 BRIEF Hybrid code generation	204
4.125 BRIEF Hybrid conditional code segment	205
4.126 Generation code of Matching Hybrid implementation	206
4.127 ORB OpenCV Detector	207
4.128 ORB OpenCV detection	207
4.129 BRISK OpenCV Detection	208
4.130 Device tree configuration	212
4.131 Register File - Direct Access Interface	212
4.132 Register File - RUNNING bit Interface Redirection	213
4.133 Register File - Register Mapping	213
5.1 create() Method	229
5.2 Platform-independent measurement of time with Boost	238
5.3 Platform-independent measurement of time with Boost	245
5.4 drawImage function	251
5.5 Software Refactoring Integration	252
5.6 Software Refactoring Integration Timing Outputs	255
A.1 Detector Testbench	278
A.2 Matching Testbench	283
B.1 Small 32x8 Image to Array Conversion Python Script	286
B.2 Real Image Snippet to Array Conversion Python Script	286

1 | Introduction

The first chapter introduces the problem statement, the motivation for the project, defines its goals and finally presents the structure of the overall document.

1.1 Problem Statement

A lot of factors have contributed to the revolutionising success of Computer Vision technologies. It is a sequential integration of four distinct processes, i.e. acquisition of images or visual stimuli from the real world in the form of binary data, image processing in form of edge detection, segmentation matching and lastly analysis and interpretation. From augmented reality games to self-driving cars to Apple's Facial Unlock feature, it has deeply impacted our life. And this influence is not free of consequences. However, on the flip side, it has been welcomed with generally encourage reviews.

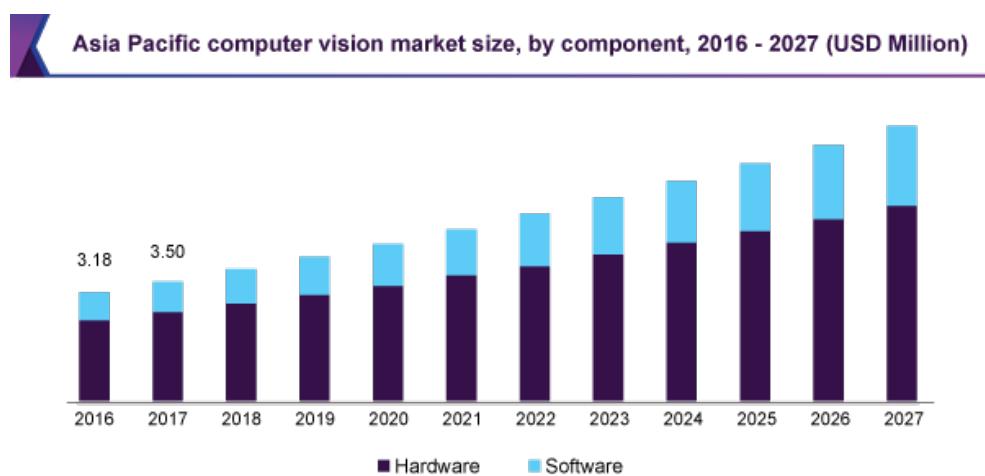


Figure 1.1.1: Expected market growth of Computer Vision Technologies

Hardware component led the market and accounted for more than 70.0 percent share of the global revenue in 2019. The high share is attributed to the availability of the latest hardware platforms that provide convenient component interconnection and advanced capabilities, such as fast processing, multi-mega-pixel resolution, and fully digital data handling. Hardware components for a computer vision system include processors and co-processors, cameras, lighting, and optical components, and frame grabbers. Also, high-performance hardware components have made the installation of vision systems easier and serve a broad scope of diverse applications by offering different networking architectures [6].

1.2 Context and Motivation

Almost all of the recent electronic applications need to strictly meet real-time constraints and can't withstand getting blocked in system bottlenecks. This calls for fast and deterministic controllers, that can mitigate the overall latency of the application and also make the system less prone to unpredictability [16]. A technique that works in that sense is computational offloading by hardware acceleration, where one profiles the application to find bottlenecks and

then proceeds to verify what can be accelerated, migrating to hardware the parts responsible for the choke points. Of course that, some refactoring is always allied with this technique because if application's bottlenecks can be avoided by refactoring the software there is no need to perform the offload since there is always the trade-off of different domain integration.

“ I've always believed that if you put in the work, the results will come. I don't do things half-heartedly.
Because I know if I do, then I can expect half-hearted results.

Michael Jordan

”

1.3 Project Goals

The project intends to create a programmable computer vision system resorting to hardware acceleration. After some research, two key goals were identified:

1. Develop a system that can identify and match two images;
2. Develop an interface between the Processing System (PS) and the accelerator via AXI;

1.3.1 To reach the first goal

It is necessary to study the feature detection and matching algorithms, as well as the outlier removal techniques, analysing and designing hardware around it and implementing improvements if feasible. Additionally, one should also study Domain-Specific Languages and profiling/benchmarking techniques. The system's accelerators should have as little latency as possible to at least ensure an equal performance with their software counterpart. The ideal is to have the accelerators outperforming the versions that exist within the software/embedded software domain, whilst being programmable through the DSL. The achievement of the goal depends on the proper profiling, benchmarking and refactoring of the software, proper simulation and deployment of the forementioned FPGA accelerators, and of the functioning of the DSL.

1.3.2 To reach the second goal

It is necessary to understand the AXI protocol and develop in a sense that doesn't overload the PS with requests. The achievement of the goal depends on whether or not the interface functions well and delivers the data in a stall-free manner.

1.4 Report Outline

This document presents the development of a hardware-accelerated programmable computer vision system, to use, as the name indicates, in computer vision applications with time constraints in terms of keypoint detection. The contents are divided into six chapters, with the present one being the first and regarding the introduction. The second chapter presents some topics that are the bedrock of the project and lay some background knowledge based on the current state of the Art. The report and work method follow the waterfall methodology (fig. 1.4.1), which is mainly comprised of the requirements and analysis phase, the design phase, the implementation phase and, lastly, of the testing phase. Regarding this, the second chapter also refers to the requirements and analysis phase, the third one to

the design, the fourth one to the implementation and the fifth one to the tests and result assessment.

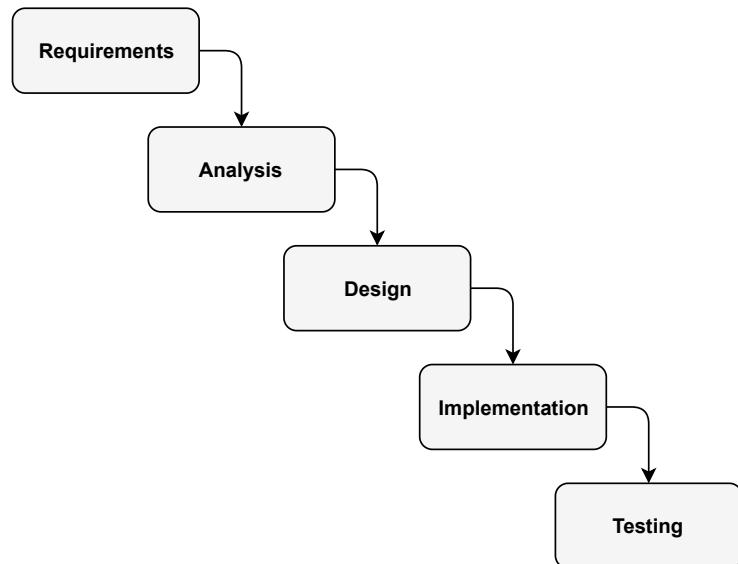


Figure 1.4.1: Waterfall Methodology

2 Analysis

In this phase, one intends to analyse the state of the Art and provide some background to topics that are the bedrock of the project. Additionally, its requirements and constraints will be presented as well, as a general system overview to better visualise the objectives. Additionally, the system is analysed in terms of its hardware architecture, proceeding with the presentation of various diagrams that allow further understanding of the overall system.

2.1 Background and State of the Art

As the main topic of the report is the development of an accelerator, this section will firstly approach current state of the Art hardware acceleration and address some development concerns. Secondly, all the computer vision algorithms will be explained in more detail, as well as HDLs, FPGAs, the AXI protocol, and profiling/benchmarking.

2.1.1 Hardware Accelerators

As stated before, hardware accelerators increase a system's speed n-fold, and in some cases improve the overall power consumption [2] at the expense of having less flexibility and upgradability. Independent of the scenario, migrating a block from software to hardware adds up to the system's complexity and, ultimately, can affect negatively and significantly the product's time-to-market if managed poorly. Hardware acceleration can be viewed as a modern technique that exploits the true parallel nature of the hardware and offloads critical points to free up CPU resources [16].

Blockingness. In terms of hardware accelerators, one of the factors to consider is its **blockingness**. A blocking call of the accelerator will leave the software idly waiting for the output and a non-blocking call will see it continuing execution and checking the result afterwards, as schematised in fig. 2.1.9. The negative effects a blocking call can be mitigated by the Operating System switching to another task in the meantime and its relative speed, which can make the slight wait time worth it [2].

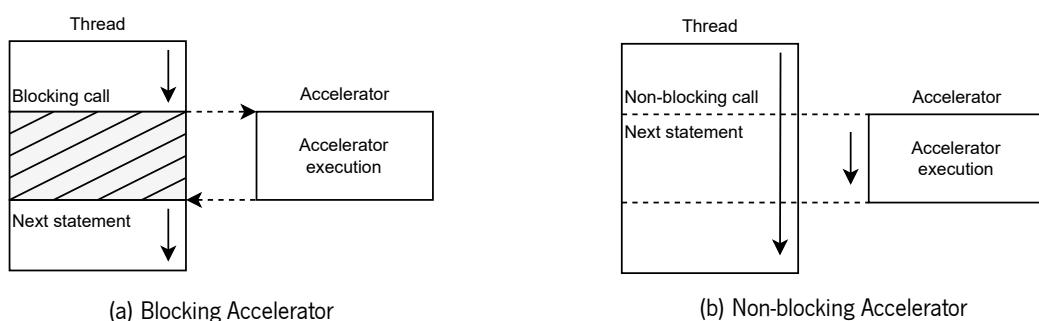


Figure 2.1.1: Overview of blocking and non-blocking hardware calls

Accelerator Sharing. Another important factor is whether the accelerator is dedicated to one processor or shared by various processors. As is the case with any other resource, sharing the accelerator makes its operation synchronisation-reliant. Of course, this only becomes problem in cases where the PL cannot finish execution before the OS picks up a new task.

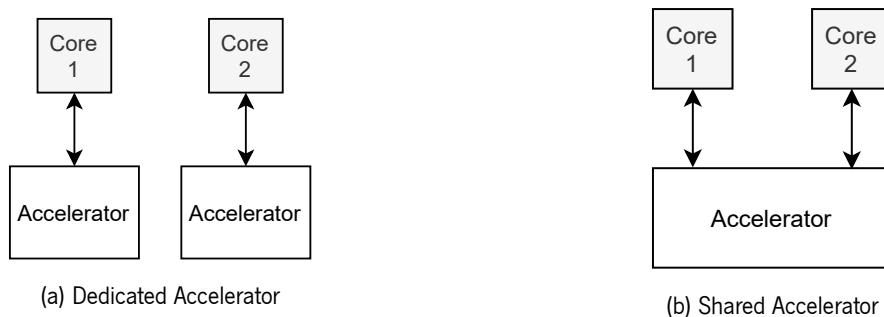


Figure 2.1.2: Overview of shared and dedicated accelerators

Multi-Processor Setups. Shared accelerators can work in setups where every part of the program can have direct access to every other globally declared part, which is known as a Synchronous Multi-Processing (SMP) setup. The opposite of this situation is an Asynchronous Multi-Processing (AMP) setup, where visibility over a certain area of memory between cores must be accomplished by communicating between processes. This is not implicated in the design of the accelerator, but it is very important to know when using it, since such circumstances can be detrimental to its **effectiveness**.

Data Copy. In processes involving the handling of large amounts of data, copying that data from the space controlled by the processor locally to the accelerator is often necessary. This sometimes done using the path of the system bus, shared with other peripherals.

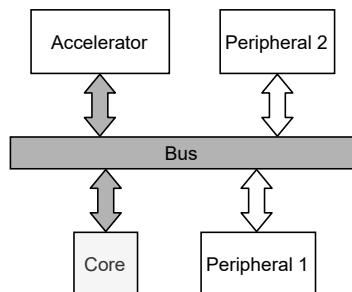


Figure 2.1.3: Common data copy between Processor and Accelerator

However, doing this each time it needs to be processed is an energy, space and time-inefficient approach, leading to latency-heavy operation and contention in the bus. Instead, **zero-copy** approaches are sometimes chosen, which introduce very little latency when compared to simply copying the data over. This is because very little information is necessary to describe the location and conditions of the information in memory in comparison to its actual contents. However, there are a couple of downsides to this, one of which is that the data must be kept intact during the operation, leading to contention if that condition is at risk. It may also not be possible to follow that approach due to risks concerning systems security and integrity, since *zero-copy* requires access to a certain space in memory that needs to be intermediated by a Memory Management Unit (MMU).

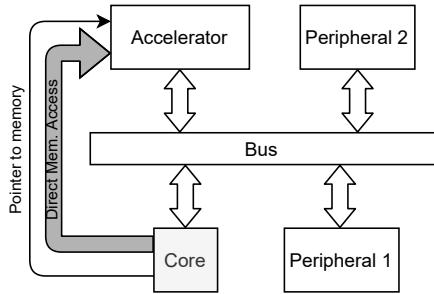


Figure 2.1.4: Zero-copy approach

Another common way to go about transferring data from one place to another is to have a system specifically dedicated for that purpose - a Direct Memory Access (DMA) controller. These machines allow the processor to offload the burden of copying the data to a peripheral, as they do it for them **in parallel**, also solving the bus contention issues. The major downside to this approach is the space that such a solution takes up.

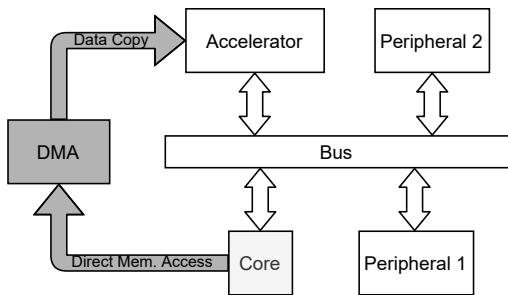


Figure 2.1.5: Data copy between Processor and Accelerator through DMA Controller

Hardware Interface. The interface between the Processing System (PS) and the Programmable Logic (PL) should be layered, for maximum flexibility and portability. At the lowest level, there is a bare-metal, hardware-driven component. In accelerators with even minimally complex function, a generic, register-driven interface, is usually adopted, to compactly accommodate every hardware control, status communication and parameter definition necessity.



Figure 2.1.6: Hardware Register Interface

Software Interface. At a software level, especially when working at a higher abstraction level and in more complex systems, it is generally undesirable to interact directly with the hardware interface. For offloading the application program from the details of the hardware interface, simplifying and encapsulating it, a device driver is needed. The device driver is then typically complimented with a suiting wrapper, which will provide a higher-level and more

distinctive interface, ensuring the best usability and maintainability. Most importantly though, should the driver suffer modifications, only the back-end of the wrapper will need to be modified to ensure that it can be used as before.

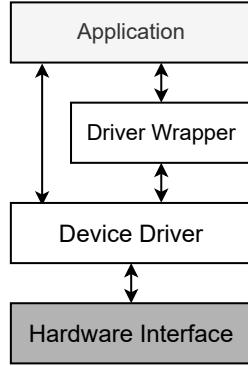


Figure 2.1.7: Software Interface

Signalling Events. When any relevant event takes place, hardware accelerators usually go about notifying the host system through firing a system interrupt, writing to a register or both. Of course, when exclusively writing to a register, the PS will have to poll the registers to be aware of such events. When going the interrupt route, however, an Interrupt Service Routine (ISR) is usually implemented in software, to be called only in the event of an interrupt, in a non-blocking fashion.

Coherency Management. In systems with multiple systems sharing common memory space, maintaining memory integrity is often a challenge. This is especially true in cached and multi-core systems, where maintaining coherency between cached data in multiple cores adds another layer to that challenge. Although this issue is largely understood and taken care of, it is important to understand also that any hardware accelerator in a cached multi-core system becomes another potential source of hazard towards memory coherency. It is then important in the analysis and design of the accelerator to understand how it will act as a player in that system and guarantee coherency when making changes to memory.

A more detailed illustration of how such objective may be fulfilled is presented in fig. 2.1.9b. It consists of the accelerator relying on an ISR to write the information to memory directly through the cache. The benefit of this approach is the simplicity in implementation, which must be balanced against the possibility of the accelerator depending on the completion of the cache invalidation process to carry on with execution.

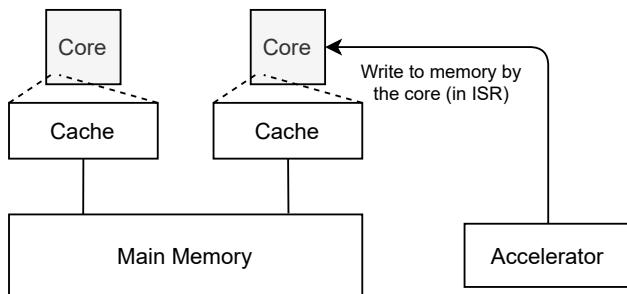


Figure 2.1.8: Memory Write handled through ISR

Alternatively, if blocking the accelerator's execution until the core is done writing the contents to memory isn't a viable option, the hardware could write directly to memory, invalidating the cache lines either through an existing Cache Coherency Infrastructure or through software, called from within the ISR.

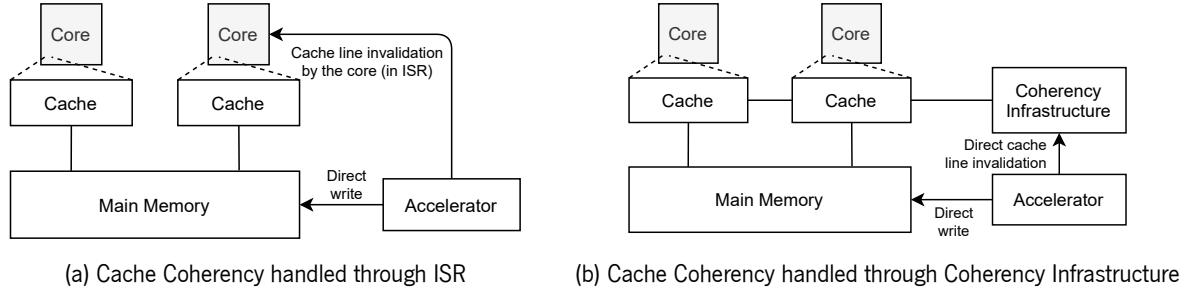


Figure 2.1.9: Overview of Cache Coherency handling when writing directly to memory

Accelerator Chaining. Complex hardware-accelerated tasks often require many stages of the same operation to be logically separated. These stages may often be able to be called independently and their results reported back to the calling processor core separately. This is typical in Video Processing, Machine Learning and Computer Vision applications. In order to simplify PL design and maximise flexibility, each stage could be called independently through software and return the results before the next stage is called.

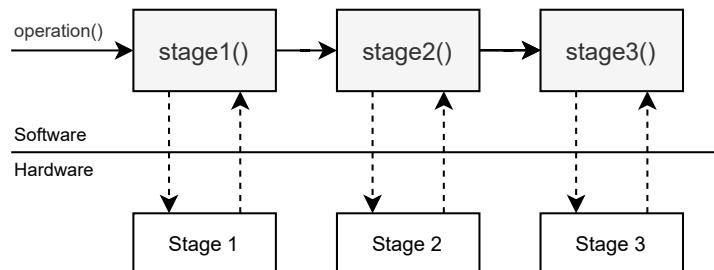


Figure 2.1.10: Independent Accelerator stages

However, in applications that mostly require the stages to be called one after another repeatedly, this is highly inefficient, as software execution is overall subject to much higher latency and lower bandwidth than hardware accelerators. As an alternative the different stages could be chained, executing the full operation in one go. Allowing all stages to be woven into the same unified pipeline helps drastically reduce the overall latency and increase the throughput of the process. A special entity could be employed as an interface between the hardware and the software, managing the flow of data from the individual stages to the processor.

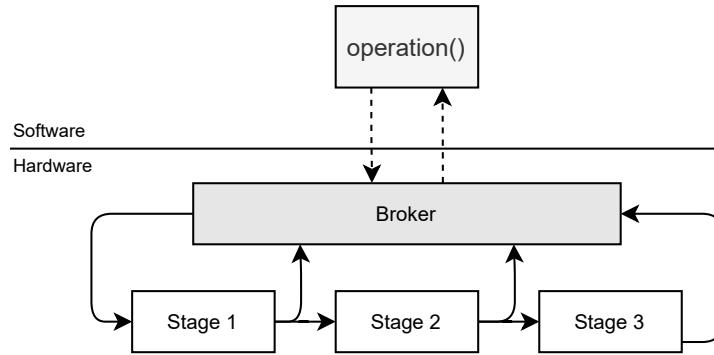


Figure 2.1.11: Chained Accelerator stages

Smaller Data Granularity. When working on large volumes of data, copying all the data over to the PL side increases the overall process latency and requires a large amount of duplicate storage space. What could be done instead is extracting only a small portion of the data each time. This would allow smaller bits of data to travel from stage to stage of the pipeline, meaning that the work on each stage starts sooner and thus the overall latency is reduced.

2.1.2 Floating-Points

The IEEE Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).

The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard which determines the format: $N = 1.F \times 2^{(E + bias)}$ where N is the floating-point number, F is the fractional part in binary notation and E is the exponent in bias representation. By biasing the exponent, the number stays within an unsigned range more suitable for comparison.

- For single-precision floating-point, exponents in the range of -126 to +127 are biased by adding 127 to get a value in the range 1 to 254;
- For double-precision, exponents in the range -1022 to +1023 are biased by adding 1023 to get a value in the range 1 to 2046.

In the 32-bit IEEE format, i.e., single-precision floating-point, 1 bit is allocated as the sign bit, the next 8 bits are allocated as the exponent field, and the last 23 bits are the fractional parts of the normalized number as illustrated in 2.1.12 .

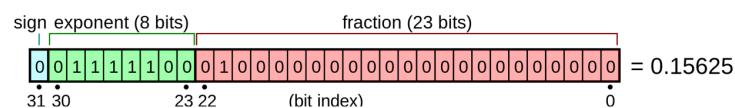


Figure 2.1.12: Single-precision (32-bit) form (bias = 127)

By arranging the fields in this way, so that the sign bit is in the most significant bit position, the biased exponent in the middle, then the mantissa in the least significant bits – the resulting value will actually be ordered properly for comparisons, whether it's interpreted as a floating point or integer value. This allows high speed comparisons of floating-point numbers using fixed point hardware.

This standard is very reliable however, there are some exceptions:

- E = 255; F = 0 - ± infinity
- E = 255; F != 0 - overflow, error.
- E = 0; F = 0 - zero
- E = 0; F != 0 - Denormalized, smaller than smallest allowed

2.1.3 Detection and Description Algorithms

This section will present all of the Computer Vision algorithms used in the project.

FAST-n

Features from Accelerated Segment Test (FAST) is a corner detection method, which could be used to extract feature points and later used to track and map objects in many computer vision tasks. The most promising advantage of the FAST corner detector is its computational efficiency. The FAST corner detector is very suitable for real-time video processing application because of this high-speed performance.

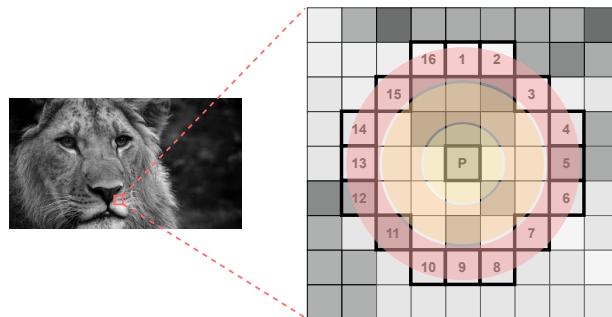


Figure 2.1.13: FAST-n: Detection Overview

FAST Detection. In the detection with FAST, pixels can be categorised into three classes:

$$S_{p \rightarrow x} = \begin{cases} d, & I_{p \rightarrow x} \leq I_p - t \\ s, & I_p - t < I_{p \rightarrow x} < I_p + t \\ b, & I_p + t \leq I_{p \rightarrow x} \end{cases}, \text{ darker, similar, brighter} \quad (2.1)$$

- $S_{p \rightarrow x}$ represents the state;
- $I_{p \rightarrow x}$ represents the intensity of the pixel x ;

- I_p represents the intensity of the central point;
- t represents the threshold value;

The algorithm to perform the detection involves the **Bresenham Circle** in which **a certain amount of pixels around a central pixel P** are selected. For a pixel to be considered a corner, n of the circle's pixels need to be all darker or brighter than the pixel on the centre. The number represented by the letter n in FAST- n denotes the amount of pixels that must fulfil that condition and the size of the circle. The different n -dependent circles are represented in fig. 2.1.14.

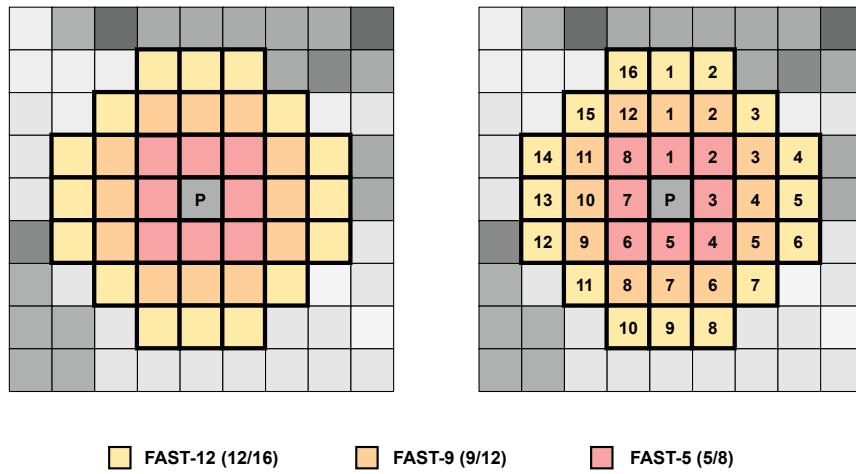
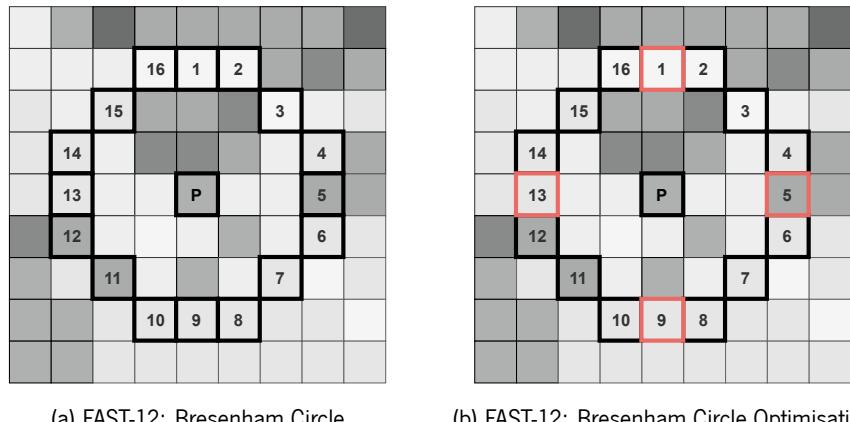


Figure 2.1.14: FAST- n : Bresenham Circle Possibilities

Optimisation. In fig. 2.1.15b, one can see an optimisation of the algorithm in which four points on the circle are compared, typically 1, 5, 9, and 13. If three of these points turn out to have an intensity value greater or lower than $I_p + t$ then the centre can be a corner, otherwise, one discards that pixel and proceeds to the next evaluation. This optimisation refers to FAST-12.



A simplification for the FAST keypoint detection algorithm is presented in algorithm 1.

Algorithm 1 Fundamental FAST Feature Descriptor implementation

```

1: for each  $i \in I_i$  do
2:   if  $iI_{i...i+12} + T < p$  or  $I_{i...i+12} - T > p$  then
3:      $p$  is a corner
4:   end if
5: end for

```

Scale Invariance. BRISK uses a method inspired by AGAST for achieving scale-space invariance, applying a FAST detector with the same threshold value and NMS to every layer in the scale-space pyramid. Each of these layers consists of n octaves and n intra-octaves. Each octave d_i obtained by downsampling the original image by a factor of 2^i and each intra-octave c_i by downsampling the original image by a factor of $2^i \cdot 1.5$.

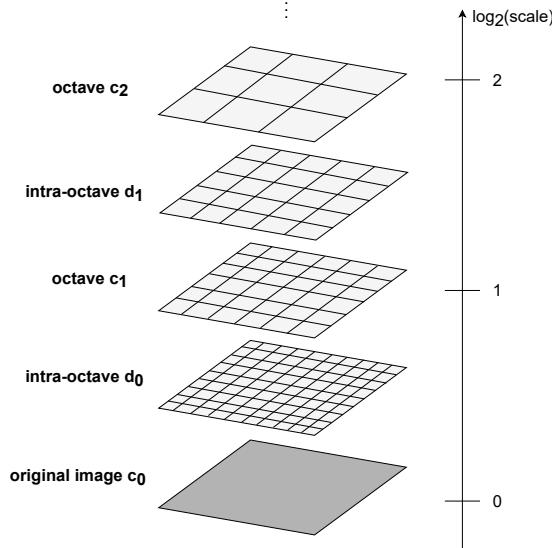


Figure 2.1.16: Scale-invariance in BRISK

FAST Scoring. Non-maximal suppression can not be applied directly to the resulting features of fast detector. Consequently, a score function, V must be computed between detected corner and non-maximal suppression. Applied to this to remove corners which have an adjacent corner with higher V .

There are several intuitive definitions for V :

- The maximum value of contiguous pixels for which pixel is still a corner.
- The maximum threshold for which pixel is still a corner
- The sum of the absolute difference between the pixels in the contiguous arc and the centre pixel.

V is given by:

$$V = \max \left(\sum_{x \in S_{bright}} \left(|I_{p \rightarrow x} - I_p| - t \right), \sum_{x \in S_{dark}} \left(|I_p - I_{p \rightarrow x}| - t \right) \right) \quad (2.2)$$

$$S_{bright} = \{x \mid I_{p \rightarrow x} \geq I_p + t\} \quad (2.3)$$

$$S_{dark} = \{x \mid I_{p \rightarrow x} \leq I_p - t\} \quad (2.4)$$

The score function can also be described in alternate ways. What is important is to define a heuristic function which can compare two adjacent corners and eliminate the insignificant one.

BRISK algorithm utilises a FAST-N detector to identify potential salient points in octaves and intra-octaves of scale-space pyramid.

The FAST detector also calculates FAST score S which is the sum of the absolute difference between the pixels in the contiguous arc and the centre pixel.

FAST Sparse Point Non-Maximum Suppression. In a computer vision algorithm, the detector outputs a large number of bounding boxes or key points, therefore it is necessary to pick the best ones. NMS is the most used algorithm for this task. In essence it is a form of clustering algorithm.

A proposed method for non-maximal suppression works without the concept of the 2D window which, despite working, is rather costly and since BRISK is performance focused we will attempt a method that has a better performance. It is split into 2 stages: Horizontal and vertical non-maximal suppression, the first involves finding the maxima along the x direction and the latter the maxima along the y direction, basically checking for neighbours $(x+1, x-1, y+1, y-1)$ on those directions and bypassing the need to slide a window through the whole image.

2D NMS. Here in fig. 2.1.17, we start by taking a small overview of the 2D non-maximum suppression requested in each octave. Then, we explain how the horizontal stage works inside, being that, after a vector reordering, the vertical one works exactly the same way.

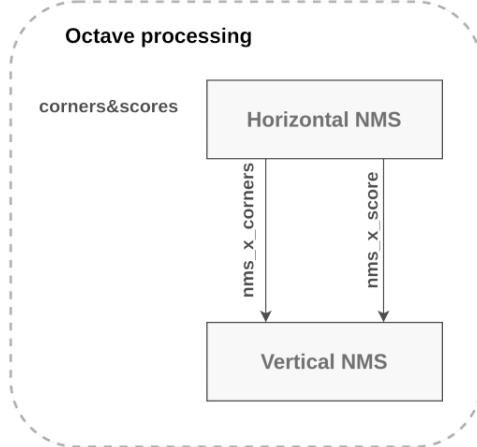


Figure 2.1.17: Octave processing

Horizontal NMS. Firstly, the inputs of the horizontal NMS are the corners and respective scores. In this context, by corner we understand a selected pixel and its position in the image, using Cartesian coordinates (X and Y).

In this stage, we find the maxima along the X direction. For every corner, we check if it has a neighbour in either the left or right direction and then compare their Fast scores to suppress the non-maximas. As it was mentioned earlier, it is important that the XY coordinates list of corners are in raster order. This would mean that the corners would be listed in buckets of Y, same Y but different X (as it is shown in figure 2.1.18).

KEYPOINTS XY	SCORE	
0x0303	Score33	
0x0403	Score43	
0x0803	Score83	
0x0404	Score44	
...	...	
...	...	

1	2	3	4	5	6	7	8
1							
2							
3		1	1				1
4			1				
5							
6							
7							
8							

Figure 2.1.18: Inputs general format.

Once we have the data in this format, we can easily make the assignments and start the verification. The central corner must follow the index counting, and the left and right corners are the next and previous vectorised pixels.



Figure 2.1.19: Assignment example.

The next steps consist of determining if a neighbour exists in the right or left by checking against (X-1,Y) and (X+1,Y) and also comparing the FAST scores accordingly to suppress the non-maximum corners along the horizontal direction. If one looks at fig. 2.1.20, it only takes one of this conditions to be false to decide not to suppress the central corner. On its time, figure 2.1.21 shows the fusion between the right and left verification outputs.

The procedure repeats itself for every line of the picture and the stage is completed, so we can move to the second part. At vertical non-maximum suppression stage, what changes is the way the vector is ordered, because it must then list corners in buckets of X. Once the score values are re-ordered, we can follow the same approach as in horizontal suppression to suppress the non-maximas in top and bottom direction.

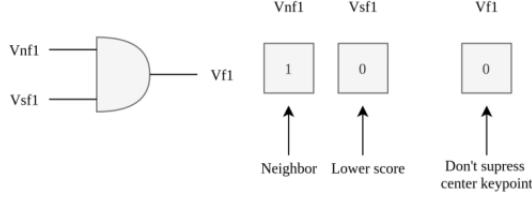


Figure 2.1.20: Conditions verification.

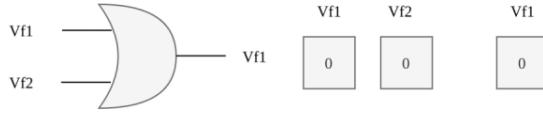


Figure 2.1.21: Stage outcome.

3D Refining. Figure 2.1.22, from [4], shows general block scheme of the proposed hardware architecture for BRISK keypoint detection. A processing layer receives required image pixels from input image or its preceding octave/intra-octave in a pipelined manner and generates windowed pixels to apply the Fast detector. Then, Fast scores are calculated for all pixels of the layer, and subsequently, the 2D and 3D maxima detection algorithms are applied to detect potential BRISK points in each layer. 3D maxima detection algorithm in a processing layer L_n requires scores of corresponding pixels in preceding and succeeding layers L_{n-1} and L_{n+1} . To this end, each processing layer communicates with preceding and succeeding layers using Score and Score Request (SSR) signals.

So, after a complete octave processing of the non-maximum suppression, a corner is only considered as such, when compared with the corresponding pixel from the closest layers. Resuming, if on those layers (below and above), the respective pixel score is lower, it is definitely a point of interest in the picture. Nevertheless, to reach that conclusion we first need to apply the interpolation factor to find the right pixel to compare.

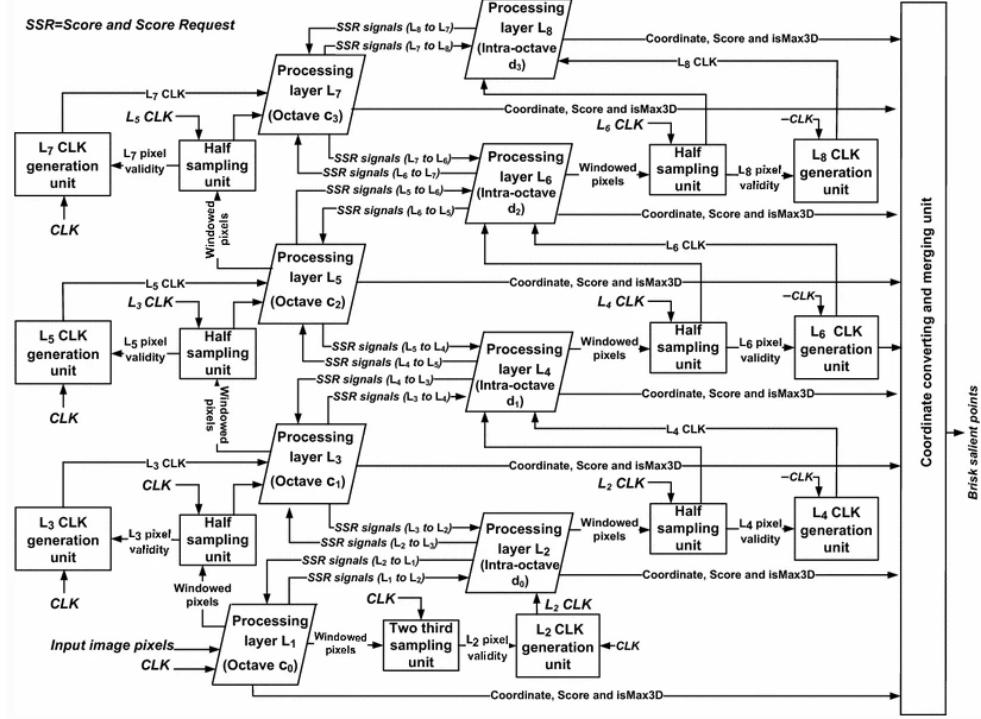


Figure 2.1.22: Processing layers.

ORB

Oriented FAST and Rotated BRIEF (ORB), is a widely used detection and description algorithm. It uses FAST-n and BRIEF, respectively, for each of those two stages. However, these methods are slightly modified to grant this algorithm a high degree of rotation inference (see fig. 2.1.27).

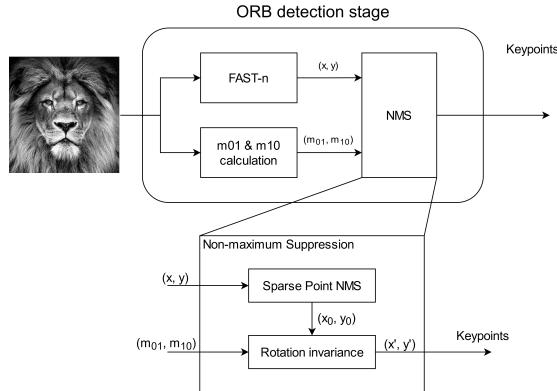


Figure 2.1.23: ORB detection stage overview

Image Momentum. This algorithm is employed by the detection stage, alongside FAST-n and can be thought of as the centre of mass of an image, where each pixel's "weight" is given by its brightness level. It can be calculated by 2.5 where the output m_{01} and m_{10} represent the centre of mass for each dimension.

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \quad (2.5)$$

A simple implementation of this formula can be done as follows:

Algorithm 2 m_{01} and m_{10} calculation implementation

```

1: for each  $i \in I_i$  do
2:   for each  $j \in J_j$  do
3:      $m_{01} = m_{01} + (j+1)patch[i][j]$ 
4:   end for
5: end for
6: for each  $j \in J_j$  do
7:   for each  $i \in I_i$  do
8:      $m_{10} = m_{10} + (i+1)patch[i][j]$ 
9:   end for
10: end for
```

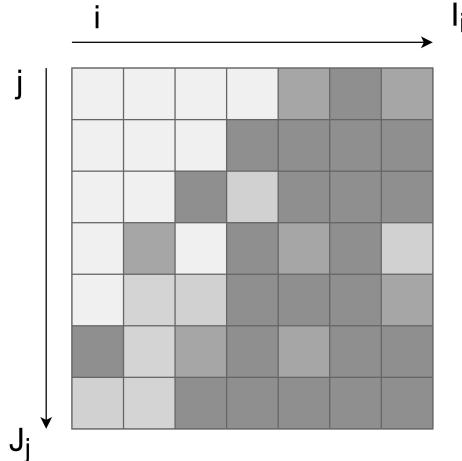


Figure 2.1.24: 7x7 pixel patch

Once the m_{01} and m_{10} values are found, the orientation of the patch can be determined and the respective offset applied, shifting the corner's coordinates to a new location. In fig 2.1.23, this would be implemented in the "rotation invariance" block.

$$\sin\theta = \frac{m_{01}}{\sqrt{m_{10}^2 + m_{01}^2}} \quad (2.6)$$

$$\cos\theta = \frac{m_{10}}{\sqrt{m_{10}^2 + m_{01}^2}}$$

$$x' = x\cos\theta + y\sin\theta \quad (2.7)$$

$$y' = y\cos\theta - x\sin\theta$$

BRIEF

When it comes to image descriptors, BRIEF is one of the fastest and most memory-efficient algorithms, but it is not rotation invariant (only two variants, rBRIEF and steered BRIEF are), which means that it is unable to measure and adjust the orientation of keypoints. Other image descriptors like SURF, which is a faster version of SIFT, rotate the image patch around the point of interest according to the main orientation of the patches before the descriptor is calculated. However, this kind of process is heavy since it involves bilinear interpolations for several pixels. The fig. 2.1.25 shows the difference in performance between the LAZY, BRIEF, and SURF descriptors.

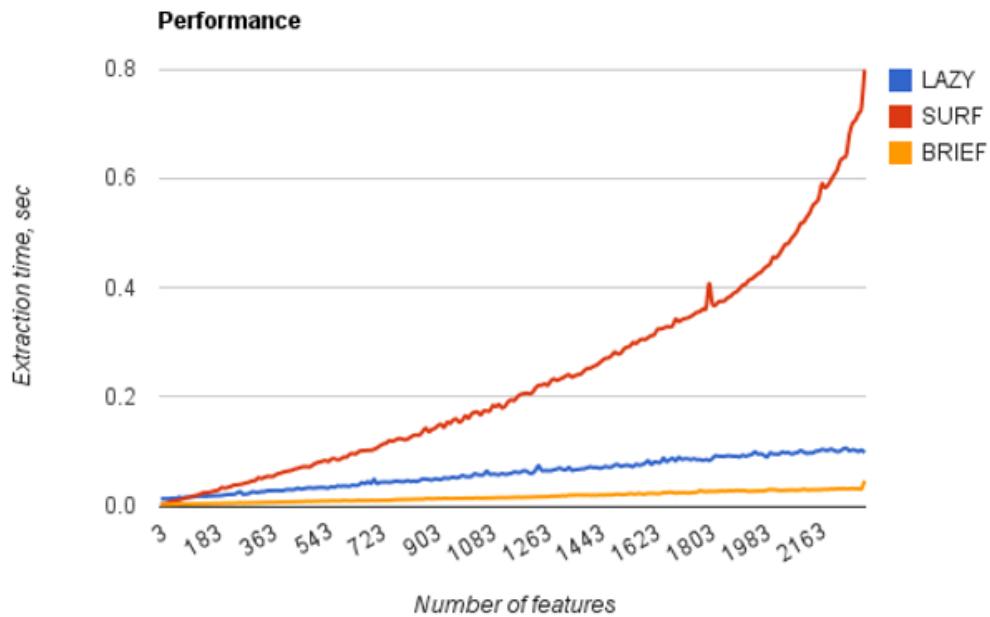


Figure 2.1.25: Performance of BRIEF against other descriptors

The essential difference between the **Binary Robust Independent Elementary Features (BRIEF)** descriptor and other algorithms is that each feature descriptor generated is accomplished with a binary string instead of a vector.

After the feature detection stage, a certain number of keypoints are presented, surrounded by an area of x by x pixels (known as patch). Since BRIEF takes the information of single pixels, there is a need to smooth the image using a Gaussian filter in order to reduce the noise-sensitivity, thus increasing the stability and repeatability of the descriptors. A binary descriptor is normally composed out of three different parts:

- **A sampling pattern:** to sample points in the region around the keypoint. In BRIEF case, it works together with the generation of the sampling pairs.
- **Orientation compensation:** a mechanism to measure the orientation of the keypoints and rotate them to compensate for rotation changes. The BRIEF algorithm, does not possess this capability.
- **Sampling pairs:** which sample points are going to be "merged" forming pairs allowing the comparison when building the final descriptor. BRIEF has the capability of having random or fixed sampling geometry.

Sampling Pattern. The pair generation can follow different approaches when sampling locations to select target points. These methodologies are based on sampling geometries, where each geometry selects its own spatial arrangement according to its algorithm. There are five different geometries used in these cases:

- (X, Y) **1_Uniform**($-\frac{S}{2}, \frac{S}{2}$): The (x_i, y_i) locations are evenly distributed over the patch and can select pairs close to the patch border.
- (X, Y) **2_Gaussian**($0, \frac{S^2}{25}$): The tests are sampled from an isotropic Gaussian distribution.
- **3_X_Gaussian**($0, \frac{S^2}{25}$), **Y_Gaussian**($0, \frac{S^2}{100}$): The first location x_i is sampled from a Gaussian centred around the origin while the second location is sampled from another Gaussian centred on x_i , forcing the locations to be more local.
- **4_**The (x_i, y_i) are randomly sampled from discrete locations of a coarse polar grid introducing a spatial quantisation.
- **5_xi = (0, 0)^T** takes all possible values on a coarse polar grid containing n points.

Considering the size of a patch p is $S \times S$, and assuming the keypoint is located in the centre of the patch, we choose one out of five possible sampling geometries in order to spread the different pairs inside the patch (fig. 2.1.26).

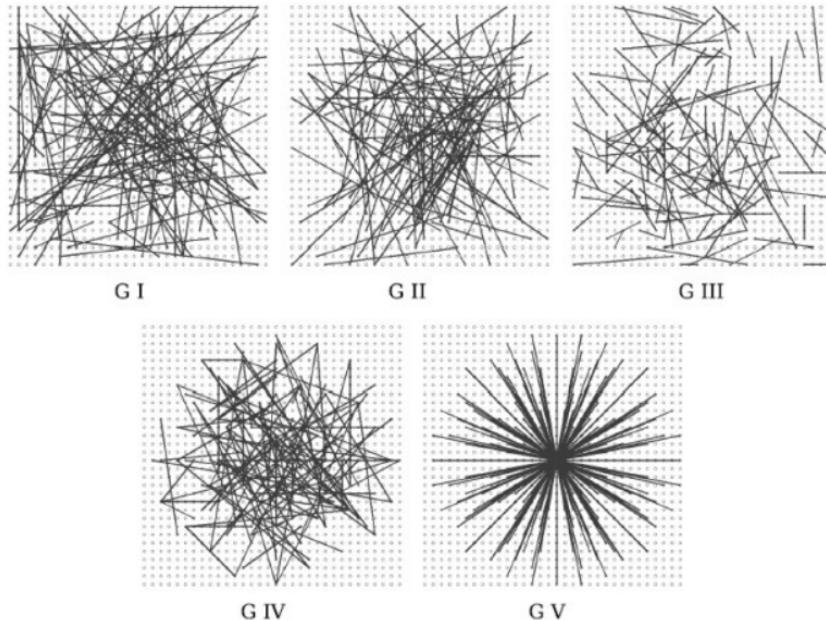


Figure 2.1.26: Different approaches for pair sampling

Orientation compensation in BRIEF. As it was previously stated, BRIEF is not rotation invariant, that means that it is incapable of measuring and adjusting the orientation of the keypoints. Such behaviour is desirable in the case of object detection, because the relative pose of objects to the camera is unknown.

Many image descriptors, like SIFT, rotate the image patch around an interest point according to the patches main orientation, before the descriptor itself is calculated. Therefore, the descriptor algorithm is not affected by the main orientation, only its input changes. However, this is a costly process since the patch rotation requires bi-linear interpolation for many pixels.

As we can observe in fig. 2.1.27, BRIEF has a huge disadvantage when compared to its competitors. If the image or the object is rotated more than 30 degrees, the recognition rate decreases by a landslide, dropping from almost 100 percent to approximately 10 percent.

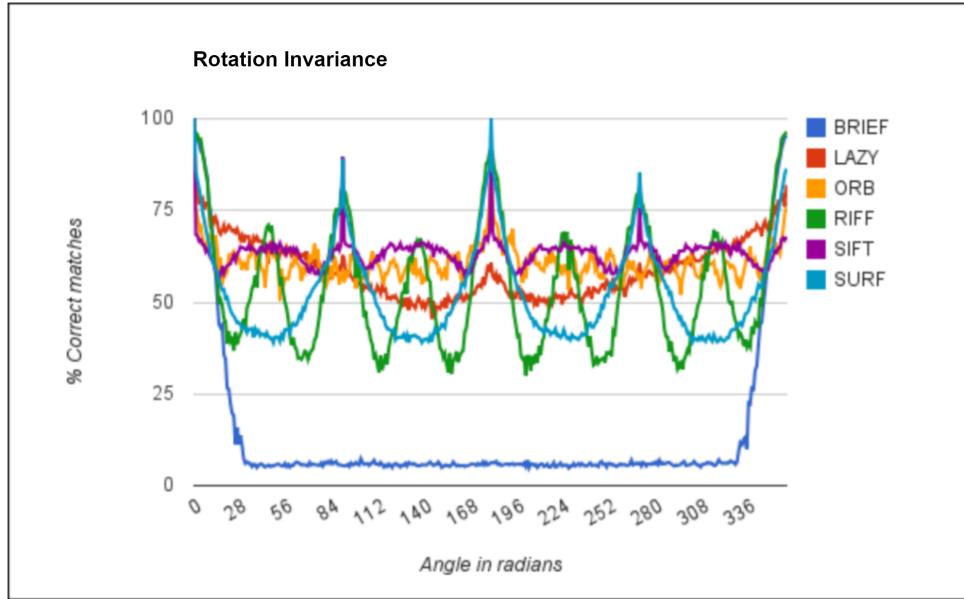


Figure 2.1.27: Rotation invariance between BRIEF and other descriptors

Sampling Pairs. The Binary Test is the last stage of the BRIEF descriptor, and its where the generation of the descriptor binary string occurs. This stage comes after the pair generation, where all the patch-pairs have been selected.

The binary test can be described with the following equation:

$$\tau(p; x, y) := \begin{cases} 1 : & p(x) < p(y) \\ 0 : & p(x) \geq p(y) \end{cases} \quad (2.8)$$

Where τ is defined as the binary test, p represents the filtered image patch and $p(x)$ and $p(y)$ means the intensities of point at location x and point at location y of a pair of points. This means that each bit of the binary vector represents the result of the comparison between the intensities of a pair of points and if the value of $p(x)$ is less than $p(y)$, then the comparison outputs the value 1, otherwise the output is 0. This can be seen in fig. 2.1.28.

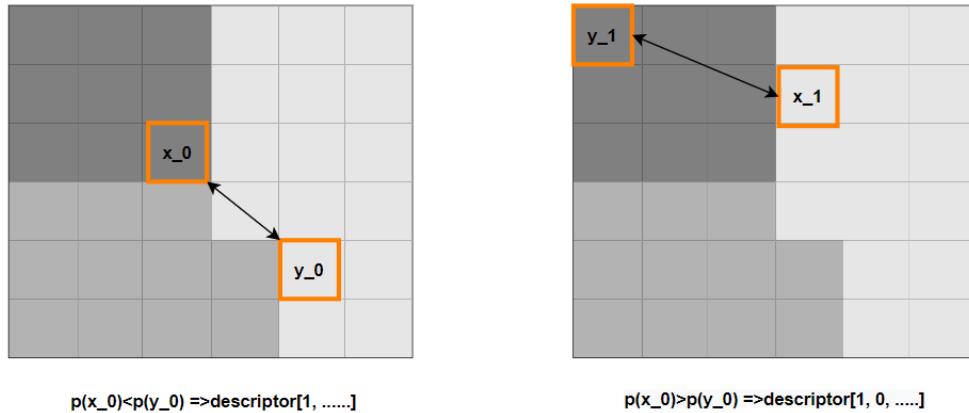


Figure 2.1.28: Example of sampling pairs

Above, two binary tests are performed on the two first pairs. On the first comparison the intensity of point y_0 is higher than the intensity of point x_0 , so for that pair the corresponding bit in the descriptor string will be the binary value of 1. For the second test, since the intensity value of point x_1 is higher than y_1 , the corresponding bit in the descriptor string will be the binary value of 1.

By choosing a set of $n(x, y)$ location pairs, which are received from the pair generation module, a set of binary tests is defined. Having selected the pairs, the BRIEF descriptor will be a bit-string of dimension n , described by the following equation:

$$f_n(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i; y_i) \quad (2.9)$$

The descriptor length to be considered is $n = 64, 128$, or 256 , and depending on the size of the descriptor, the same number of comparators will be used in parallel to perform the binary test.

2.1.4 Matching algorithms

Concerning the matching phase of the system, there are several algorithms that can be used. The ones that will be considered and discussed in this section are the Brute-Force and the FLANN algorithms, which can incorporate techniques such as cross checking and KNN, along with some outlier removal methods.

FLANN

FLANN (Fast Library for Approximate Nearest Neighbors) is a library for performing fast approximate nearest neighbor searches in high dimensional spaces. It contains a collection of algorithms we found to work best for nearest neighbor search and a system for automatically choosing the best algorithm and optimum parameters depending on the dataset.

FLANN is written in the C++ programming language. It can be easily used in many contexts through the C, MATLAB and Python bindings provided with the library.

FLANN can be used in these different applications:

- Cluster analysis
- Pattern Recognition
- Statistical classification
- Computational Geometry
- Data compression
- Database

The FLANN library includes the following algorithms:

- Linear
- Kd-trees
- K-means
- Composite
- Hierarchical
- LSH

For this specific case, one is gonna focus in the cluster analysis. Clustering algorithms are unsupervised algorithms that give a set of objects groups. It groups them in such a way that objects in the same group – cluster – are more similar to each member of the group than to those in other groups. Cluster analysis is not a specific algorithm, but a general task, therefore it can be achieved by various algorithms. These ones vary from each other in their understanding of what constitutes a cluster and how to find them. An example of clustering can be seen in the following figure 2.1.29.

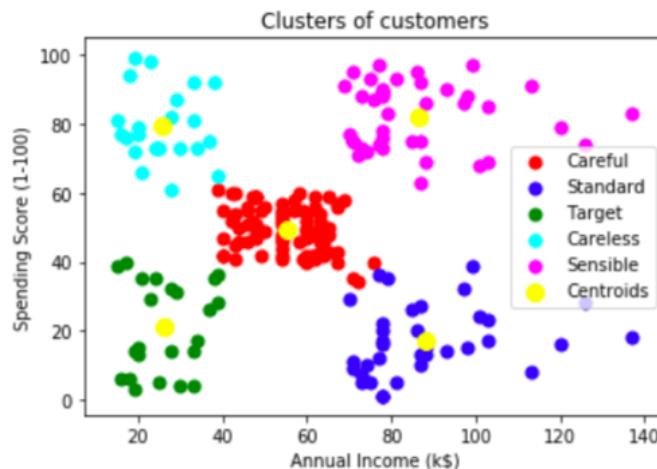


Figure 2.1.29: Clustering Example

To perform matching one would have to compare one resulting set of clusters with the other. This task may seem easy but there is a various amount of ways to resolve this issue, each one with a different level of accuracy and speed from the other.

Some of the most usual methods are:

- *set matching based comparisons* - this one being a greedy best effort 1 - to - 1 matching between clusters.
- *pair counting methods* - each pair of items is ascribed to a category out of 4. The sizes of the 4 classes can be used in several always to compute their resemblance including the Chi square coefficient, the Rand Index,etc.

- *information theoretical method* - it computes the variation of information between the 2 clusters.
- *Optimal transportation based methods* - these aim at mapping the clusters with one another and also at accommodating the case of soft clustering. It involves the distances between the clusters representatives (centroids) and solves for the weight assigned to the match between the 2 clusters.

Brute-Force

Brute-force is the other matching algorithm considered to resolve the problem at hand. It is a feature matching algorithm which compares all descriptors from a training set to each descriptor of the query set of descriptors in order to find the closest match for each one of them. This method of finding matches by exhausting all the other possibilities is what gives it its name. For this reason, this algorithm is considered more accurate and better for smaller datasets, in contrast to FLANN.

A method of Hamming distance calculation may be considered for algorithms with bit-string descriptors, such as BRISK or ORB, while Euclidean distance is preferred for SIFT or SURF [13]. As such, this algorithm is pretty straightforward and a representation of how it works can be seen in figure 2.1.30.

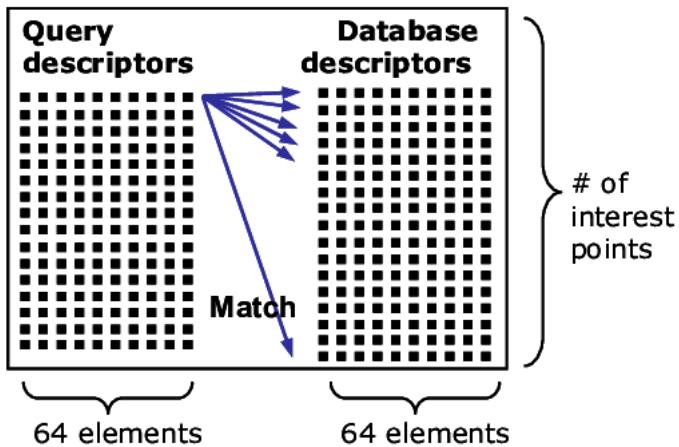


Figure 2.1.30: Brute-force algorithm

Complementary techniques

Cross-checking. To allow for more consistent feature matching, some brute-force matching algorithms also employ cross-checking. This algorithm states that a match is only a match if the descriptor of a given feature in one data set is the best match for the descriptor of a feature in the other data set, and vice-versa. This can be better understood by analysing 3, which implements the cross checking algorithm with brute-force.

Analysing the first procedure, which employs the brute force algorithm, one can observe that the first 2 arguments will be the descriptors that come from the previous stage. In the general brute-force algorithm, each descriptor in the first set will be compared with all descriptors in the second set of descriptors.

This comparison is performed through the match procedure that returns the minimum hamming distance and the index of the descriptor with the best distance. To perform cross-checking, it is necessary to verify that a descriptor from the second set, with the best distance for a given descriptor from the first set, also has as its best match, this said descriptor from the first set, thus validating the cross checking algorithm.

Algorithm 3 Simple Brute-Force Matching Implementation with cross checking

```

procedure brute_force(query_set, training_set, cross_check)
  for i = 0, . . . , length of query_set do
    match  $\leftarrow$  MATCH(query_set[i], training_set)
    if cross_check = true &
      MATCH(training_set[match.nearest_neighbour], query_set)
      .nearest_neighbour = i then
        append match to matches
    end if
  end for
end procedure

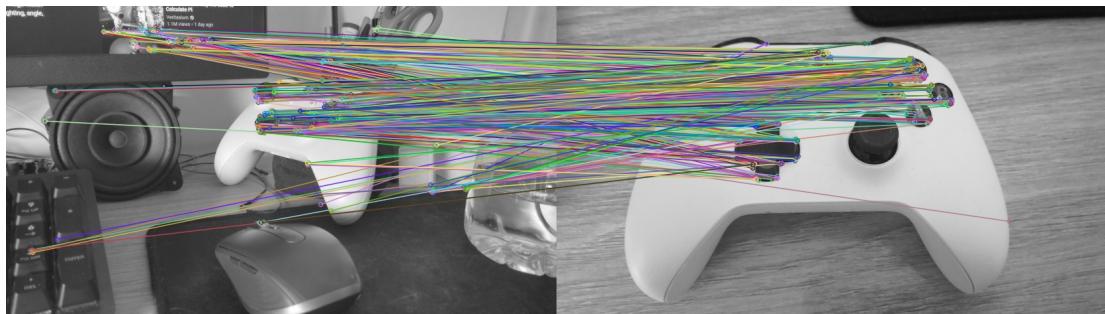
procedure match(desc, desc_set)
  for i = 0, . . . , length of desc_set do
    d  $\leftarrow$  distance(desc, desc_set[i])
    if d < closest_distance then
      closest_distance  $\leftarrow$  d
      nearest_neighbour  $\leftarrow$  i
    end if
  end for
  return closest_distance, nearest_neighbour
end procedure

```

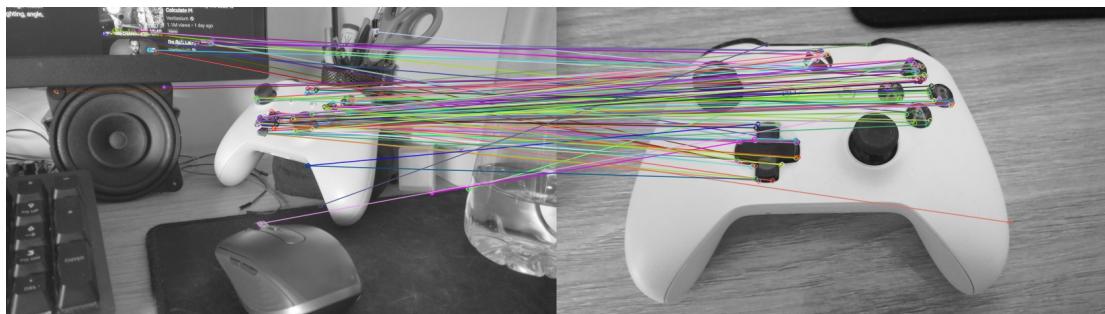
An example of how cross checking can influence the results of the matching phase can be seen in the following figure 2.1.31.

K-nearest Neighbour. This is a supervised machine-learning classification algorithm that stores available cases and classifies new cases based on their similarity which is measured through a distance function.

In this algorithm a case is classified by a "majority vote" with it being classified to the class most common amongst its K nearest neighbours measured by a distance function. In that sense, the brute-force matching algorithm presented before, which will be the algorithm chosen for the matching phase of this system going forward, can be refactored to include any *k* number of nearest neighbours. This is exemplified in figure 2.1.32 and in algorithm algorithm 4.



(a) Brute-force matching of two images without cross-checking



(b) Brute-force matching of two images employing cross-checking

Figure 2.1.31: Comparison of brute-force matching with and without cross-checking

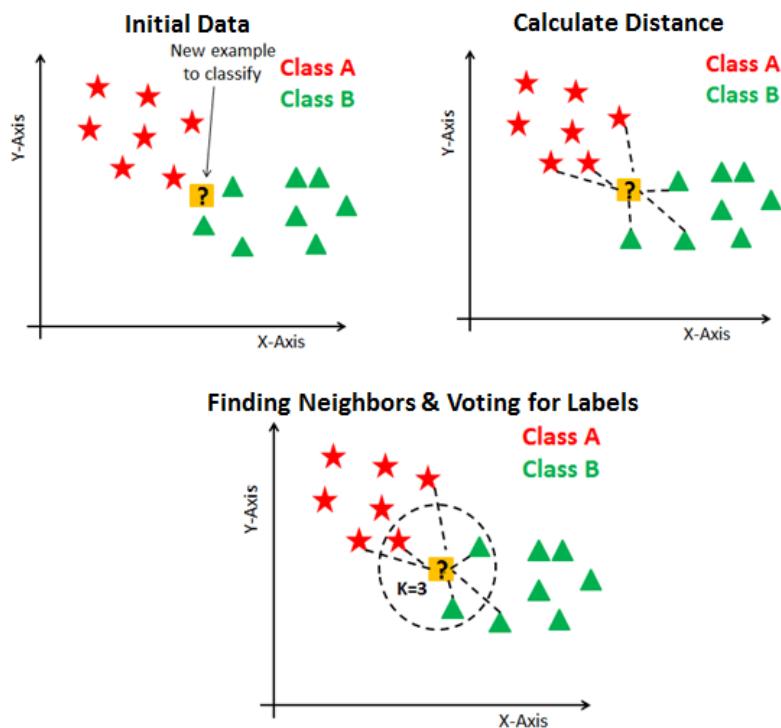


Figure 2.1.32: Brute force matching using kNN

Algorithm 4 Simple Brute-Force Matching (kNN) Implementation

```

1: procedure brute_force(query_set, training_set, k, cross_check)
2:   assert  $0 < k <$  length of training_set
3:   assert !( $k > 1$  & cross_checking = true)
4:   for i = 0, . . . , length of query_set do
5:     neighbours  $\leftarrow$  MATCH(query_set[i], training_set, k)
6:     if cross_check = true then
7:       _neighbours  $\leftarrow$  MATCH(training_set[neighbours[0].index],
8:                               query_set, k)
9:     if _neighbours[0].index = i then
10:      continue
11:    end if
12:  end if
13:  append i and neighbours to neighbourhoods
14: end for
15: return neighbourhoods
16: end procedure

1: procedure match(desc, desc_set, k)
2:   for i = 0, . . . , length of desc_set do
3:     d  $\leftarrow$  distance(desc, desc_set[i])
4:     append i and d to neighbours
5:   end for
6:   sort neighbours
7:   return neighbours[0 : k - 1]
8: end procedure

```

This technique can sometimes be interpreted as one that adds more outliers, which is something that one does not expect for these kind of complementary techniques. But, if a outlier remover method, such as Lowe's ratio (which will be explained in the section below), is added with KNN, the accuracy can be highly improved. The result should look as represented in fig. 2.1.33, for $k = 2$, using Lowe's ratio test with $ratio = 0.8$. One visible improvement over using $k = 1$ was the disappearance of the matches associating the edges of the buttons in the controller with letters on the screen. It also reduced the total number of matches to process.

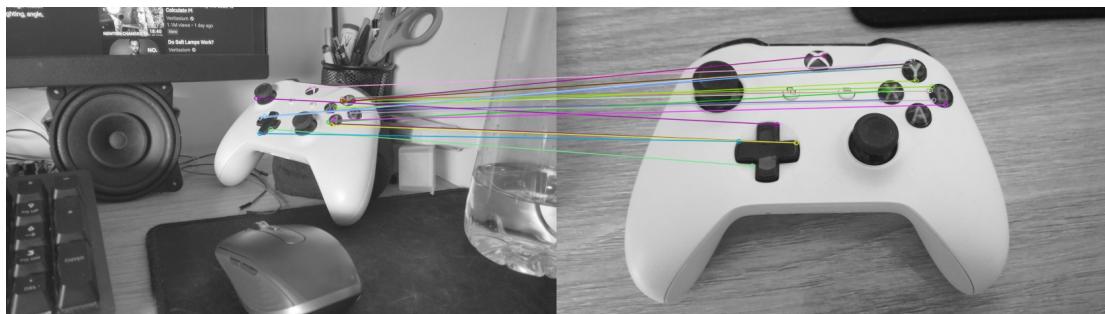


Figure 2.1.33: Brute-force matching using $k = 2$ and outlier removal

2.1.5 Outlier Removal

The outlier removal is the step that consists of removing the false positives, on the words, wrong information or at least information that is not relevant to the processes. There are many methods to remove said information and a few of the most promising, as far as accuracy, time and resource consumption is concerned, will be laid out in this section.

An outlier can be understood as a false match, i.e., even though the Brute-Force algorithm finds the best match for each one of the data set's features, the matches may not always be a realistic match. The simplest way to avoid outliers is to sort every match by distance and consider only the ones who have a distance lower than a determined threshold. However, there are also more efficient alternatives to remove outliers that can fit better in some applications' demands.

Lowe's ratio test

The threshold method has its problems because two features can have a small distance measurement due to most of the variables inside of their descriptors possess similar values, but those variables might be irrelevant to the actual matching. What Lowe proposes is to match every feature in the first data set with the two closer features in the second data set. Considering there is only one valid match, the "good" match is the one with the smallest distance, being the other match considered random noise. However, if the distance of the best match is within ratio times the second best match, it is considered that the best match is not viable and both of the matches are rejected. The ratio must be a number between 0 and 1, the filter being harsher as it gets closer to 1. An algorithm example can be seen in 5.

Algorithm 5 Lowe's Ratio Test Implementation

```

1: procedure ratio_test(bestmatch, secondbestmatch)
2:   if bestmatch.distance < (secondbestmatch.distance × ratio) then
3:     append bestmatch to matches
4:   end if
5: end procedure
```

RANSAC

The random sample consensus is an iterative method used to robustify the matching by fitting the matched features to a mathematical model of the transformation from one data set to the other. RANSAC tries really hard to identify a large set of matching features that are acceptable, in the sense that they agree with each other on supporting a particular value of the model.

As far as FPGA implementation goes however, RANSAC is known for using far too many resources and processing power along with taking a long time to process.

Slope-based Rejection algorithm

This rejection algorithm evaluates the slope between point correspondences [7]. When finishing the detection and matching of feature points in two adjacent images, the lines between point correspondences have the similar slope. Based on that, the points that presents strange slopes are rejected. The correct point correspondences have the same or similar slopes (such as L1, L3, and L4), while the slopes of falsely point correspondences will be different (such as L5). All this is present in the following figure 2.1.34.

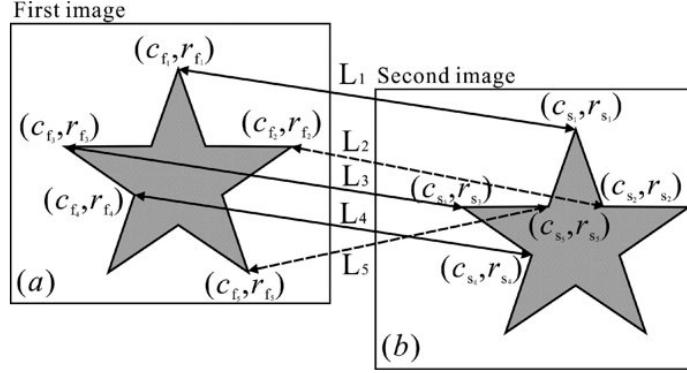


Figure 2.1.34: Example of Point correspondences with outliers

The formula for the slope is given as follows:

$$K_i = \frac{r_{si} - r_{fi}}{c_{si} - c_{fi}} \quad (i = 1, 2, \dots, n) \quad (2.10)$$

Where (c_{fi}, r_{fi}) and (c_{si}, r_{si}) are the column and row of point correspondences in the first image and the second image (one image pair), respectively, and k_i is the slope of the line that connects the (c_{fi}, r_{fi}) and (c_{si}, r_{si}) , 'f' and 's' mean the first image and the second image.

Correlation-coefficient-based rejection algorithm

This algorithm was created to make up for a failure of the previous, slope-based rejection algorithm, which was not capable of rejecting false points with similar slopes [8] (an example of this is the L2 point in figure 2.1.34).

One of the basic conclusions that can be taken and is, in fact, at the very core of matching is that a correct point correspondence implies that the descriptor of the points are similar and therefore the sub-images are related to each other; this is not what occurs with false point correspondences and can be used to remove them, through the **correlation coefficient**:

$$R = \frac{n \sum_{i=1}^n I_{fi} I_{Si} - \sum_{i=1}^n I_{fi} \sum_{i=1}^n I_{Si}}{\sqrt{n \sum_{i=1}^n I_{fi}^2 - (\sum_{i=1}^n I_{fi})^2} \sqrt{n \sum_{i=1}^n I_{Si}^2 - (\sum_{i=1}^n I_{Si})^2}} \quad (2.11)$$

I_{fi} and I_{Si} are the intensity values of the first and second image respectively and n is the number of I_{fi} and I_{Si} . Since square root calculations in an FPGA waste too much time the formula must be converted so that they are avoided:

$$R^2 = \frac{(n \sum_{i=1}^n I_{fi} I_{Si} - \sum_{i=1}^n I_{fi} \sum_{i=1}^n I_{Si})^2}{(n \sum_{i=1}^n I_{fi}^2 - (\sum_{i=1}^n I_{fi})^2)(n \sum_{i=1}^n I_{Si}^2 - (\sum_{i=1}^n I_{Si})^2)} \quad (2.12)$$

Two sub-images that are centered on a single point are selected for computing R^2 , if this coefficient is greater than a given threshold t , then it is considered a correct match, otherwise it is considered an outlier.

2.1.6 DSL

A domain-specific language, abbreviated to DSLs, is a computer language specialised to a particular application domain, as opposed to a general-purpose language, which is broadly applicable across domains. There are a wide variety of DSLs, such as, HTML, JSON, etc. Language-oriented programming considers the creation of special-purpose languages for expressing problems as standard part of the problem-solving process.

Creating a domain-specific language (with software to support it), rather than reusing an existing language, can be worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing language would allow and if the type of problem in question reappears sufficiently often. Pragmatically, a DSL may be specialised to a particular problem domain, a particular problem representation technique, a particular solution technique, or other aspects of a domain. A DSL uses the concepts and rules from the field or domain, focusing on solving a task related to a topic in a fast and easy way, when comparing to a typical GPL. Language-oriented programming defends the usage of multiple DSLs to solve different tasks, masking the intricacies of each problem. Those tasks can be the creation of a webpage (HTML), managing databases (SQL), or even document preparation with the LaTex domain specific language.

Resuming, DSLs are not meant to provide features for solving all kinds of problems, but if the domain of the problem is specific, it is easier and faster to solve it by using the DSL instead of a GPL. A program or specification written in a DSL can then be interpreted or compiled into a GPL. In other cases, the specification can represent simple data that will be processed by other systems.

Generative Programming. A DSL works in a distinct way from the General Purpose Languages (GPLs). The DSL compiler usually produces code written in a GPL. The low level compilation is left to the compiler of the target GPL, (which will be C++ and Vivado for this project). A DSL compiler defines some mapping of the high-level information and features of a DSL into the target GPL and underlying layer. [26]

Our methodology for the DSL compiler development is to translate the logic of the program into a GPL representation that is suitable to generative programming approaches. According to what was said previously, the DSL compiler converts the high-level information of a DSL into a lower-level target GPL, meaning that the final compilation into machine code becomes the responsibility of the underlying target language compiler. It can be defined as a metaprogramming technique since our DSL engine will treat other programs as their input data. In this project, automatic programming will be used to generate the final output code according to the inputs introduced by the user in the DSL script and the variability points inside the code repository.

Summing up, generative programming is a concept whereby programs are written to manufacture software components automatically. The goal of generative programming is to improve a programmer's productivity.

Internal DSLs (also called Embedded DSLs), are languages written inside an existing host language. They reuse the grammar and the parser of those host languages, exploiting available extension options of the host language, giving them the feel of a particular language in the form of an API.

External DSLs have their own custom syntax instead of being built on top of a language. Since external DSLs do not depend on any host language, they can be much richer in syntax and expressivity. In our project, we opted for the external DSL implementation, recurring to the Xtext software framework.

Xtext and Xtend. The main source of material used to learn how to develop a DSL will be the book "*Implementing domain-specific languages with Xtext and Xtend*". [25] This book describes all the necessary tools to create a DSL, such as the one intended for this project. The focus of this book is the use of Xtend/Xtext integrated with the Eclipse IDE, to simplify the implementation of the DSL. Xtext is the one-stop solution for building DSLs, and is widely used in industry and research. It not only lets you define a parser, but also provides you with a full environment, including rich IDEs and text editors to support your DSL.

Eclipse IDE Integration. Nowadays, with the emergence of powerful IDEs, such as Eclipse, programmers are deeply accustomed to these types of tools, because it helps in increasing productivity and making it more user-friendly. For this reason, a DSL should be shipped with good IDE support, since it will increase the likelihood of DSL adoption and success in the long run. Some of the most important points where an IDE should support a DSL are: syntax-aware editor, incremental syntax checking, suggested corrections, and auto-completion. With these conveniences, the interaction with the DSL environment becomes more pleasurable and easy to use. The chosen IDE was the Eclipse IDE since it eases the integration with other frameworks, as Xtext, and is becoming an industry standard. Here are some of its main features related to DSL IDE integration.

- Syntax Highlighting gives the ability to see the program coloured and formatted with different visual styles according to the elements of the language: comments, keywords, strings, and so on. It also gives immediate feedback concerning the syntactic correctness of what the user is writing.
- Background validation offers to the programming environment the ability to continuously check the program in the background while the programmer writing the code.
- Content assist is the feature that automatically, or on-demand, provides suggestions on how to complete the statement/expression. The proposed content should make sense in that specific program context to be effectively useful. In Eclipse, the content assist can be accessed with the keyboard shortcut.
- Error markers this feature allows for highlighting the parts of the program that have errors detected by the DSL parser, underlining those pieces of code in red. It also shows error markers with an explicit message on the editor in correspondence to the lines with error.
- Quickfixes offers suggestions to the user, so that he can fix his errors in the written code.
- Hyperlinking is a feature that makes it possible to navigate between references in a program. Our DSL provides declarations of any sort and a way to refer to them. It should also produce hyperlinking from a token referring to any declaration. Using this feature is possible to directly jump to the corresponding declaration.
- Outline is used for large programs, and it is helpful to have an outline showing only the main components. When the programmer clicks on an element of the outline, this feature brings him directly to the corresponding source line in the editor, that corresponds to the chosen component.

Xtext. Xtext is an open-source software framework for the development of programming languages and domain-specific languages. Xtext allows a developer to implement languages quickly by covering all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a full Eclipse IDE integration, with all the typical IDE features such as editor with syntax highlighting, code completion, error markers, automatic build infrastructure, and so on. Unlike standard parser generators, Xtext generates not only a parser but also a class model for the abstract syntax tree.

To sum up, Xtext simplifies the implementation of a DSL by abstracting the developer from some compilation states and the design of the Abstract Syntax Tree.

Xtend. Xtend is a statically-typed programming language which, unlike most languages that run on the JVM, does not generate byte code, but Java code. Syntactically and semantically Xtend has its roots in the Java programming language but improves on many aspects, such as extension methods, lambda expressions, type inference, etc.

Xtend has zero interoperability issues with Java: Everything written interacts with Java as expected. At the same time, Xtend is much more concise, readable, and expressive. Xtend's small library is just a thin layer that provides useful utilities and extensions on top of the Java Development Kit (JDK).

An advantage of Xtend is that the language is already fully integrated with the Eclipse IDE, which allows access to all development tools available in the environment, such as code completion, automatic debugging during typing, use of the debugger with inspection of variables and breakpoints.

2.1.7 Benchmarking

What is Benchmarking ?

Benchmarking can have different definitions depending on the subject, if we talk about the definition of Benchmarking when developing embedded systems it can be defined as follows. The process of measuring the performance of a system/product against the considered to be the best in the industry.

How is Benchmarking done?

Benchmarking is done by studying products with superior performance, breaking down what makes such superior performance possible, and then comparing those processes to how your system operates, you can implement changes that will yield significant improvements. For example in this project the objective is to Benchmak BRIEF, which is a machine vision descriptor algorithm, having this in mind, the objective will be to compare the efficiency of this algorithm correctly compare keypoints when compared with others.

Why do Benchmarking ?

Benchmarking is an important part of the development of any product, because analysing what makes other products so successful can show us the path to the development of a product with even better specs. Benchmark tests allow measuring an application's performance, both in execution speed and memory consumption. It is relevant to make simple performance comparisons, determine load limits, test the application's ability to deal with change and find potential problem areas. Comparing two different algorithms in terms of their performance will lead to a performance enhancement to the algorithm that is being developed. This improved efficiency in the process leads to better results by the developers. This step may raise the question "Compare what to what?". Determining load limits implies identifying and exercise the various points of contention in the system. Evaluating the limits of the target platform is an excellent way of understanding it. Dealing with change means how would the algorithms perform under varying hardware configurations, for example. Benchmark tests give the user the ability to identify potential problems on a broad scale. It will not point defective code, but it helps determine which general parts of the code are the weakest and can be improved.

2.1.8 Profiling

What is Profiling ?

In software engineering, profiling is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls.

With the data retrieved when doing Profiling it is possible to find out how each function is affecting the behaviour of the system therefore it is possible to determine critical pieces of code and optimal code placement in a design. Routines that are frequently called are best suited for placement in fast memories, such as cache memory. You can also use profiling information to determine whether a piece of code can be placed in hardware, thereby improving overall performance.

How is Profiling done?

Profiling can be done using programs called profilers that give us statistic data about the program or by manipulating the code to then output important data. By Profiling it is possible to get important information from each function, such as the average time and the CPU usage of each function.

Why do Profiling ?

Through profiling it is possible to find opportunities for optimisations, because it is possible find the origin of problems that before weren't possible to detect, such as Bottlenecks.

Profilers and diagnostic techniques enable the user to procure information about memory consumption, locking, response times, and process counts from the engines that execute the application code. A profiler is a full-blown application and is responsible for conducting what are called traces on application code passed through the profiler. These traces contain information about the breakdown of function calls within the application code block analysed in the trace. Most profilers commonly contain the functionality of debuggers in addition to their profiling ability, which enables to detect errors in the application code as they occur and sometimes even lets to step through the code itself. Additionally, profiler traces come in two different formats: human-readable and machine-readable. Human-readable traces are nice because they can be easily read the output of the profiler. Nevertheless, machine-readable trace output is much more extensible, as it can be read into analysis and graphing programs, which can use the information contained in the trace file since it is in a standardised format. Many profilers today include the ability to produce both types of trace output. Diagnostic techniques, on the other hand, are not programs *per se*, but methods that can be deployed, either manually or in an automated fashion, to grab information about the application code while it is being executed. This information can be used, sometimes called a dump or a trace, in diagnosing problems on the server as they occur

2.1.9 Profiler

Programs capable of retrieving data from the target program or the whole system running the program. The data retrieved may vary depending on the profiler used and the type of analysis selected, normally more data means more overhead, so the profiler and type of analysis should be carefully chosen depending on the specs that the user wants to study.

How profilers work

There two main types of profilers, Sampling profilers and Instrumentation profilers, which work very differently. They both have Pros and Cons. The user has to analyse the specs to conclude which has the best trade-off for the target application.

Sampling profilers

Sampling profilers take periodical samples of the state of the program/system. These samples contain data such as what function is being executed at the time which the sample was taken. After profiling the program/System the profiler will use the sample to calculate which functions are using the CPU the most. This type of profilers are usually less intrusive, meaning that the code injection is lower than other types of profilers. This is a positive aspect for the user because less or no code injection generally means less overhead. This type of analysis is great to find bottlenecks

in the system, because it gives a general view of the CPU usage for each function with low overhead. This means that it is possible to identify which function is slowing down the program with data that is not greatly affected by the profiler. The bottleneck may be identified by looking at abnormal CPU consumption.

Important Specs of Sampling Profiler

To choose the best tool the group started by pointing out which was the objective when using a statistical profiler to then gather the most important characteristics. The main objective of an embedded systems engineer when using a Statistical Profiler is to find Bottlenecks (functions that are dragging the system), to then optimise the system. To do this type of analysis it is really important that the profiler contains low overhead so that the data is viable, also, the data needed to detect Bottlenecks isn't too complex, a statistical analysis of the amount of CPU time used by each function is more than enough. The final point when choosing a profiler to analyse embedded systems is that the profiler must be supported in as many platforms as possible, because this gives the user an option to analyse the program in many different platforms.

Instrumentation profiling

Instrumentation profilers instead of periodically sampling the state of the program they insert special code at the beginning of each Function/Task, this way it is possible to keep track of the whole behaviour of the system. The Instrumentation profilers are useful to get a lot of data from the behaviour of the system that may be useful to have a better idea of the improvements that the program may have, this comes with a big trade off because the routines introduced at the start of each function take some time themselves, this means that the profiler introduces a big level of overhead. The type of overhead introduced is usually subtracted at the end of the profiling process so that the calculations are accurate, so for the profiling this type of overhead is negligible yet there are other points to have in mind when analysing the pitfalls of the instrumentation profilers. Modern processors performance due to the use of Cache memory and modern pipelines introduce a high level of unpredictability on the system, this turns the process of counting the overhead out hard, specially when the routine is short, this may lead to "fake" bottlenecks.

Important Specs of Instrumentation profiling

To choose the best tool the group started by pointing out which was the objective when using a statistical profiler to then gather the most important characteristics. The main objective for the group when using the instrumentation profiler was to have access to the most data possible of the open-cv program. To do this there is a program which is clearly the best, V-Tune. This program created by Intel included in the OneApi package is the best instrumentation profiler for any open-CV application because this library already comes prepared to be profiled by the V-Tune profiler. Another important point when choosing the instrumentation profiler is if it can gather all the data needed. V-Tune Comes with many types of analysis which end up covering all the important aspects of profiling any program.

2.1.10 Hardware Description Languages (HDL)

A Hardware Description Language (HDL) can be viewed as a programming language used to describe the behaviour or structure of digital electronic circuits (ICs), and are extremely important tools for modern digital designers. These languages allow for circuit stimulation, response checking and even synthesis for deployment onto the hardware, mitigating the developing time needed to specify digital systems. Another advantage of HDLs is the faster debug cycle if the hardware implications of the code written are known.

Abstraction Levels. There are four main abstraction levels in hardware design [16]:

1. System Level;
2. Register-Transfer Level (RTL);
3. Gate Level;
4. Transistor Level;

Each level specifies the amount of detail put into the design, starting from a higher abstraction level (System Level) and iterating until reaching the lowest level (Transistor Level).

Hierarchical Organisation. The hierarchical organisation of a HDL design (fig. 2.1.35) comprises a set of black-box modules that act on their inputs and provide the respective outputs. Additionally, one module can contain other sub-modules, and the combination of all of them results on the hardware device [16].

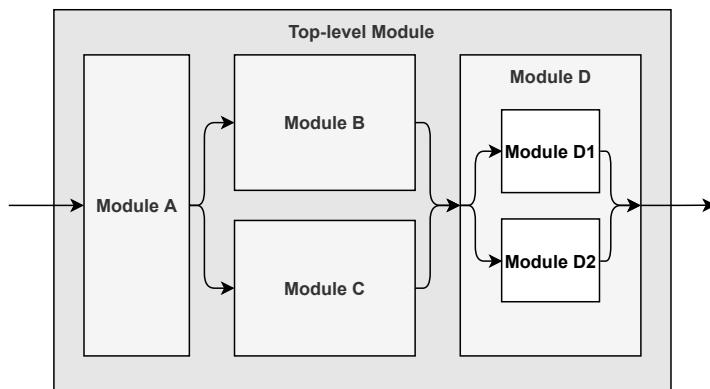


Figure 2.1.35: Hierarchical Organisation of an HDL Design

Testbenches. Before synthesis tools generate the design bitstream, that corresponds to its physical implementation, based on the code written in a HDL, it must be simulated using HDL simulation tools. To perform these simulations testbenches (fig. 2.1.36) are used to provide the Device Under Test (DUT)/Unit Under Test (UUT) with its inputs, so it can produce outputs. In these files (testbenches) non-synthesisable constructs are used to generate the forementioned input stimuli. The behaviour is then monitored in simulation [16].

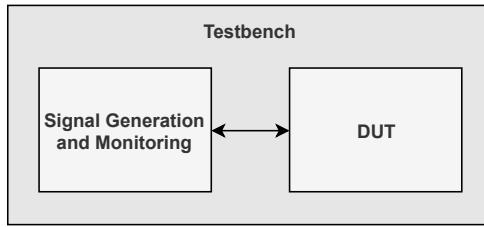


Figure 2.1.36: HDL Testbench

HDL Gap Analysis. There are many HDLs available but Verilog/SystemVerilog and VHDL are by far the most popular. For the project the Verilog HDL will be used, since its has a C-like syntax, providing a good learning curve at entry level. Additionally, it performs well when designing for lower level abstractions. Of course this factor can also be a disadvantage because Verilog is not as semantically strict as VHDL, but that is a trade-off one is willing to make to speed up project development. The two HDL gap analysis can be seen on table 2.1 [19].

Based on	Language	Learning Curve	Verbose	Common Context
Verilog	C	Weakly-typed	Good	Less
VHDL	Ada	Strongly-typed	Not so good	More

Table 2.1: HDL Candidates Gap Analysis

2.1.11 Field Programmable Gate Array (FPGA)

As the name implies, a Field-Programmable Gate Array (FPGA) is a semiconductor Integrated Circuit (IC) in which the hardware fabric can be reprogrammed to serve different applications or functionality requirements after manufacturing, like implementing certain hardwired and silicon-based chips. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASIC)s, which are custom manufactured for specific design tasks. Although one-time programmable (OTP) FPGAs are available, the dominant types are SRAM-based which can be reprogrammed as the design evolves [20]. As stated before, these devices are based around reprogrammable elements, with each vendor adopting a certain designation for it. For example, Xilinx specifies a matrix of Configurable Logic Blocks (CLBs) and Altera specifies one of Logic Array Blockss (LABs), both connected via programmable interconnects (fig. 2.1.37 [18]).

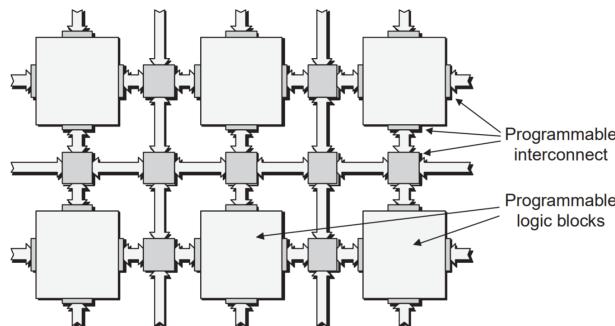


Figure 2.1.37: Underlying FPGA Fabric

Elements. The FPGA chip contains two types of elements:

- Flip-flops;
- Look-Up Tables (LUTs).

With the combination of these two types of elements, one can implement a plethora of combinational or sequential circuits. Additionally, it is important to note that architectures that use a significant amount of LUT-based logic blocks might experience a drop in speed and an increase in power consumption. Not to mention these designs are usually more expensive than the ones using hardwired elements such as memories or DSPs. The configuration file (bitstream) of a modern FPGA chip is usually stored in non-volatile memories that come accompanied with circuitry to program the FPGA on power-on.

2.1.12 Advanced eXtensible Interface (AXI)

The Advanced Extensible Interface (AXI) interface protocol was defined by ARM as part of the Advanced Microcontroller Bus Architecture (AMBA) standard. This family of microcontroller buses was first introduced in 1996. The AXI protocol increases productivity by standardising the interface, so developers only need to learn one protocol by IP, but also improves flexibility and availability, since it provides the right protocol for the applications and expands the access to the Xilinx IP catalogue and the community of ARM partners, respectively [22].

Interfaces. There are three 3 types of AXI4-Interfaces (AMBA 4.0) [21]:

- **AXI4 (Full AXI4)**, for high-performance memory-mapped requirements;
- **AXI4-Lite**, for simple and low-throughput memory-mapped communication (for example, to and from control and status registers);
- **AXI4-Stream**, for high-speed streaming data.

Channels. The AXI protocol defines five channels:

- Two are used for Read Transactions: *read address* and *read data*;
- Three are used for Write Transactions: *write address*, *write data* and *write response*.

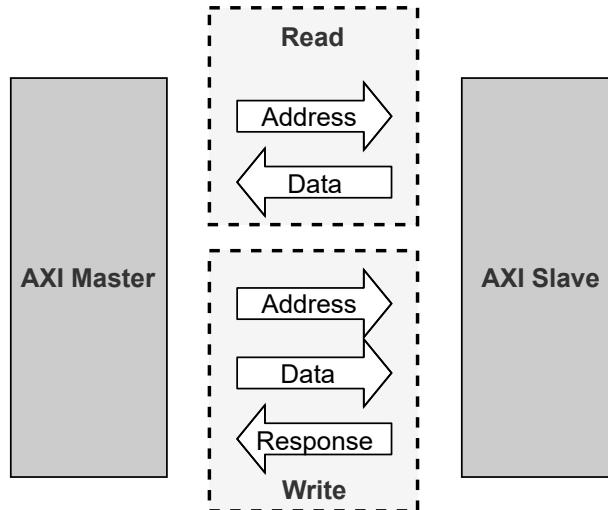


Figure 2.1.38: AXI Read and Write Channels

Every single channel is group of AXI signals associated with the **VALID** and **READY** signals. Additionally, any piece of data transmitted on a single channel is called a transfer. A transfer happens when both the VALID and READY signals are high while there is a rising edge of the clock, as one can see in fig. 2.1.39 in the instant T3 [21].

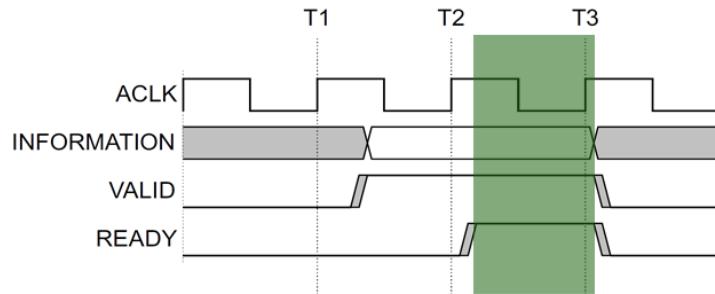


Figure 2.1.39: Example of an AXI Transfer

AXI Memory Mapped

The memory mapped protocols in AXI are **AXI3**, **AXI4** and **AXI-Lite**. Note that an AXI3/AXI4/AXI4-Lite interface can be read only or write only. All of these protocols revolve around the concept a target address within a system memory space and data transfers.

Read Transactions. An AXI read transaction requires multiple transfers on the 2 read channels (*read address* and *read data*). In these read-only interfaces, the *address read channel* is sent from the Master to the slave alongside some control signals to set the target address. Then, the data from this target address is transmitted from the Slave to the Master via the *read data channel*. If in a transaction there are multiple data transfers per target address, this type of transaction is called a *burst* transaction. An example of an AXI read transaction can be observed in fig. 2.1.40.

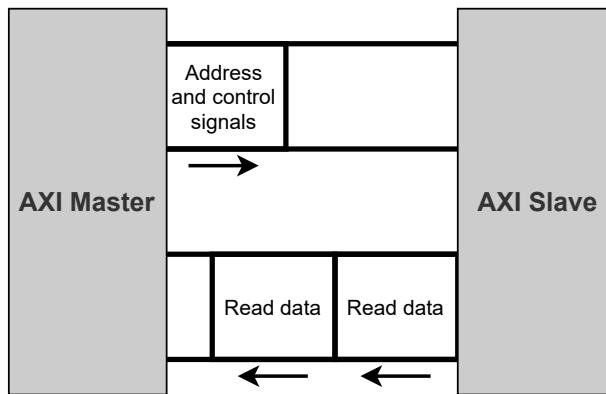


Figure 2.1.40: Example of an AXI Read Transaction (Memory Mapped)

Write Transactions. Similarly, an AXI write transaction requires multiple transfers on the 3 read channels (*write address*, *write data* and *write response*). In these write-only interfaces, the transaction starts with the Master sending to the Slave the *address write channel* alongside some control signals, once again, to set the target address. Secondly, the data from this address is sent from the Master to the Slave via the *write data channel*. Finally, the *write response* is sent from the Slave to the Master on the *write response channel* to indicate the positive or negative outcome of the transfer. An example of an AXI write transaction can be seen in fig. 2.1.42.

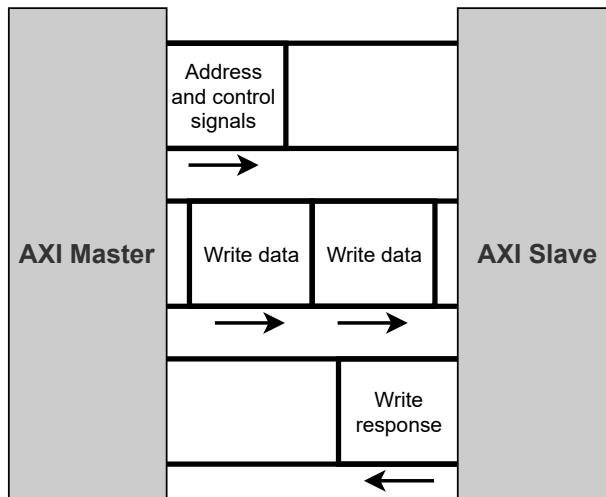


Figure 2.1.41: Example of an AXI Write Transaction (Memory Mapped)

Response Values. The possible response values on the *write response channel* are [21]:

- **OKAY (0b00):** Normal access success. Indicates that a normal access has been successful;
- **EXOKAY (0b01):** Exclusive access okay;
- **SLVERR (0b10):** Slave error. The slave was reached successfully but the slave wishes to return an error condition to the originating master (for example, data read not valid);

- **DECERR (0b11)**: Decode error. Generated, typically by an interconnect component, to indicate that there is no slave at the transaction address.

It is also important to note that, read transactions also have a response value but it is transmitted as part of the *read response channel*. Moreover, a *write response* must always follow the last write transfer in the write transaction of which it is a part. Also, read data must always follow the address to which the data relates, and a slave must wait for both ARVALID and ARREADY to be asserted before it asserts RVALID to indicate that valid data is available.

Memory mapped systems often provide a more homogeneous view of the system, because the IPs operate around a defined memory space [22].

AXI4-Stream

The AXI4-Stream protocol (lower level) is commonly used in applications with no present concept of an address or where the latter is not needed. This usually happens in systems that follow a data-flow paradigm. In the protocol, each stream acts as a single simplex channel (unidirectional) to establish a data flow based on handshakes, which means that the inter-IP data movement mechanism is solid and efficient but it isn't associated with any address context that connects the IPs via system memory space [22].

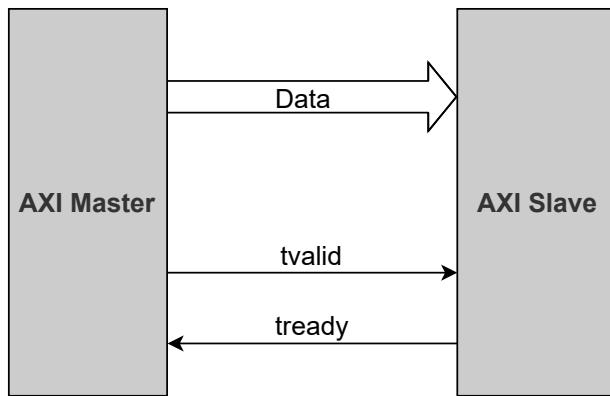


Figure 2.1.42: Example of an AXI4-Stream Transaction

The **TVALID** and **TREADY** handshake indicates when information is passed across the interface. In this protocol, a two-way flow control mechanism allows for both parts (Master and Slave) to specify the rate of transmission of the data and control information.

For a transfer to take place both TVALID AND TREADY signals must be driven high. It doesn't matter which one is put to high state first but it is imperative that both are asserted to initiate the transfer. The Master interface controls TVALID and the Slave interface controls TREADY. Regarding this, both can even be driven to high state in the same ACLK cycle. On the one hand, a Master mustn't wait for the Slave interface to assert TREADY before asserting TVALID, but, on the other hand, the Slave is permitted to wait for the master to drive TVALID to an high state before asserting TREADY and even deassert TREADY if the Master hasn't yet asserted TVALID. Note that once TVALID is asserted by the Master interface it should remain asserted until the handshake occurs [23].

In fig. 2.1.43 the Master presents the data and control information and drives the TVALID signal HIGH. Once the Master has asserted TVALID, its data or control information must remain unchanged until the slave drives the TREADY

signal HIGH, indicating that it can accept transaction. In this case, the transfer takes place once the slave asserts TREADY HIGH. The black arrow indicates the moment when the transfer occurs.

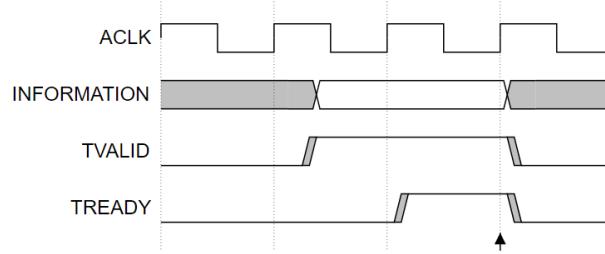


Figure 2.1.43: Example of an AXI4-Stream Handshake (1)

In fig. 2.1.44 the slave drives TREADY HIGH before the data and control information is valid (TVALID is asserted HIGH by the Master). This indicates that the destination can accept the data and control information in a single cycle of ACLK. In this case, the transfer takes place once the Master asserts TVALID HIGH.

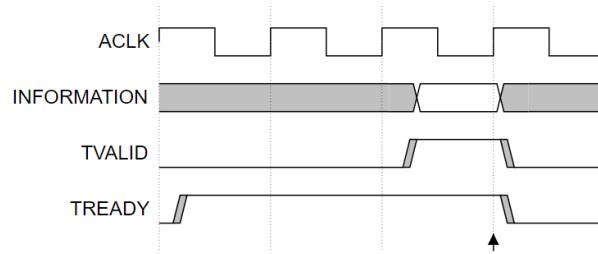


Figure 2.1.44: Example of an AXI4-Stream Handshake (2)

In fig. 2.1.45 the Master asserts TVALID HIGH and the Slave asserts TREADY HIGH in the same cycle [23]. In this case, the transfer takes place in the same cycle [23].

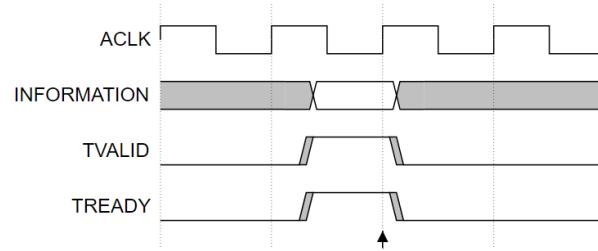


Figure 2.1.45: Example of an AXI4-Stream Handshake (3)

2.2 Requirements and Constraints

The development requirements are divided into functional and non-functional if they pertain to main functionality or secondary one, respectively. Additionally, the constraints of the project are classified as technical or non-technical.

2.2.1 Functional Requirements

- **Control** through the ZYNQ7 Processing System.
- **Identify and match** features from two different images.

2.2.2 Non-Functional Requirements

- Use as little logic elements as possible;
- Be as configurable as its software counterpart;
- Offer a significantly better latency and throughput over its software counterpart.

2.2.3 Technical Constraints

- Use ZYBO Z7 ZYNQ-7010 as a prototyping board;
- Use Xilinx Vivado as the development platform;
- Use Xtext and Xtend for a DSL generative model.

2.2.4 Non-Technical Constraints

- Project deadline at the end of the semester;
- Class workflow.

2.3 System Overview

In fig. 2.9.1, one can see the where the each stage fits within the big picture. This is to further contextualise the problem and aid in objective setting and project management.

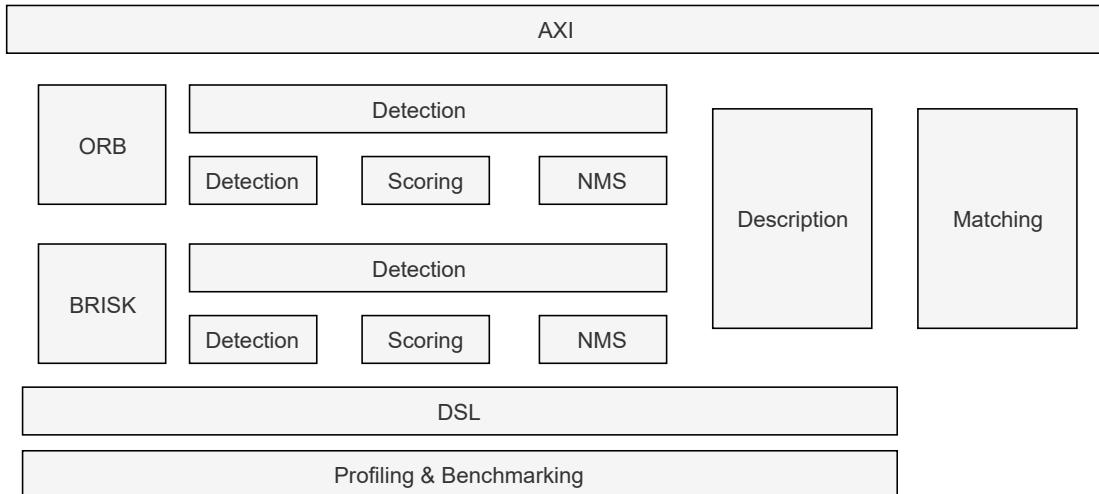


Figure 2.3.1: System Big Picture

2.4 Hardware Specification

The specifications for the ZYBO Z7 ZYNQ-7010 are presented in this section.

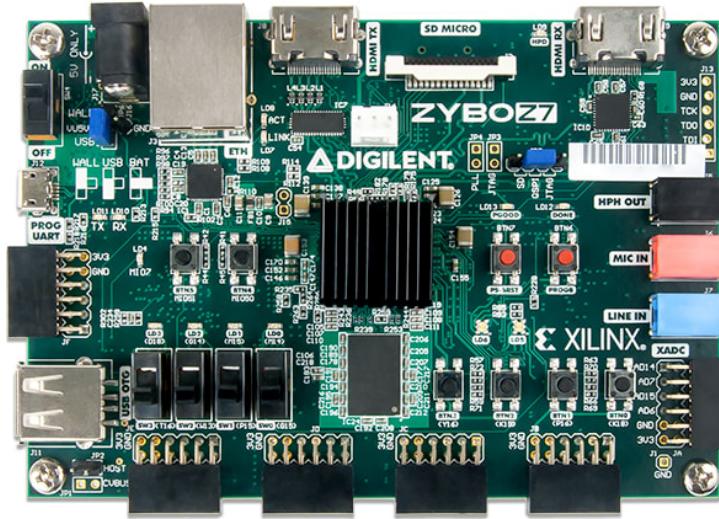


Figure 2.4.1: Zybo Z7-10 Development Board

Processing System (PS) Specifications:

- ARM Cortex-A9 processor
- 8-Channel DMA Controller
- High-bandwidth peripheral controllers:
 - SD/SDIO 2.0: 2
 - USB 2.0: 2
 - 10/100/1000 tri-speed Ethernet: 2
- Low-bandwidth peripheral controllers:
 - CAN 2.0B: 2
 - Full-duplex SPI: 2
 - High-speed UARTs (up to 1 Mb/s): 2
 - I2C: 2
- Caches:
 - L1: 32kB
 - L2: 512kB
 - Byte-parity support

On-Chip Memory:

- On-chip boot ROM
- On-Chip RAM (OCM): 256kB
- Byte-parity support

Programmable Logic (PL) Specifications:

- Clock Resources:
 - PLL: 2
 - MMCM: 2
- Part Number: Zynq XC7Z010-1CLG400C
- Logic Slices: 4400
- LUTs: 17600
- Flip-Flops: 35200
- BRAM: 270kB
- DSP Slices: 80

2.5 System Architecture

The architecture for the overall system will be composed by a Processing System (PS) and Programmable Logic (PL) within a ZYNQ Board (ZYBO) Field-Programmable Gate Array (FPGA). The system will have two AXI buses for data exchange, the AXI4-Stream one, focused on higher throughput and the AXI4-Lite one, focused on smaller latency. The algorithm in question will detect keypoints on the desired image, the latter will first pass through a pre-processing stage in software to convert it to grey scale, and then a smoothing process in hardware with the application of a filter. The hardware accelerator will be on the PL, and will access certain image regions stored on the On-Chip Memory. To store the image slices on this type of memory one will need to access the PS' DDR memory to read the image. An overview of the system's architecture can be seen on fig. 2.5.1.

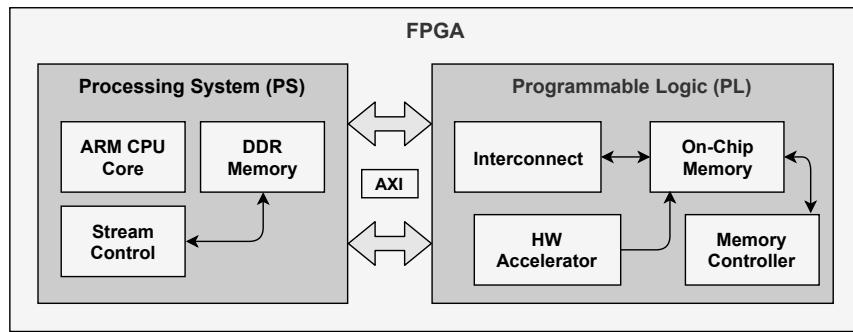


Figure 2.5.1: System Architecture: Overview

2.6 Interface

The interface between the Processing System and the Accelerator is comprised of two main channels:

- A set of AXI4-Stream-powered interfaces allowing full-duplex communication, for streaming the images from the Memory to the Accelerator and the results from the Accelerator back to the Memory;
- An AXI4-Lite Interface as a communication medium between the PS and the PL for register-based control logic.

In order to meet these requirements, a solution based on the Universal Controller for Streaming Processors [1] will be implemented, as schematised in fig. 2.6.1.

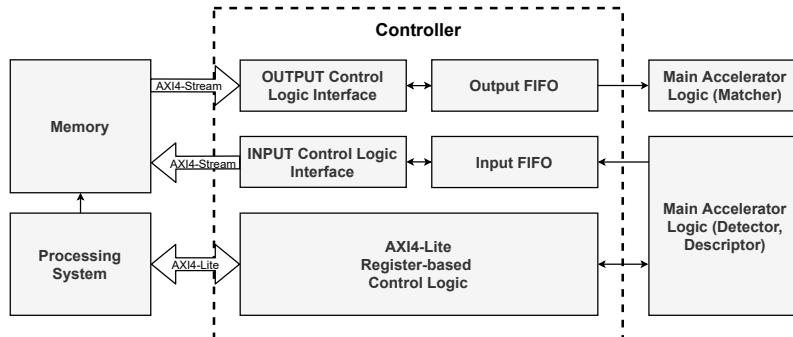


Figure 2.6.1: Hardware Architecture: Processing System Interface

2.7 Detection

The block diagram in figure 2.7.1 shows the connections between the detection algorithms and the enveloping modules required to create a working interface with other stages.

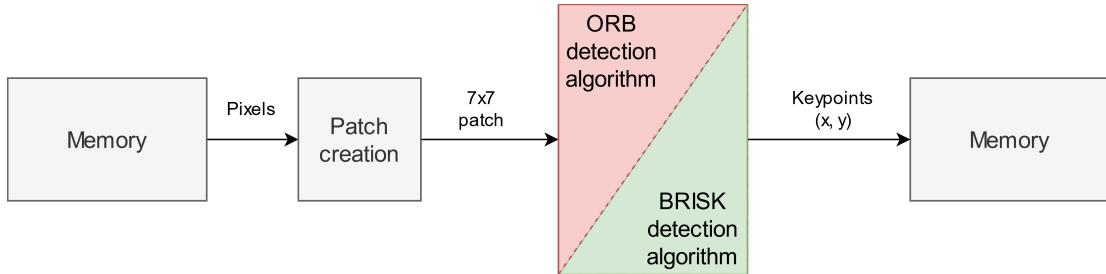


Figure 2.7.1: ORB and BRISK block diagram

In order to ensure modularity, the input and output of both ORB and BRISK algorithms are setup so that these can be interchanged without affecting the connecting modules.

The input to the detection stage is a 7×7 patch, which matches the maximum size allowed by the FAST-n algorithm. Its output is an array containing the various corners found by the detector. However, some glue logic is required to generate the patches. This is handled by the Patch creation block, who's inputs are the pixels that make up the main image over which the computer vision process is being done.

2.8 ORB Detection

Firstly, we will analyse the ORB algorithm's states, use case and sequence diagrams, keeping in mind both hardware and software implementations.

2.8.1 Overview

The patcher module is responsible for selecting the pixels that make up the patch that matches the FAST-n requirement. This patch is used to select the pixels contained not only in the appropriate Bresenham circle but also to calculate the corresponding m_{01} and m_{10} . This is done resorting to BRAM FIFOs and shift file registers, as depicted in 2.8.1. An auxiliary module is required to stream a single pixel at each time step. The outputs are then fed into their respective modules. The *Dark/Bright classifier* block contains the classification algorithm employed by FAST-n, as seen in section 2.1.3.

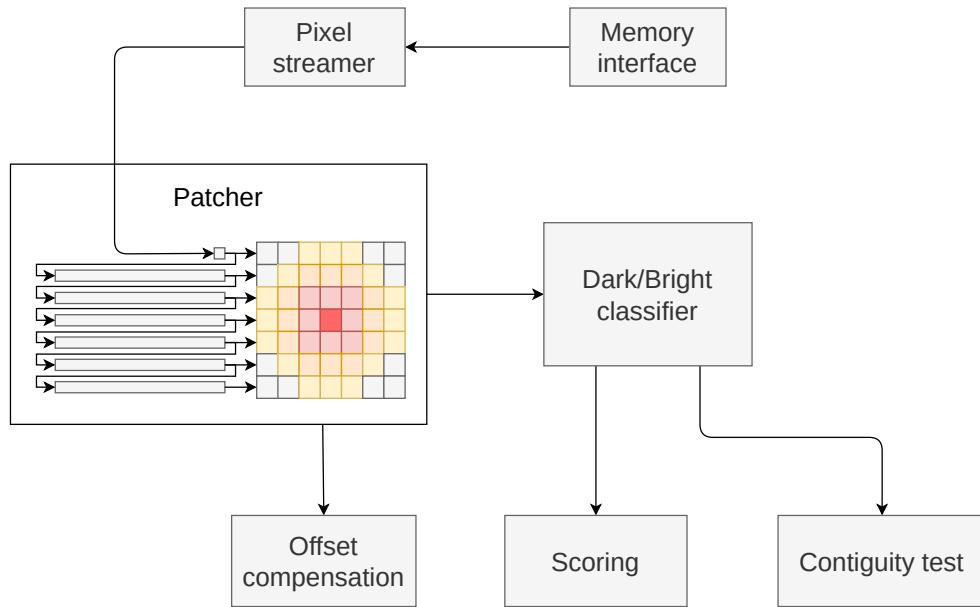


Figure 2.8.1: Overview of ORB detector architecture

2.8.2 Rotation Invariance

In the fig. 2.8.2 it is possible to observe how the ORB's rotation invariance module fits in the non-maximum suppression module and how the Sparse Point module interacts with it. The non-maximum suppression module has as inputs the location of a certain keypoint and its current patch momentum, and, as the overlapped keypoints are discriminated by the Sparse Point algorithm, the effective keypoints locations are passed to the rotation invariance module that makes use of the patch momentum to compute these keypoints' orientation (given by the sin and cos of the rotation angle) and output their oriented coordinates.

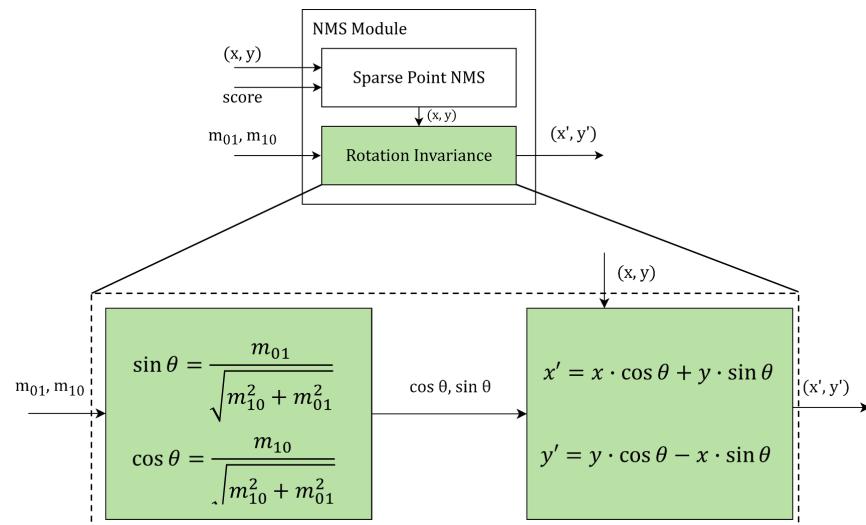


Figure 2.8.2: Module Overview

To better understand the module being developed, the flowchart in the Figure 4 is presented as an internal point of view, where it does not matter where the inputs come from neither where the output goes. As inputs, this module receives the coordinates of a certain pixel, that previously was detected as a corner by the ORB detector and then consummated as a keypoint by the Sparse Point non-maximum suppression algorithm, and the moments of the patch (m_{10} and m_{01}) regarding that keypoint. Then, the cos and sin of the rotation angle are computed, resorting on the eq. (2.6), to characterise the orientation. The new coordinates, the rotated/oriented coordinates, are finally obtained applying the eq. (2.7) and stored in a memory shared with the subsequent module, the description module.

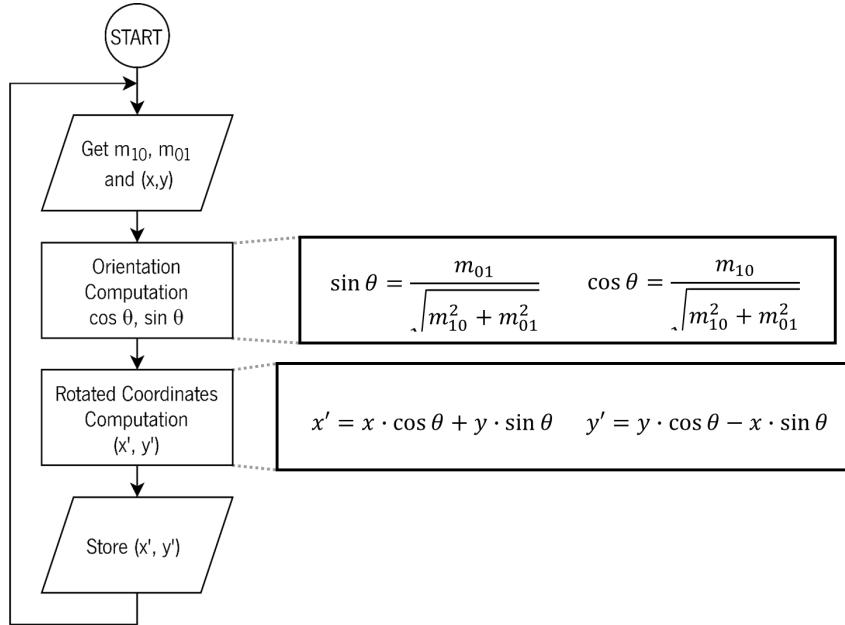


Figure 2.8.3: Module Overview Flowchart

Implementation Approaches

Even though the problem of this project is relatively simple to understand and has a straightforward solution if implemented in software, it is very complex to solve when inserted in the current context, i.e., in a hardware implementation with a FPGA board as target. Under the inability of accomplishing an optimal answer, several implementation approaches are studied in order to keep the resources footprint to a minimum, while seeking for a solution that does not compromise the module's purpose.

Hybrid Approach. One of the possible approaches is one that can bypass the previously described difficulties of a hardware implementation and process the calculations demanding floating-point precision. To accomplish this, the moments of the patch and the keypoint location need to be transferred from the Programmable Logic to the Processing System through an AXI communication interface and after the stated calculations are done, the rotated keypoint locations are transferred back from the Processing System to the Programmable Logic recurring to the same type of communication strategy. Due to this combination of software and hardware, this approach is named as hybrid.

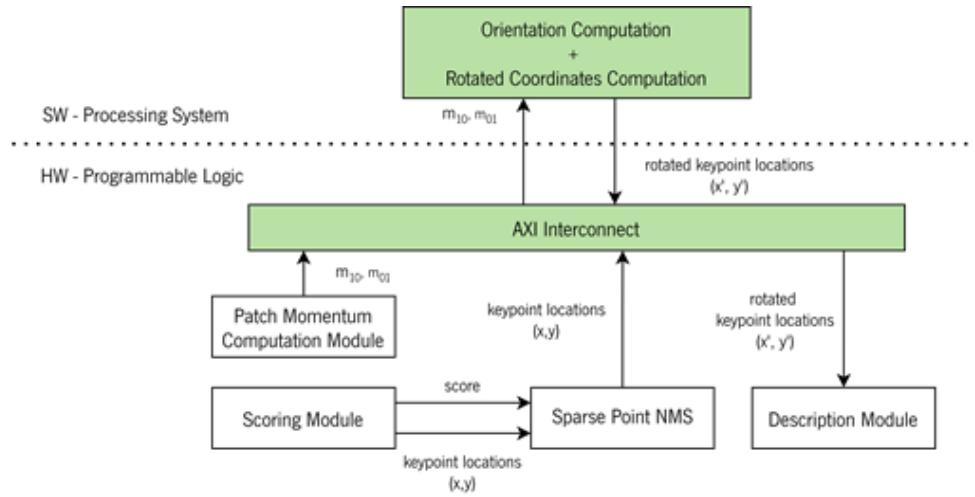


Figure 2.8.4: Hybrid Approach Diagram

Hardware Approaches. An implementation in hardware is flexible enough to be manipulated in order to suit the requirements in terms of resources, performance, efficiency and quality of results. Hardware approaches, although they consume PL resources, they also allow a combinational execution and do not possess overhead related to the shifting of domains.

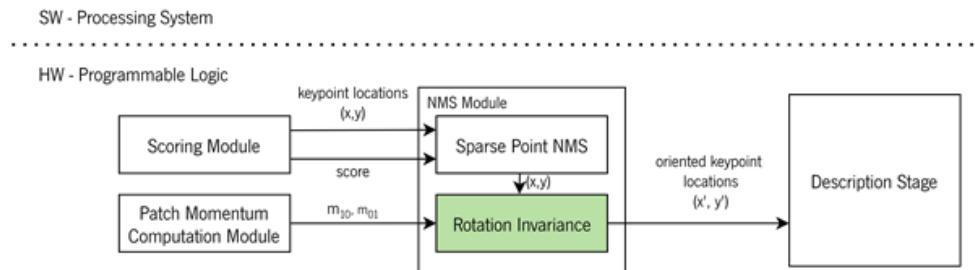


Figure 2.8.5: Hardware approach diagram

In terms of quality of results, an approach capable of computing the precision eager operations, namely the square root and the divisions, using floating-point numbers is the ideal. However, when seen from a resource perspective, this approach might not be viable because the manipulation of floating-point numbers and operations are not cheap to do, as it was hinted in the [2.1.2 Floating-Points](#) section. Besides, the resulting performance impact of the said manipulation and operations of the floating-point numbers can be, by itself, a factor that compromises its implementation in the project it is inserted, as this module needs to be quick to avoid stalls in the project's execution flow.

The second approach consists of using fixed-point numbers and rounding the result of the operations that oblige greater precision. In an attempt to minimise the loss of precision due to the lack of the utilisation of floating-point numbers, a scalar N is introduced in the numerators of the fractions used to compute the sin and cos of the rotation angle. This scalar N is suppressed later, on the multiplications used for the computation of the oriented coordinates. Resorting to this approach, the orientation computation is then done following the Equations [2.13](#), while the rotated coordinates computation is performed following the Equations [2.14](#).

$$\sin \theta = \frac{m_{01} \times N}{\sqrt{m_{10}^2 + m_{01}^2}}, \cos \theta = \frac{m_{10} \times N}{\sqrt{m_{10}^2 + m_{01}^2}} \quad (2.13)$$

$$x' = \frac{x}{N} \times \cos \theta + \frac{y}{N} \times \sin \theta, y' = \frac{y}{N} \times \cos \theta - \frac{x}{N} \times \sin \theta \quad (2.14)$$

2.8.3 Hardware acceleration

State chart

In spite having multiple sub-processes, the nature of an accelerated hardware implementation implies a certain degree of pipeline where each sub-process feeds the following one, without the need for a state change. For this reason, the only two states considered are *Idle* and *Detect*, as can be seen in figure 2.8.6.

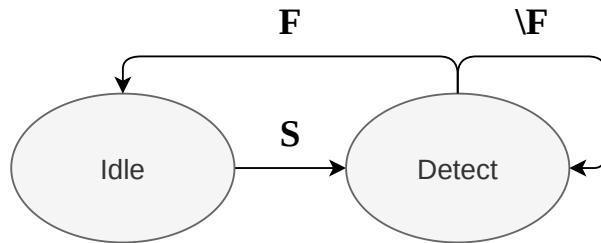


Figure 2.8.6: ORB module state chart

Where:

- **S** - Start signal
- **F** - Finished processing image

Use cases

The use case diagram in figure 2.8.7 highlights the role of the DSL, controlling the image size and the *n* order of the FAST algorithm. The memory block serves as the data interface for both inputs and outputs. As it can be seen, the only data input is the raw image pixels while the outputs are the keypoint coordinates, allowing the following stage to retrieve the information.

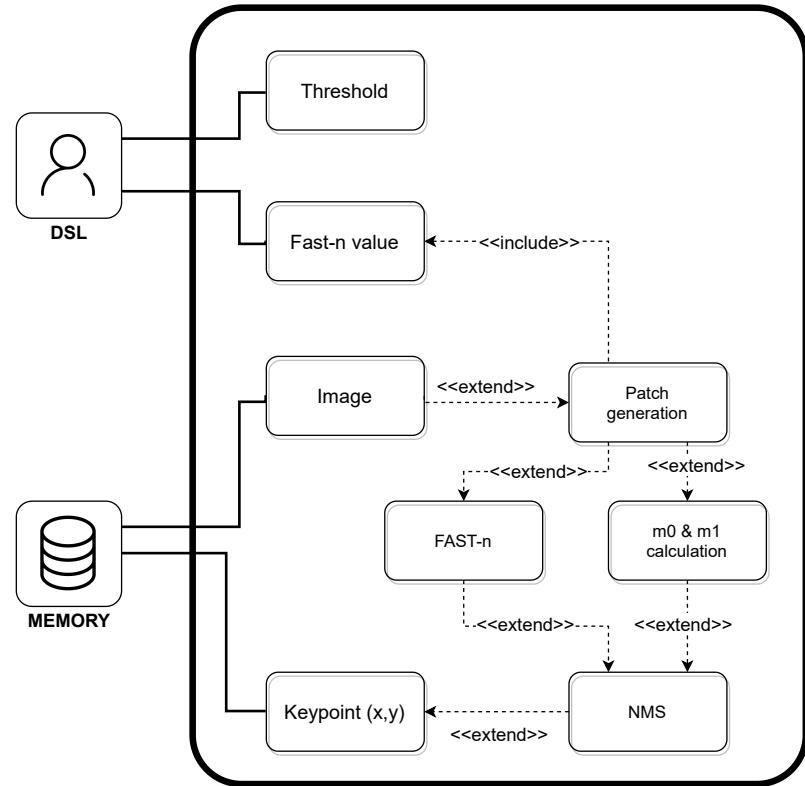


Figure 2.8.7: ORB Use Case diagram

Sequence Diagram

As the use case diagram in figure 2.8.7 states, the only input to the detection module is the image itself. This image is segmented into patches that vary in size depending on the *FAST-n size* argument passed by the DSL. Once segmented, the patches can be computed to determine the corner coordinates and their respective offset to account for the rotation invariance. Lastly, the detected corners are passed through the *Non-maximum Suppression* block and written back to memory for future access.

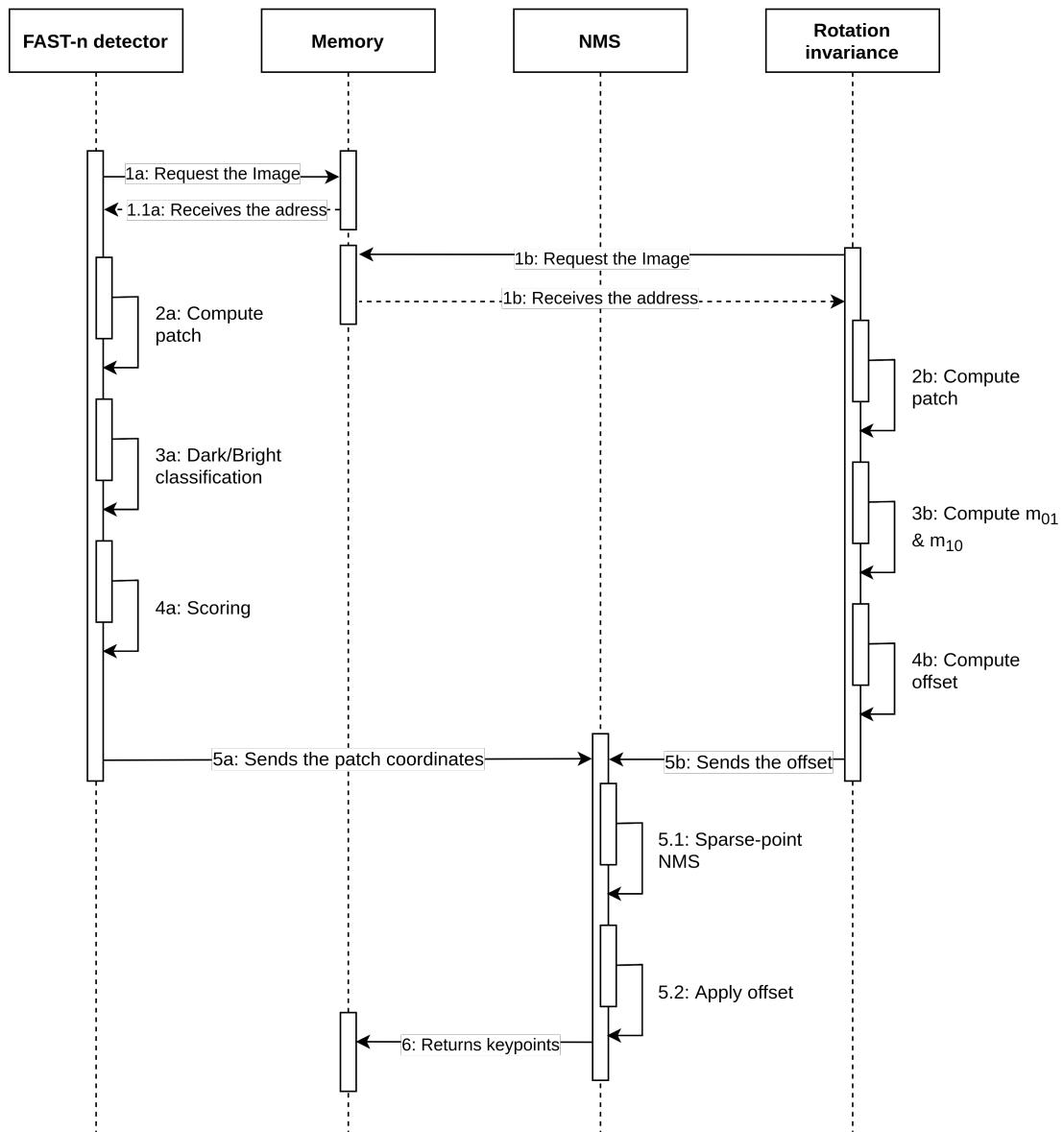


Figure 2.8.8: ORB Sequence Diagram

2.8.4 Software refactoring

After analysis of OpenCV implementation of ORB, it was decided to make two functions, the first function (create) receives the FAST-n type to do score, the second one (detect) receives the image and the threshold.

Uses Cases. In the follow image it is possible see the uses cases diagram.

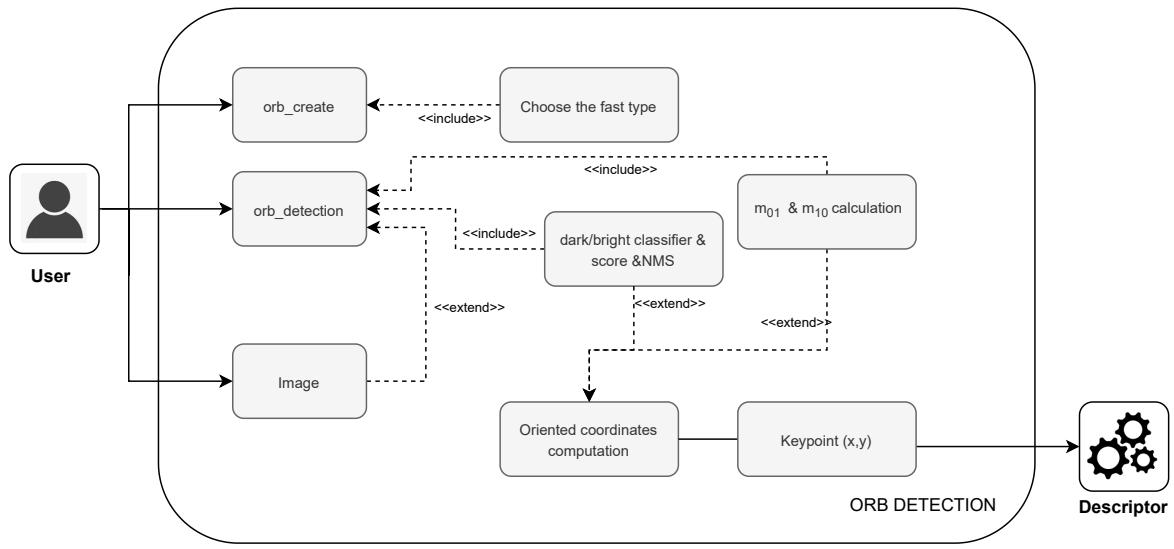


Figure 2.8.9: Use Case Diagram (Software Refactored)

In the figure below it is possible to see the use of the two functions. The first one performs step (1) and the second one performs the step (2).

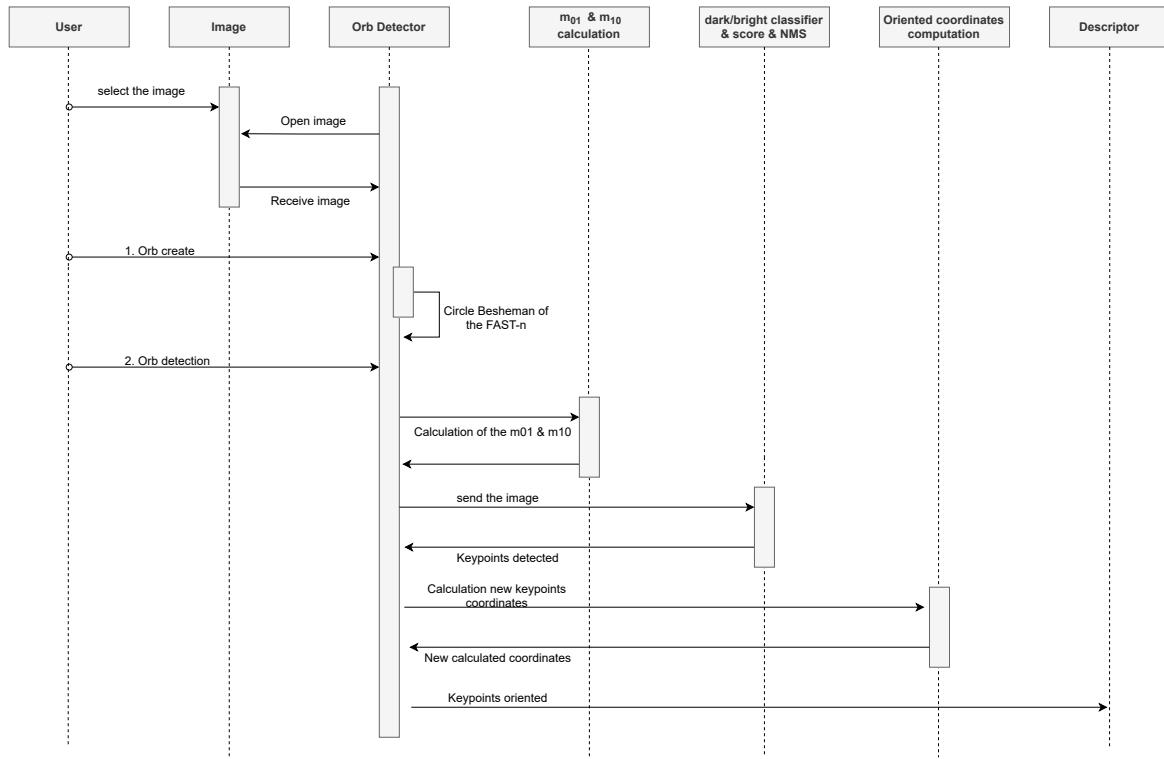


Figure 2.8.10: Use Case Diagram (Software Refactored)

2.9 BRISK Detection

Regarding hardware, the image will be stored in Block Random-Access Memory (BRAM), interfacing with a memory controller that dictates how data is read and written to it and a detection controller to set the overall flux of information and control signals. Additionally, the latter and the register file will also interface with the bright/dark classifier, which will then propagate the results of the detection stage to the scoring stage.

Overview. The figure 2.9.1 represents the FAST detector module hardware architecture. This proposal stipulates that the image data must be fed via Advanced Extensible Interface (AXI) as depicted. The rectangular 7x7 image sub-region containing the Bresenham circle aforementioned in fig. 2.1.15a will be available within the register file. The register file is composed of programmable logic flip-flops, facilitating independent and simultaneous accesses to all the pixel values. The n pixels, including the centre one will then serve as input of a Bright/Dark classifier that will populate two n bit arrays (`is_dark` and `is_bright`) with one or zero if the pixel in question is darker or brighter than the centre pixel (P) of the circle, respectively. If the centre pixel passes a contiguity test performed later in the scoring phase the pixel is considered to be a keypoint candidate or, in other words, a corner.

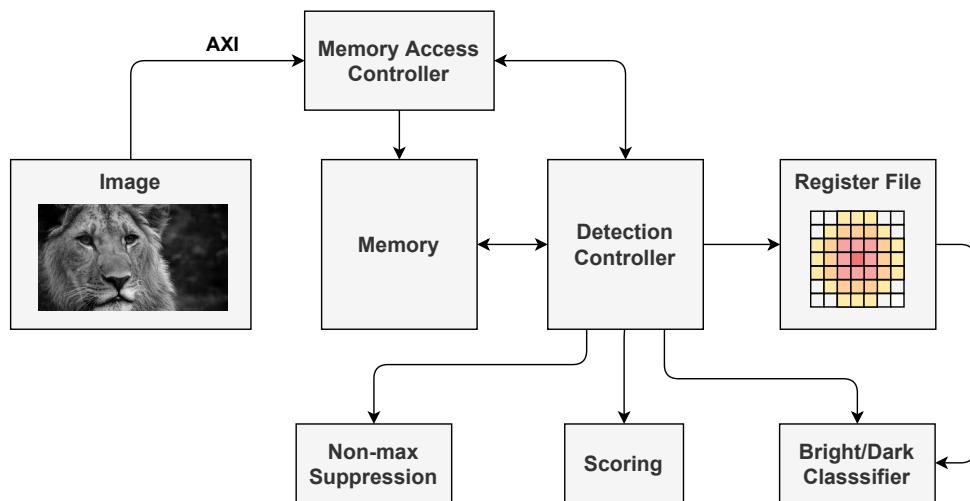


Figure 2.9.1: Hardware Architecture: Detector Overview

2.10 Description

Analysing the BRIEF Descriptor Module in more detail through its respective diagram presented in fig. 2.10.1, the overview of BRIEF, the modules required and their connections are displayed.

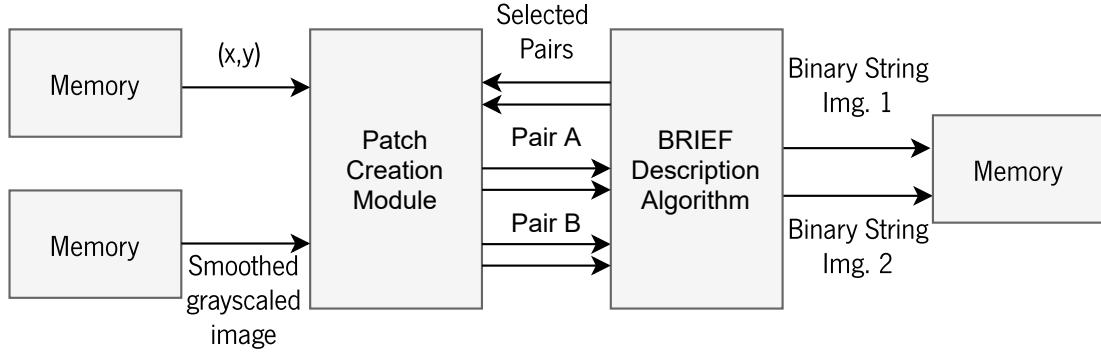


Figure 2.10.1: BRIEF block diagram

As it can be seen in fig. 2.10.1, which shows an overview of the BRIEF descriptor module, the BRIEF descriptor has two inputs, the location of the keypoints and the smoothed greyscaled image. The output produced is a binary string for each keypoint.

For each keypoint fetched, a patch of $N \times N$ pixels has to be created around it. Binary tests will then be computed on pairs requested to the patch created following a sampling pattern generated in an earlier stage. This produces a binary descriptor string for each keypoint received which is then stored in the memory to be accessed by the next stage of the algorithm, matching.

2.10.1 State Chart

When approaching BRIEF implementation on both hardware and software, they share the same line of thought, which is exemplified in fig. 2.10.2.

So the BRIEF module will always create a patch (where an $N \times N$ area around the keypoint is selected), followed by the pair generation (which selects the sampling pairs), and finally computes the descriptor for that said keypoint.

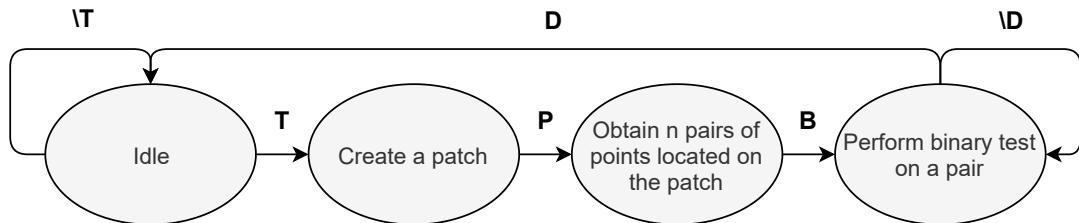


Figure 2.10.2: State Chart of BRIEF module

Where:

- **P** - Patch created;
- **B** - All pairs are chosen;
- **T** - Keypoint available on memory;
- **D** - The descriptor is complete;

2.10.2 Hardware Acceleration

At this stage, the hardware acceleration process will be analysed through different diagrams. These diagrams show the DSL usage, the interface between the memory and its necessary modules, from the patch creation leading to the descriptor generation.

Use Cases

Through the analysis of the use-case diagram, it is noticeable that DSL will change the size of both the descriptor and the patch. This two changes reflect heavily in the stages that refer to patch generation pair sampling (pattern generation), and consequently in the computation of the descriptor.

In fig. 2.10.3 the memory will act as bridge, providing the descriptor with the required inputs for its computation, and also storing the value of the descriptor allowing the matching stage to get it later.

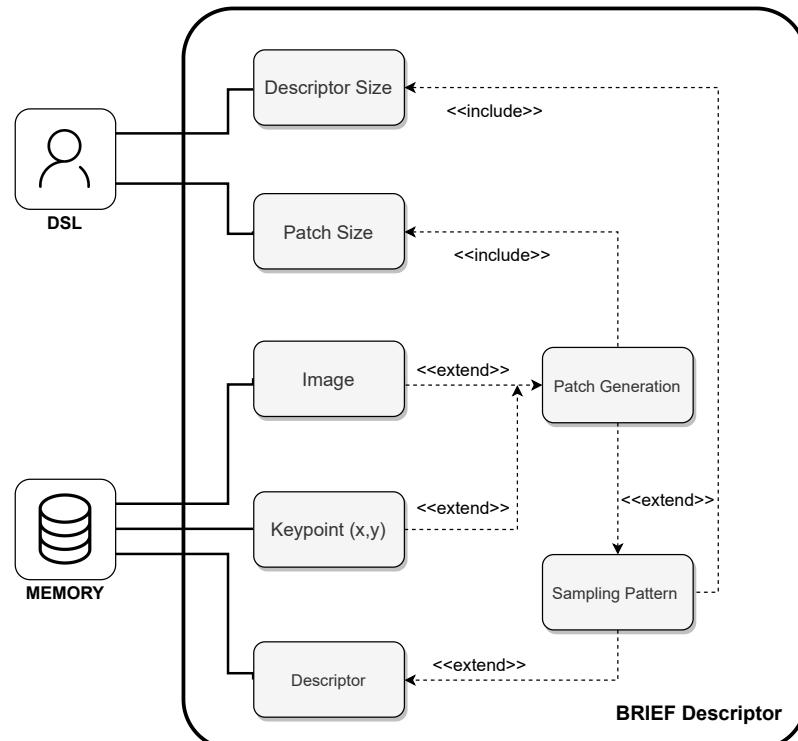


Figure 2.10.3: BRIEF Use Case Diagram (Hardware)

Sequence Diagram

As previously stated BRIEF first requires both the image and the keypoints location to work, so as we can see in fig. 2.10.4 step 1 and 2 is getting both from memory. Having the information needed for the generation of the descriptor the necessary NxN patch is computed (step 3), giving it to the pattern generation module (step 4), received the N pairs necessary for generating the descriptor its is computed and saved in the memory (step 5 and 6).

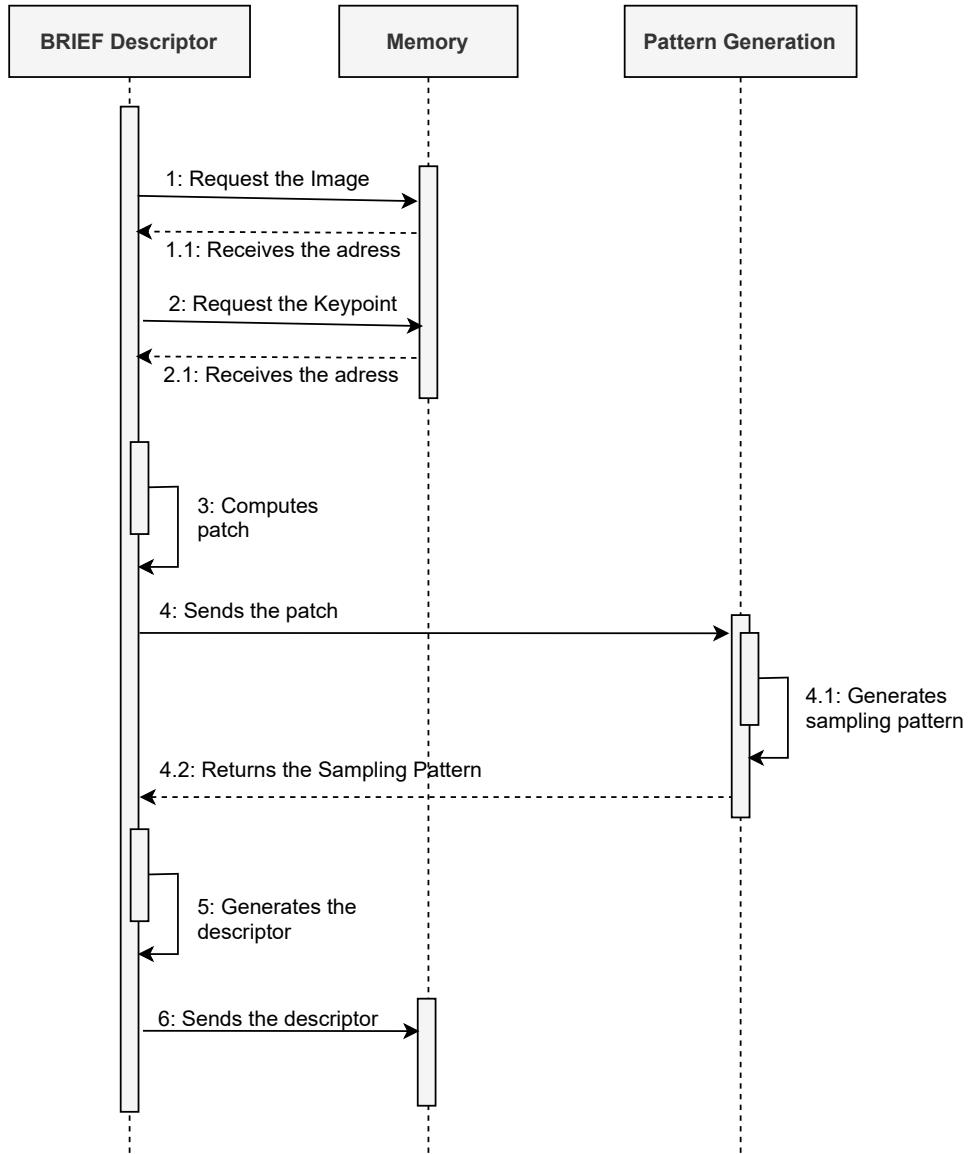


Figure 2.10.4: BRIEF Sequence Diagram (Hardware)

2.10.3 Software Refactoring

After analysis of OpenCV implementation of ORB, it was decided to make two functions, the first function receive the FAST-n type to do score, the second one the image and the threshold. In the follow image it is possible see the uses cases diagram.

In the figure below it is possible to see the use of the two functions. The first one performs step 1 and the second one performs the step (2).

Since Domain-Specific Language (DSL) will allow different methods for the execution of the computer vision system, a software refactoring is needed in parallel with the hardware acceleration.

Use Cases

After the analysis of OpenCV implementation of BRIEF, the decision to make the same two functions was made, the function create which will allocate the necessary space, and the function compute which will receive as inputs the pre-determined image and keypoints, and return the corresponding descriptor for each keypoint.

As seen in the image 2.10.5 the compute function uses the patch and pattern generation module, allowing later the computation of that said keypoint.

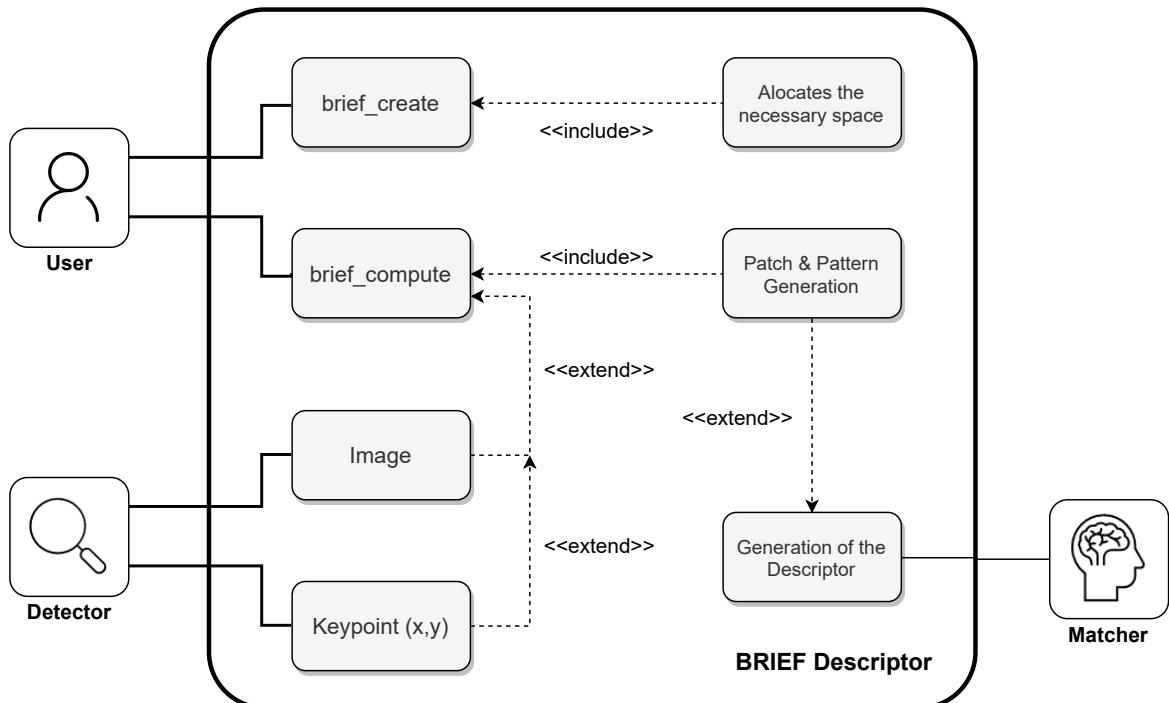


Figure 2.10.5: BRIEF Use Case Diagram (Software)

Sequence Diagram

In the figure below the use of both functions would be carried out, first the create would allocate the necessary for the desired descriptor (step 1). Compute would follow getting both the image and keypoints (steps 2.1 and 2.2), computes the necessary patch and sampling pattern (steps 2.3 and 2.4), and generates the descriptor returning it.

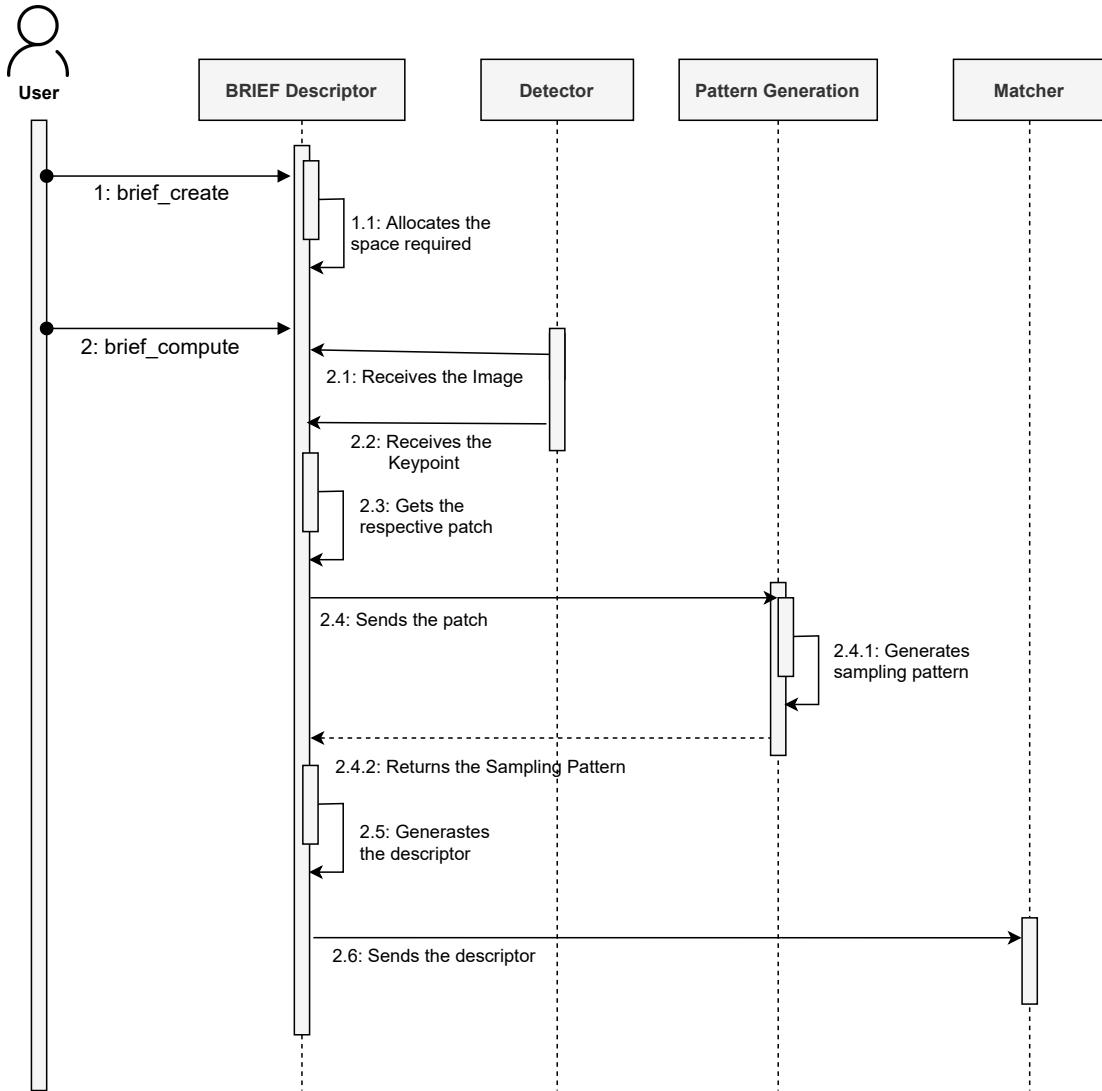


Figure 2.10.6: BRIEF Sequence Diagram (Software)

2.11 Matching

The state of the art concerning the application to solve the problem statement introduces the Brute Force algorithm and FLANN library, respectively, as two possible proposed solutions. Considering that the dataset to be compared is small, and the Brute Force algorithm is both simple to implement and has a better performance and accuracy for a compact dataset, it is expected to be a far better choice within the scope of this project and it will be the solution to be designed going forward.

Furthermore, to assure that the system selects only the best matches from both images, the cross checking algorithm allied with the brute force algorithm will also be implemented. As an extra step to guarantee the removal of some outliers that might still be present, the choice to implement one outlier removal method was also made, being the Slope-based Rejection algorithm the one chosen.

Even though both the software refactored implementation and the hardware implementation will have the same flow and use the algorithms discussed above, there is still a need to discriminate what needs to be done in hardware in the hybrid mode as well as what needs to be done in software for the software refactored mode and for both modes. For this, two sections representing each one of these implementations will be presented below.

2.11.1 Software Refactored

In the figure 2.11.1 below, it is possible to see a software overview of the system divided by the inputs and outputs that this program will have. Firstly, the inputs are the descriptors, computed by the BRIEF descriptor, from both images being compared. Then, a XOR operation to calculate the hamming distance and find the minimal hamming value, followed by the cross checking and the slope based algorithms is done. All this will be implemented in software, since this is part of the software refactored mode. Lastly, the output will be the best matches from both images.

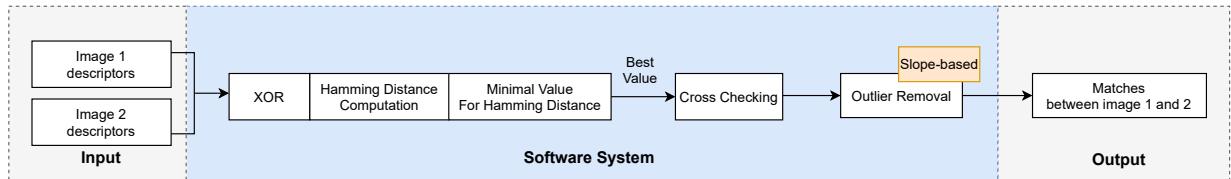


Figure 2.11.1: Matching Software Overview

Use Cases

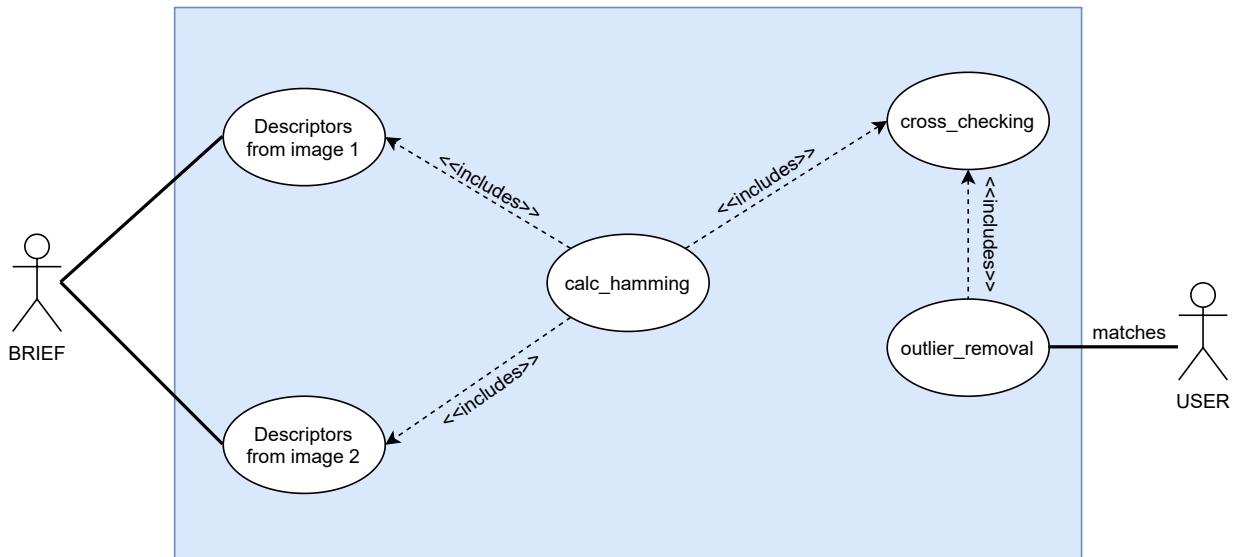


Figure 2.11.2: Matching Software Use Cases

The matching is the last step of the project as a whole, as such its primary actor is the previous stage, BRIEF, which will communicate directly with the matching process and send the descriptors, after acquiring said descriptors the process is sequential. Once the descriptors are obtained the cross checking will start which includes the calculation

of the hamming, reading the descriptors and lastly the outlier removal that will be done after the cross-checking. The user is the secondary actor in this stage, as it is the last one and once it is over the purpose of the entire system has been fulfilled.

2.11.2 Hardware

Concerning the hardware implementation, the inputs and output are the same. However, the modules chosen to be accelerated in hardware were the brute force algorithm with cross checking, as it makes all sense to accelerate these components in hardware due to the operations both algorithms make. The decision to not implement the slope-based outlier removal in hardware comes from the fact that the FPGA resources are limited and it is a long algorithm that does not pay off being implemented in hardware. An overview of this, with the hardware and software components distinguished, can be consulted in the figure ?? below.

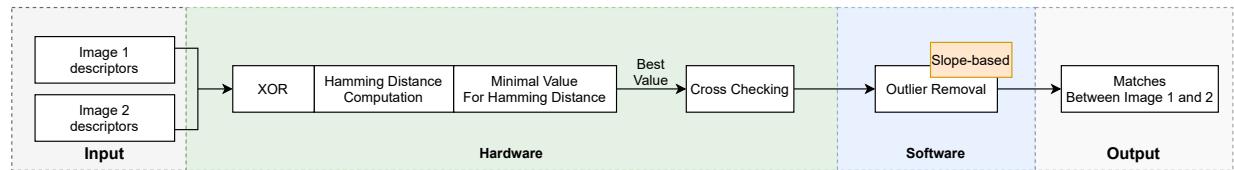


Figure 2.11.3: Contextualization of the hardware implementation with the project

Use Cases

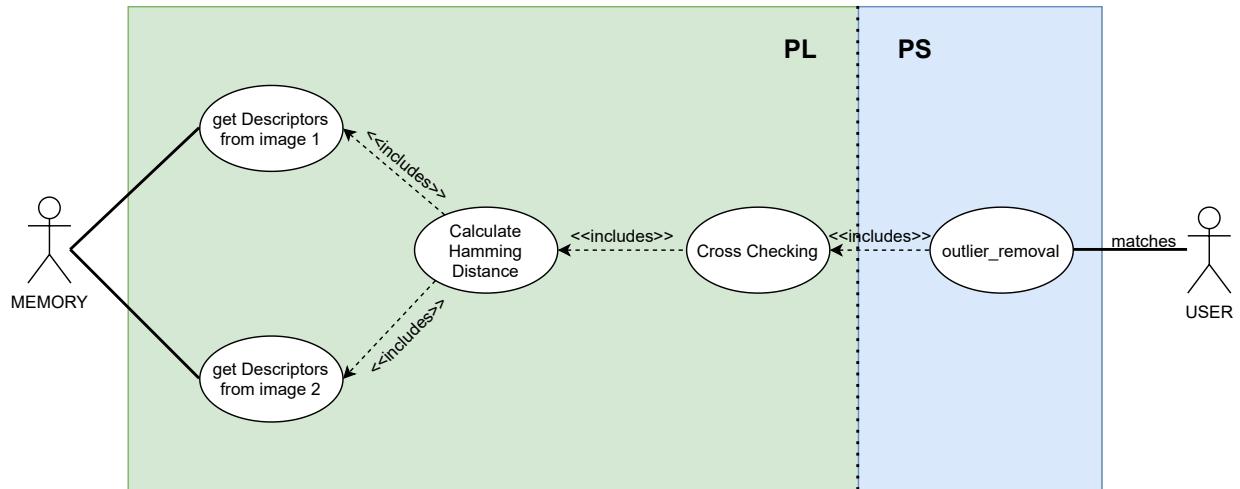


Figure 2.11.4: Matching Hardware Use Cases

The matching is the last step of the project as a whole, as such its primary actor is the memory, where the previous stage, BRIEF, stored the descriptors, without the descriptors from both images this stage cannot start, after acquiring said descriptors the process itself is sequential, although its hardware will be designed so that several cores are working in parallel to accelerate the entire process; once the descriptors are obtained the cross checking will start which includes the calculation of the hamming and reading the descriptors from the memory and lastly the outlier

removal that will be done after the cross-checking. The user is the secondary actor in this stage, as it is the last one and once it is over the purpose of the entire system has been fulfilled.

The obtaining of the descriptors, calculation of the hamming distance and the crosschecking are all steps that can take a long time to realise, as such they will be implemented in Programmable Logic so that they can be accelerated, since the cross-checking is going to be used then the outlier removal will have a lot less matches to remove as such it will be implemented in Processing System.

State Machine

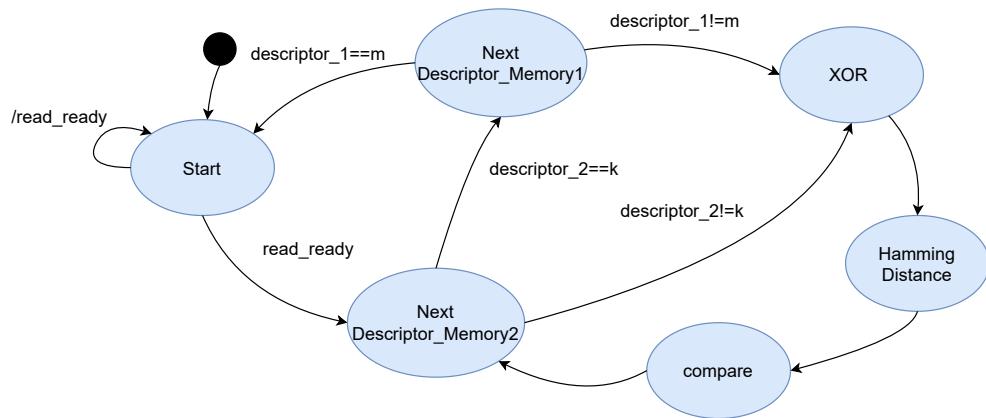


Figure 2.11.5: Matching State Machine

As mentioned before, the system must verify if two entities in different images are similar, this will be done through comparing the descriptors of both images (specifically, the term of comparison will be the Hamming distance). In terms of the state machine of the system, the process will be implemented with n matching cores architected as a pipeline so that it can be as fast as possible, it will go as follows: in the **START** state the system will wait until reading from memory is allowed (*read_ready*). Once it is, it will fetch the next descriptor of the second image (**NEXT DESCRIPTOR_MEMORY2** state) so that it can be compared to a descriptor of the first image. Then, each descriptor of image one will have to be compared to all the descriptors of image two.

The next state will depend on whether it has reached the end of the list of descriptors from image two or not; if it has, it will fetch the next descriptor from image one (**NEXT DESCRIPTOR_MEMORY1** state), otherwise it will compare the descriptor from image one with the descriptors from image two, which means performing the XOR operation (**XOR** state), calculating the Hamming distance (**HAMMING DISTANCE** state) and performing the comparison per se. Afterwards, when all descriptors of image one have been compared to all the descriptors of image two, it will return to the **START** state and thus the end of the list has been reached. The process will restart when two new set of descriptors of two different images have to be matched.

2.12 DSL

2.12.1 Objectives

The domain of this project is computer vision. For this purpose, a DSL will be developed, which has important design goals, that diverge from GPLs' goals.

- Domain-specific languages are more intuitive;
- Domain-specific languages are much more expressive in their domain;
- Domain-specific languages should exhibit minimal redundancy;
- Domain-specific languages are easier for the user to learn and configure
- Domain-specific languages increase the productivity (due to high abstraction)

The main DSL goal in this project, is to create a language that facilitates the work of the developer in the computer vision domain. This language will allow the user to insert the input parameters more simply and abstractly. It will also permit a generic user to be more productive, easily correct errors, and do optimisations. For this, a generative programming approach was used, through a repository with annotated code with variability points.

The Domain Specific Language will have the ability to receive and process the input script, wrote by the user. After that, this script will be syntactic and semantic verified, and then it will be generated the final code, without the annotations, substituted by the values that the user desired.

2.12.2 DSL constitution

A generative DSL process can be decomposed into three steps: the DSL script creation, scanning/parsing/semantic analysis, and finally the pretended code generation. In figure 2.12.1, we can see an overview of this process. The user is responsible for filling the DSL Script with high-level commands, according to its intended specifications, and then the DSL engine will receive the script as input, starting the scanning, parsing, and semantic analysis stage. This step is only possible because the DSL designer has previously defined the grammar of the language, a set of rules that describe elements according to the language syntax, enabling code validation. The designer is also responsible for annotating the source files according to the system variability points. If the syntactic and semantic analysis finishes without detecting any error, the annotations in the repository source files will be filled with the values selected by the user in the script. Finally, the code generation will take place generating the final code in a selected GPL (C++ in the example) ready to be compiled, completing the final stage of the DSL.

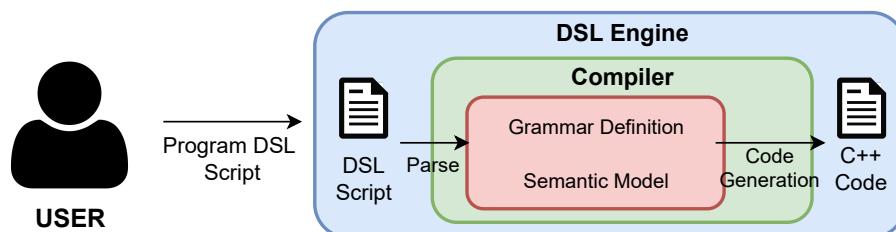


Figure 2.12.1: DSL constitution

2.12.3 DSL compilation states

Going deeper into the DSL, first, the user will have to write code using the DSL. Then, it will begin with lexical analysis where the entry code is separated into tokens, which are terminal characters specified in the DSL. Next, comes the syntactic analysis that is used to analyse whether the retrieved tokens exist in the grammatical rules defined. Thirdly, is the semantic analyser that checks if the meaning of the sentence makes sense regarding the DSL. Finally, if no errors have occurred up to this point, the code is generated.

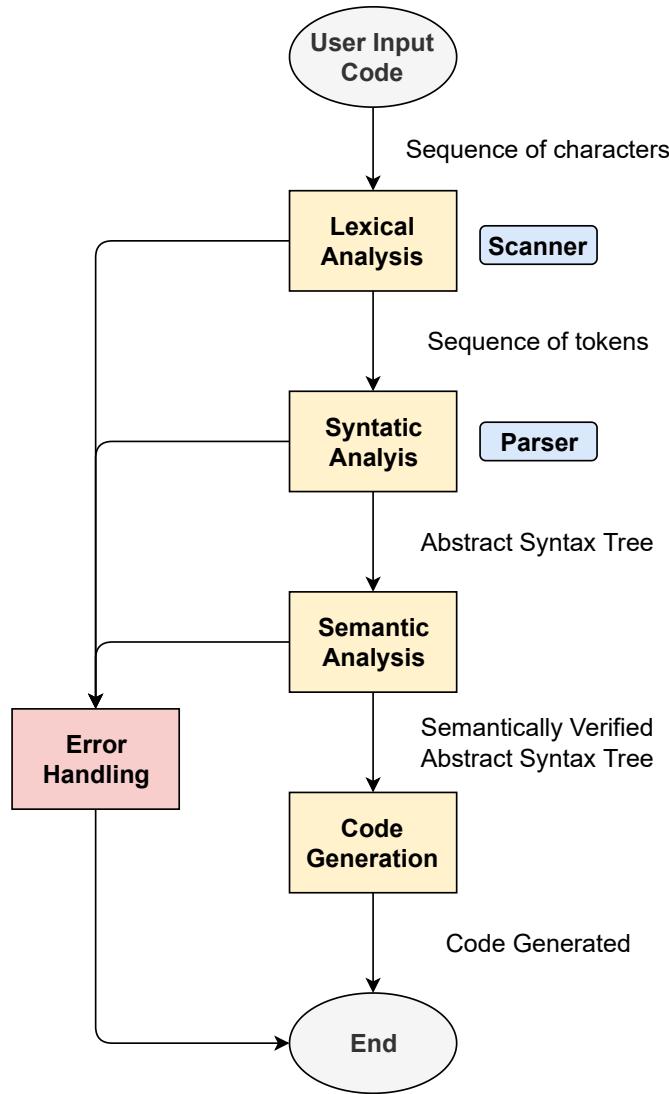


Figure 2.12.2: DSL compilation states

2.12.4 Code Generation

In the diagram of figure 2.12.3 is presented a completed view over the DSL. This DSL follows a generative approach, and that means that the designer's role is to create the DSL's model, as well as to create a repository with all the

annotated source-files. In this repository is present all the annotated code of all the algorithms with variability points. These variability points are points where the system can be customised, this means that all the parameters of the DSL's grammar are going to be variability points. When the user writes the DSL script, his code will be analysed and then the annotations of the annotated code correspondent to the algorithm that the user chose, will be changed and all the variability points will disappear, since their values were changed to the values desired for each parameter. After this, the code without variability points will be generated.

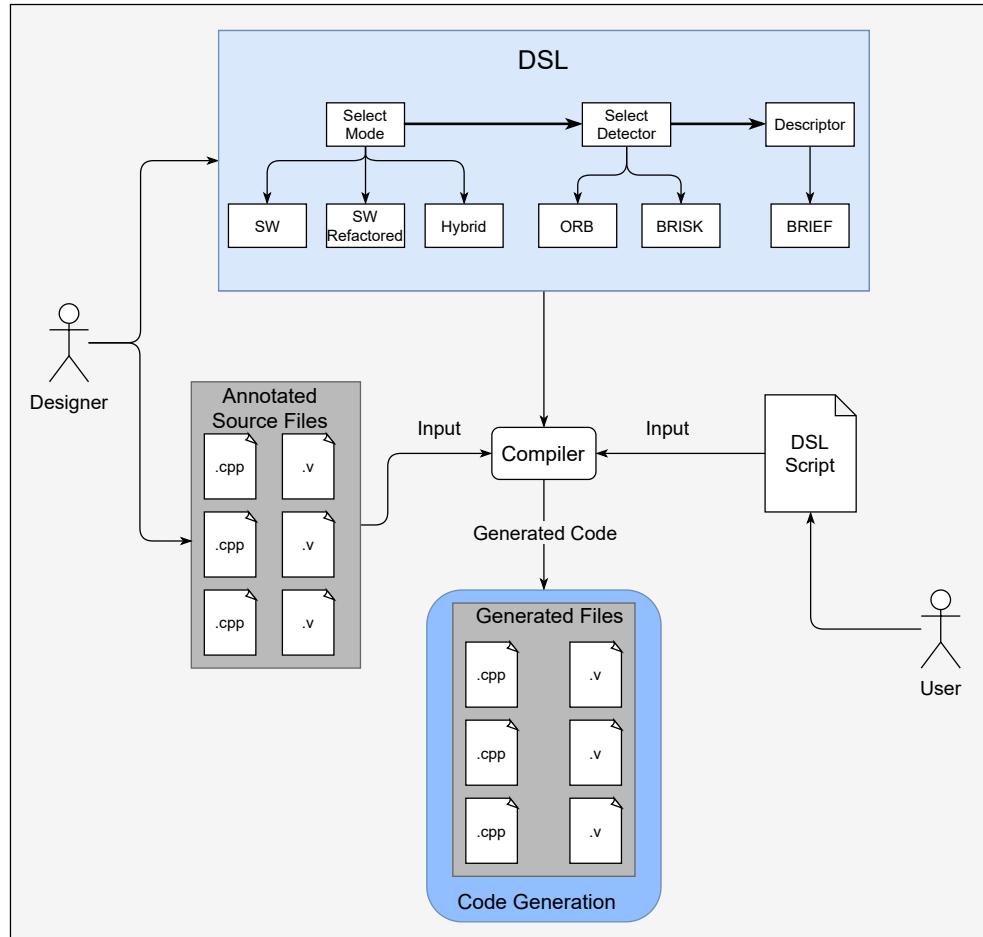


Figure 2.12.3: Code Generation Overview

2.12.5 Use Cases

A key concept of use cases modelling is that it helps to design a system from the user's perspective. It is an effective technique to communicate the system behaviour in the user's terms by specifying all externally visible system behaviour. In the figure 2.12.4 the use cases of the DSL is presented.

The primary actor is the user with the capability to customise the system. The user is capable of defining the operation mode of the system, this is, to choose if the algorithm used is implemented in software, software refactored or hybrid. Concerning the choice of the algorithm that will be used to the detection of the keypoints, the user can either choose between the BRISK algorithm or the ORB algorithm. Inside each one of these algorithms the user can also

choose the values of two parameters: Fast-N and threshold. In the descriptor algorithm (BRIEF) the user can choose the string's and the patch's sizes.

In the code generation, it may be necessary the creation of some new back-end entities depending on the code that will be generated, which are represented in the figure by the suspension points.

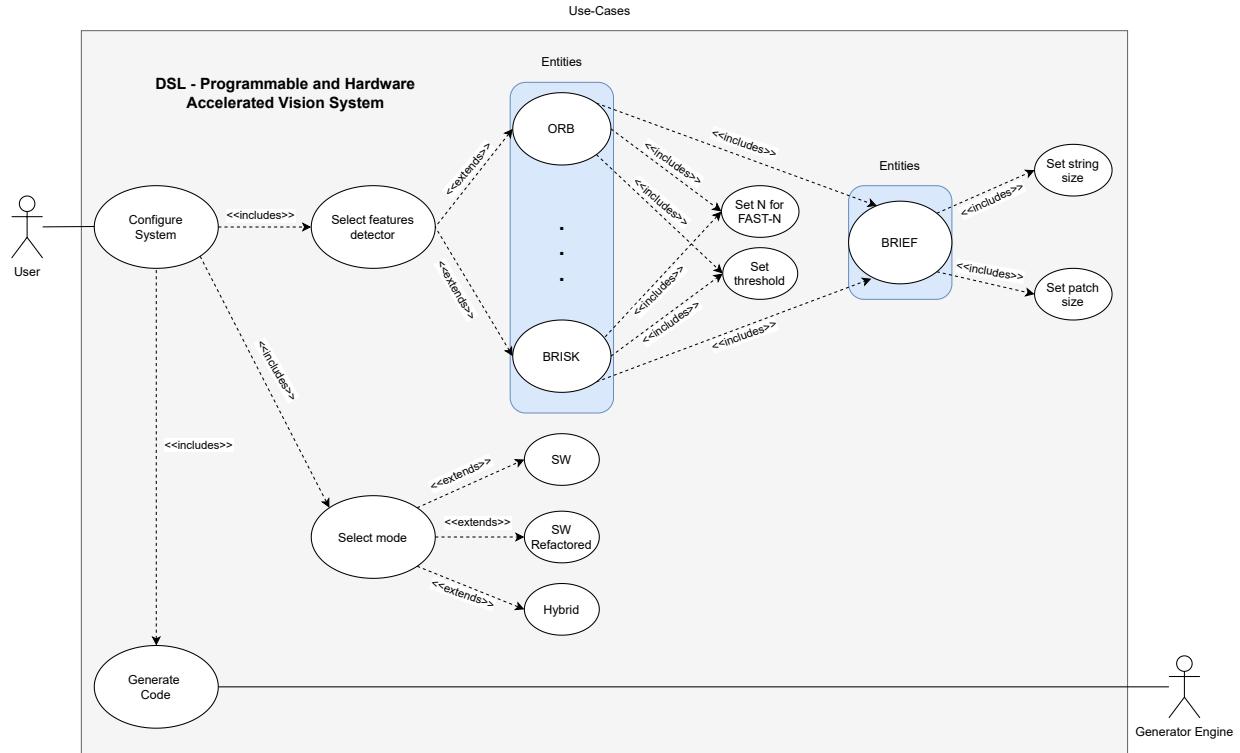


Figure 2.12.4: DSL use cases

2.12.6 Grammar

On the following tables, table 2.2 and 2.3, the entities that will be used and the keywords of the grammar are presented. Beyond the entities already presented (ORB, BRISK and BRIEF) there is also another abstract entity, named *System*, that will contain all the other entities and parameters of the system, as well as the operation mode of the system.

Entities	Description
System	Abstract element where all elements will be configured
ORB	Type that defines the detector used
BRISK	Type that defines the detector used
BRIEF	Type that defines the descriptor used

Table 2.2: Grammar Entities used

Keywords	Description
N-FASTN	Variable used for defining detection's FAST N value
Threshold	Variable used for defining detection's threshold value
StringSize	Variable used for defining descriptor string size value
PatchSize	Variable used for defining descriptor patch size value
OperationMode	Variable used for defining the operation type
Sw	Keyword used for defining software operation mode
SwRefactored	Keyword used for defining refactored software operation mode
Hybrid	Keyword used for defining hybrid operation mode
INT	Type used to set an integer value
STRING	Type used to set a string value

Table 2.3: Keywords used

2.13 Benchmarking/Profiling Tools and Compilers

This section will approach benchmarking and profiling tools, as well as compilers, so one can make an informed decision on what to use later on when profiling the software. With it, one can decide what to accelerate and proceed with the offload of the choke points, after the refactoring was taken into account.

2.13.1 OProfile

Overview

OProfile is an open source project that includes a statistical profiler for Linux systems, capable of profiling all running code at low overhead. In version 0.9.9, an event counting tool, ocount, was added to the project. OProfile is

released under the GNU GPL. It has proven stable over a large number of differing configurations and it is being used on machines ranging from laptops to 16-way NUMA-Q boxes. For versions 0.9.7 and earlier, the profiler consisted of a kernel driver and a daemon for collecting sample data. In version 0.9.8, with the introduction of `operf`, the legacy kernel driver/daemon method of collecting sample data was deprecated in favor of profiling with the Linux Kernel Performance Events Subsystem (kernel version 2.6.31 or higher). As of version 1.0.0, the legacy profiler has been removed. OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications. Several post-profiling tools for turning profile data into human readable information are available.

Features

- **Unobtrusive** - No special recompilations, wrapper libraries or the like are necessary. Even debug symbols (-g option to `gcc`) are not needed unless one wants to produce annotated source. Kernel patches are usually unnecessary, except when the running kernel may not yet support some newer processor models.
- **System-wide profiling** - All code running on the system is profiled, enabling analysis of system performance. It should be noted that root authority is required to do system-wide profiling.
- **Single process profiling** - Application developers will find the single process profiling feature very convenient since it does not require root authority, and profile data is collected only for the specified process (or command). This method has the added benefit of "following" fork/execs and collecting profile information on those child processes as well.
- **Event counting** - OProfile can be used to count native hardware events occurring in either a given application, a set of processes or threads, a subset of active system processors, or the entire system.
- **Performance counter support** - Enables collection of various low-level data and association with particular sections of code.
- **Call-graph support** - With an x86 or ARM 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.
- **Low overhead** - OProfile has a typical overhead of 1-8%, dependent on sampling frequency and workload.
- **Post-profile analysis** - Profile data can be produced on the function-level or instruction-level detail. Source trees annotated with profile information can be created. A hit list of applications and functions that take the most time across the whole system can be produced.
- **System support** - OProfile works across a range of CPUs, including the Intel range, AMD's Athlon and AMD64 processors range, the Alpha, ARM, IBM PowerPC, and more. OProfile will work against almost any 2.2, 2.4 and 2.6 kernels, and works on both UP and SMP systems from desktops to the scariest NUMAQ boxes. In addition, as of version 0.9.8, only 2.6 kernels are supported.

2.13.2 VTune

Overview

VTune Profiler (previously VTune Amplifier) is a performance analysis tool for x86 based machines running Linux or Microsoft Windows operating systems. Many features work on both Intel and AMD hardware, however advanced hardware-based sampling requires an Intel-manufactured CPU. VTune is available for free as a stand-alone tool or as part of Intel's oneAPI Base Toolkit. Optional paid priority support is available for the oneAPI Base Toolkit.

Features

- **Languages** - C, C++, Data Parallel C++ (DPC++), C#, Fortran, Python, Java, Go, OpenCL, assembly and any mix. Other native languages that follow standards can also be profiled.
- **Profiles** - Profiles include algorithm, microarchitecture, I/O, parallelism, system, thermal throttling and accelerators (GPU and FPGA).
- **Local, Remote, Server** - VTune supports local and remote performance profiling. It can be run as an application with a graphical interface, as a command line or as a server accessible by multiple users via web browser.
- **Get the Big Picture and the Critical Details** - With today's variety of computer architectures, a performance profiler must be able to diagnose many kinds of bottlenecks. Intel VTune Profiler includes: System and Application Overview Profiles: These identify where to focus tuning or how to configure systems. Also, they collect fewer data and can run for a longer time; Focused Profiles: These take a more detailed look at different classes of bottlenecks such as compute, memory, threading, accelerator offload, throttling, or I/O. Precise profiling information is annotated onto the code source.
- **Turn Raw Data into Answers** - Performance optimization is a bit like solving a mystery. One gathers the evidence, analyzes it, and makes sense of what's actually going on. Summaries, tips, filtering, and annotations make people more productive. Source View: See the profiling results on the source; Timeline Filtering: Focus on bottlenecks as they change with time; Summary Reports: Get the big picture and tips for further analysis.

2.13.3 Microsoft Visual C++ Compiler

Microsoft Visual C++ is a compiler for C/C++ programming languages by Microsoft. It was initially a standalone product but later became part of Visual Studio being nowadays the default compiler for all C++ projects on the IDE. It features tools for developing and debugging C++ code, especially code written for the Windows API, DirectX and .NET.

2.13.4 Intel C++ Compiler

Intel C++ Compiler is a group of C/C++ compilers made by Intel Corporation. It is available for Linux, Microsoft Windows and Mac OS platforms. The compilation is supported for the Intel processors IA-32, Intel 64, Itanium 2 and XScale. This C++ compiler for x86 and Intel 64, features an automatic vectorization tool that generates SIMD, SSE and SSE2 instructions. Since its introduction, ICC has been greatly used for the development of Windows Applications.

3 | Design

In this stage, the hardware specifications for the BRISK detection stage as well as the interface with the Processing System will be approached, with the intent of outlining a strategy for the implementation going forward. As the design proceeds, one will specify their interfaces and main components, including the IPs planned to be used to speed up and standardise development.

3.1 Interface

In order to have an efficient interface with the ZYNQ7 Processing System allowing for stall-free data transactions between the Programmable Logic and the Processing System, one intends to devise the hardware interface presented in fig. 3.1.1. The latter will integrate the two counterparts of the overall system, hardware and software, and the software will generally dictate the behaviour of the Programmable Logic fabric. In a more detailed manner, the aforementioned interface is intended to be DMA-free, accessing directly a shared memory region within the Processing System's DDR. This was thought out to reduce the amount of stalls caused by PL to PS interrupts. On the one hand, to achieve this, the Processing System will behave as a Master of the AXI4-Lite bus, to configure the system through the underlying memory space (registers). The memory mapped protocol will take advantage of the general-purpose port GPO. On the other hand, the Programmable Logic Master will use two high-performance slave ports (HP0 and HP2) to ultimately access the target memory region of the DDR. The data exchange through these ports will be done using the AXI4-Stream protocol. Additionally, it is imperative to have a memory controller that dictates how the image data is stored on the BRAMs. In this case one is only referring to the detector's BRAM, but from the global point of view, there will also exist a description-dedicated BRAM. Lastly, it must also exist an interrupt controller responsible for generating the hardware-to-software interrupts. The blocks highlighted in red are meant to represent Xilinx IPs.

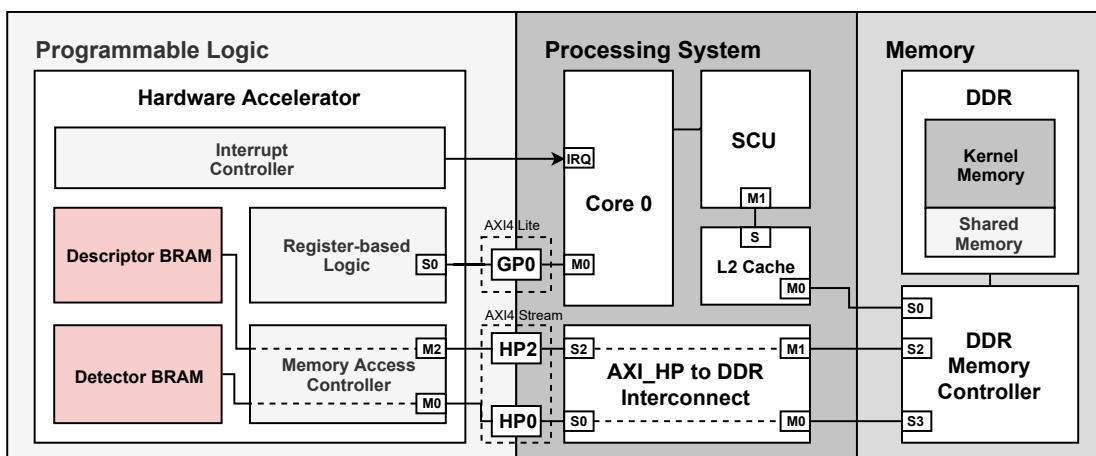


Figure 3.1.1: Hardware Architecture: Processing System Interface

3.2 ORB Detection

3.2.1 Patcher

The main objective with this design was to keep the resource footprint low while not sacrificing performance. With that in mind, the proposed schematic in figure 3.2.2 depicts a mechanism that relies on interconnected BRAM FIFOs and shifting file registers. As each brightness value enters the first position of the register file matrix, it also is passed into the first index of the BRAM FIFO, subsequently pushing all other values to the next index. The last index of each FIFO not only links to the corresponding line in the file register matrix, but also to the first index of the one below. Values in the register files are shifted to the next column at each time step and discarded after reaching the last column. This arrangement causes the image to appear inverted on both axis but that is irrelevant for the purpose of the FAST-n algorithm.

The pixel streamer module sequences the image's pixels' brightness levels, orderly outputting each pixel's value at every time frame. When the end of a line is reached, the following line begins, uninterrupted. This causes the occurrence of an edge case where the output is meaningless. As the image's pixels' brightness levels are streamed continuously, the system cannot know when the values in the register file matrix correspond, in reality, to different lines in the image. To overcome this, an auxiliary module, the *Coordinate counter*, counts the clock signal to keep track of the pixel's index. When the image width size is reached, the counter is reset and the height coordinate incremented. Based on these coordinates, the *Validity check* module enables or disables a flag to signal downstream modules that the outcoming data is invalid.

It is imperative that the whole system is parametrical so different sized images and variable order FAST-n algorithm work. To accommodate this, the BRAM FIFO length must match one of image's dimensions. The bresenham circle size also changes so corresponding masks must be applied to the register file matrix (figure 3.2.1). Lastly, the condition within the *Validity check* module also changes based on the FAST-n order.

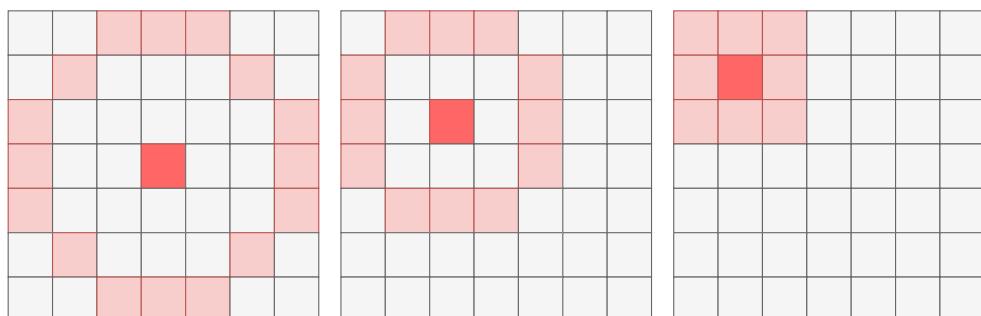


Figure 3.2.1: Bresenham circle mask for FAST-12, FAST-9 and FAST-5, respectively

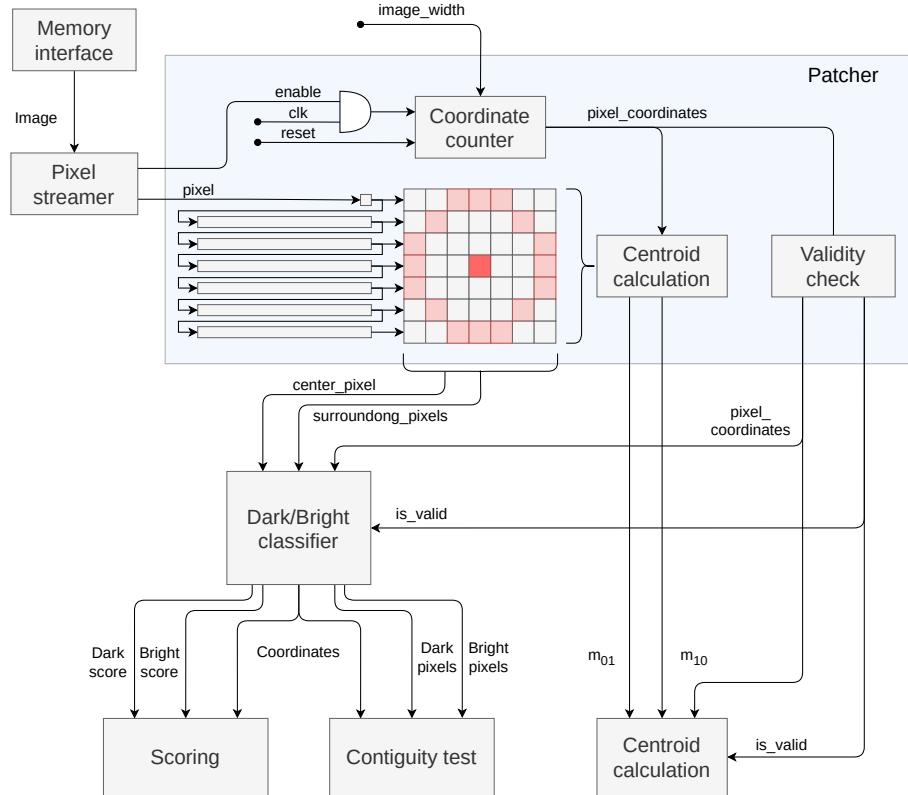


Figure 3.2.2: Patcher design overview

3.2.2 Dark/Bright classifier

This module is responsible to compare each pixel from the bresenham circle to the centre pixel. In order to perform this task as fast as possible, the comparisons should be done in parallel. Additionally, to increase throughput, the operations described in 2.1.3 regarding this module should be pipelined. This adds a slightly higher resource footprint but highly benefits performance.

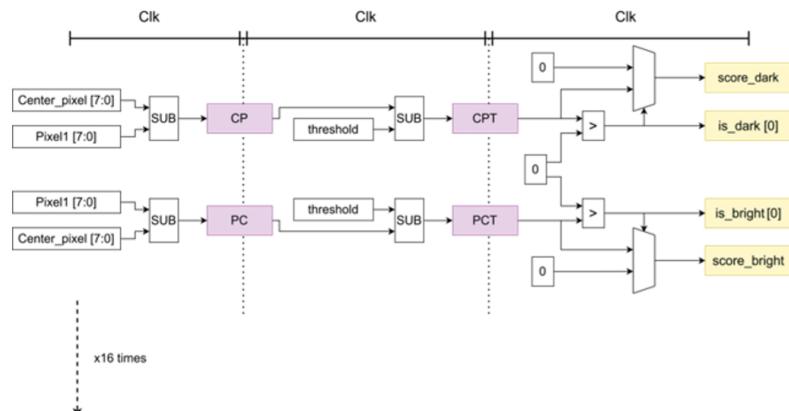


Figure 3.2.3: Pipelining stages of the dark/bright classifier stage

3.2.3 Contiguity Counter

Since the scoring algorithm has to count the number of active contiguous bits in a circle it was needed to start by finding a way to do so. By the input being an array which is linear, instead of circular, one has to pay attention because there is no start and end to the circle.

Our approach consists of running through the array twice, while counting the number of active bits, comparing and restarting the counter every time zero is encountered.

The counter will only be saved when it is greater than the current maximum number of contiguous bits. A possible problem is if the n-bit circle is fully active, the counter would reach twice the size of the circle. Thus, a workaround to this would be to limit the counter to the width of the circle. Also with the size of the circle being varied and the input size being fixed to 16-bit the circle will be placed from the 0-bit up to n-bit ($n = \text{size of the circle}$), and all other information should be ignored. By having the input's size fixed and varying the circle's size, one has to only take into account the part of the input that is intended to be used. This will be done by applying a mask when accessing the input data, thus restoring the index to the right position.

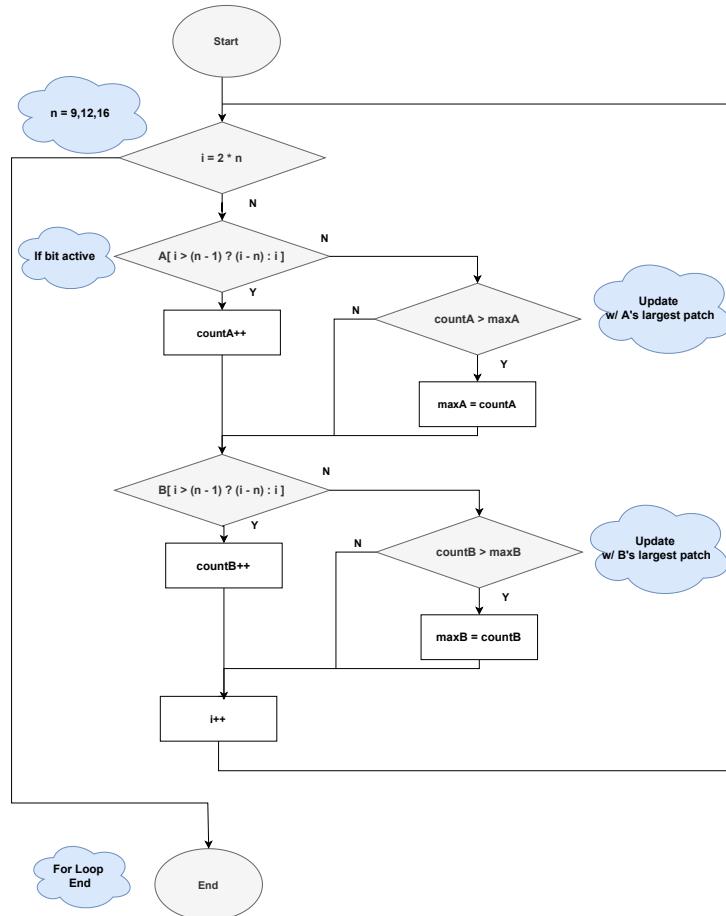


Figure 3.2.4: Contiguity Counter Algorithm.

3.2.4 Scoring

In order to proceed to compute the final key-point score, after getting both scores they need to be compared against each other, and the greater one is rerouted to the output. Since the contiguity counter needs the size of the circle, it should operate on the input n will be used to select of 3 possible sizes 8,12,or 16.

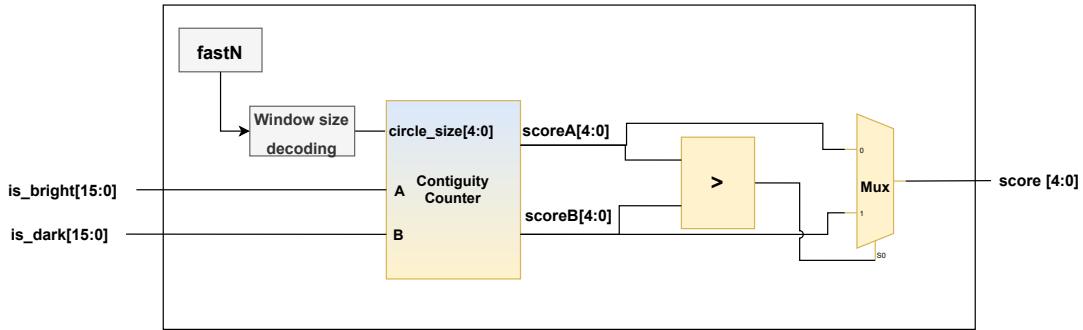


Figure 3.2.5: Scoring Module.

3.2.5 Contiguity Test

Since the contiguity test's function is to check whether there exist n -contiguous bits in the circle, this could be easily accomplished by using the previously stated block, the Contiguity Counter. With the later one being able to count the largest amount of contiguous bits for either a 8,12,16-bit circle, then by simply comparing both scores with the number of n contiguous active bits the key-point is classified as corner or not.

When it comes to the number of contiguous bits to look for, it will be selected, similarly to the circle size needed by the counter, using the input n to select the optimal value of contiguous active bit's for each circle size. Them being 5 for a circle of 8, 9 for a size of 12 and 12 for a 16-bit circle.

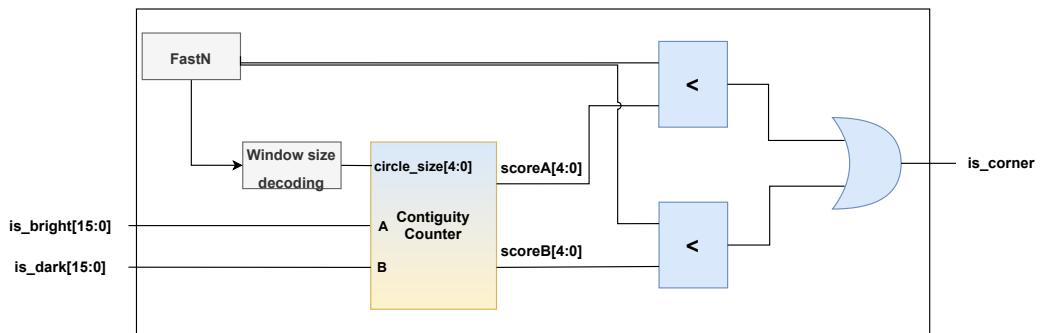


Figure 3.2.6: Contiguity Test Module.

Final Module - Corner Score Computation. The final module will be the combination of the 2 previously noted blocks, the Scorer and the Contiguity Test.

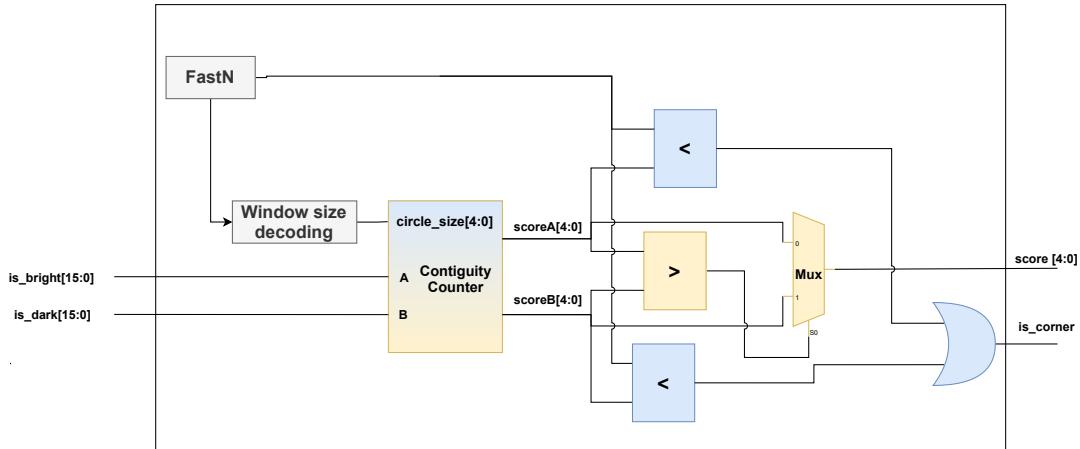


Figure 3.2.7: Corner Score Computation Module.

With the intent of the module to be capable of keeping up with the workload from the detector, it is added a pipeline to the module. With this, it can receive a new set of inputs, each clock cycle and after an initial latency, equal to the number of pipeline stages, it can produce a new output at each new clock cycle.

Therefore, it is required to add registers along the circuit, so that it ends up being divided into stages, for each stage can act with its current valid information. The signals like *input_enable*, *output_valid* and *stall* have been omitted for clarity purposes.

2-stage pipeline

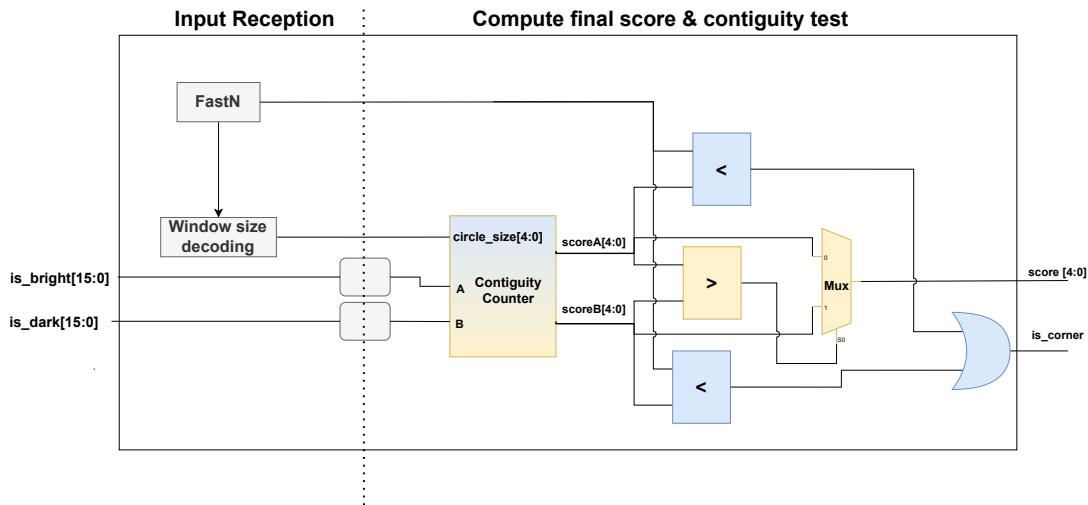


Figure 3.2.8: Corner Score Computation Module.

When *reset* signal is active, *output_valid* should clear all the current information that are still in the pipeline. *Stall* will prevent the pipeline from passing information between each stage, thus when released it will start where it stopped, not losing out on any of its data. To avoid unnecessary computation, power consumption as well as a control signal *input_enable* was added.

3.2.6 Non-maximum Suppression

ORB detection third stage consists of corners' non-maximal suppression. This module implementation is based on the sparse point algorithm whose detailed explanation can be found in the BRISK section. The major difference between these two detection stages is spatial awareness. Meanwhile BRISK takes into account the three dimensions notion when filtering the corners ORB only requires a two-dimension filter application.

Rotation Invariance

Hybrid Approach. Considering this implementation is software oriented, Vitis IDE is used in order to create a baremetal application. This application itself is very simple and straightforward as the hardware implementation problems previously mentioned, which greatly increase its complexity, are non-existent in a software implementation. A timer is used to calculate the latency of the application as shown in the flowchart below.

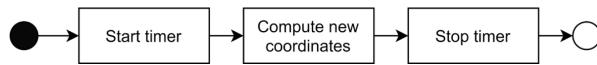


Figure 3.2.9: Flowchart of execution time measurement

Hardware Approaches. Recurring to a hardware implementation, regardless of the utilisation of floating-points or not, the square root of the moments of the patch's sum used in the computation of the cos and sin of the rotation angle can always be calculated in a step common to any subsequent method because the result is always a integer number. Put this, the first step of any hardware approach is represented in the Figure 3.2.10, where the patch's moments are multiplied by themselves to get the square and then are summed.

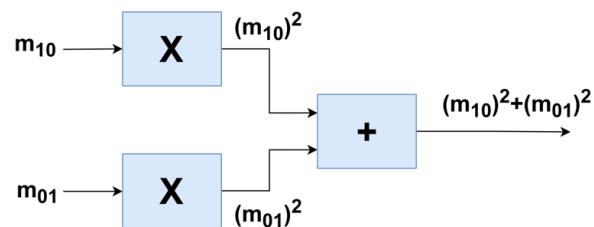


Figure 3.2.10: - First step of any hardware approach

Regarding the number of bits needed to store relevant variables along the algorithm, the following study is made:

- Inputted keypoint's coordinates, as well as the outputted oriented keypoint's coordinates are a concatenation of a keypoint's X coordinate and Y coordinate. Since this project is designed to process images with a width of 320 pixels and a height of 240 pixels, the maximum X size is 320 and the maximum Y size is 240. Hence, the X coordinate needs to be stored in a 9-bit variable and the Y coordinate needs to be stored in a 8-bit variable, forming a total size of 17 bits for the concatenation.

- The moments of the patch, computed as demonstrated in the Moments of the Patch section, possess their maximum positive value when all the pixels of the patch have an intensity of 255. Considering this, the maximum positive value for them is

$$\begin{aligned} & [(255 \times 7) + (255 \times 7 \times 2) + (255 \times 7 \times 3) + (255 \times 7 \times 4) + (255 \times 7 \times 5) \\ & \quad + (255 \times 7 \times 6) + (255 \times 7 \times 7)] = 49980 \end{aligned} \quad (3.1)$$

. Therefore, m_{01} and m_{10} need to be stored in variables of 15 bits.

- Having the maximum values for the moments of the patch it is possible to estimate the maximum value of the previously mentioned common part, i.e., the square of the sum of the moments of the patch, as $49980^2 \times 2 = 499600800$. To fit this number, a variable of 33 bits is required.
- The maximum truncated value for the denominator of the cos and sin of the rotation angle is then

$$\sqrt{229408200} = 70682 \quad (3.2)$$

, which demands a variable of 17 bits to be stored.

- Finally, the sin and cos of the rotation angle, being naturally numbers between 0 and 1, have a maximum size equal to the scalar N and the variables for storing them need to have an according number of bits.

Hardware with Floating-Point Numbers Approach. As it was previously mentioned in the Analysis Phase, this method is capable of executing its task with a very high precision output due to the single-precision floating-point used, which has 12 decimal places. However, when using single-precision floating-point, each value is represented by 32 bits, regardless of its actual size, which drastically increases the resource usage in storing and manipulation of these values. In order to convert a value to floating-point and manipulate values in the IEEE 754 single-precision representation, an IP (Intellectual Property) core is used. This IP core, Floating-point (7.1), is capable of a large number of logic and arithmetic operations upon values in the floating-point representation.

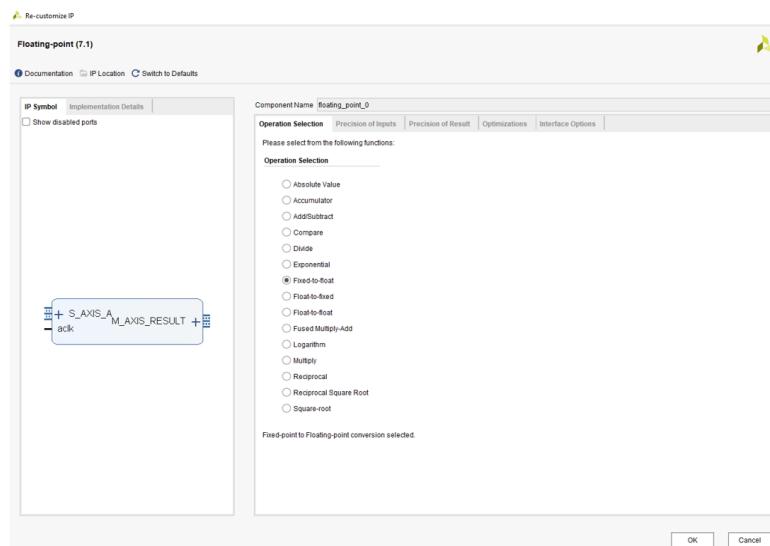


Figure 3.2.11: Floating-point (7.1) IP core customisation window

In the Floating-point core's customisation window it is possible to configure the precision of the inputs and outputs, which only matters in the cases of a fixed-to-float and float-to-fixed conversions as all the floats are in the 32-bit single-precision format. Configuring the IP core instances with the correct input/output precision, resource wastage is avoided, and a much lighter implementation is conceivable.

As the IP core acts as a black box which is already implemented, it is possible to verify how many resources are going to be used by each instance right after the configuration. What was thought to be the main problem of this method is confirmed as a single instance of the IP configured for a division of two floats requires about 795 LUTs (\approx 5% total LUTs available) and 1386 Flip-Flops (\approx 4% total LUTs available).

Name	Progress	LUT	FF	BRAM	URAM	DSP	LUTRAM
✓ floating_point_2_synth_1	100%	795	1386	0.0	0	0	37

Figure 3.2.12: Resource usage of Floating-point IP core configured for division operation

The high latency (time between data input and processed data output) of some operations with floating-point numbers is considerably high, which is a problem. To solve this, a pipeline architecture is designed in order to compensate the high latency of some operations. Each stage is dimensioned so that they all have approximately the same latency, as this increases pipeline efficiency. A diagram of the design is presented below.

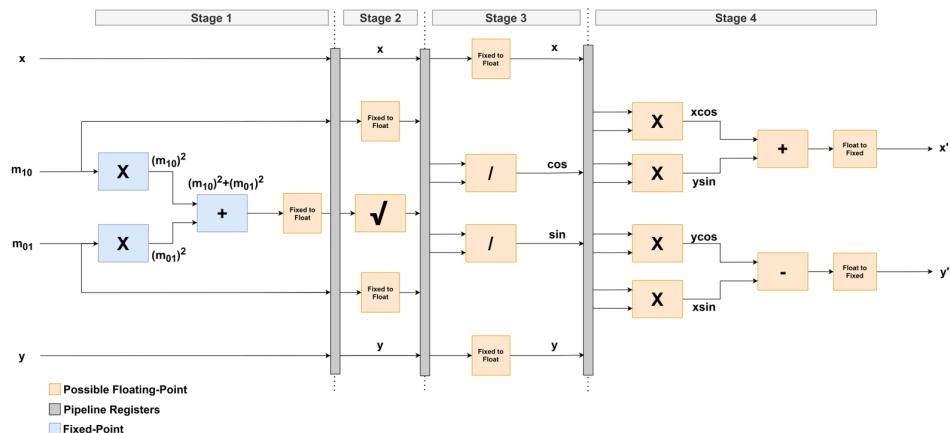


Figure 3.2.13: Diagram of the Hardware with Floating-Point Numbers Approach

The high latency (time between data input and processed data output) of some operations with floating-point numbers is considerably high, which is a problem. To solve this, a pipeline architecture is designed in order to compensate the high latency of some operations. Each stage is dimensioned so that they all have approximately the same latency, as this increases pipeline efficiency. A diagram of the design is presented below.

Hardware with Fixed-Point Numbers Approach. As an alternative to the floating-point approach, this approach is designed to operate only with fixed-point numbers and make use of a scalar to facilitate rounding results to the operations that require decimal places precision, as it was seen in the Analysis Phase. Since this approach is expected

to have a very low execution period, fast enough to never have a new input while still in processing, a pipeline architecture is expendable.

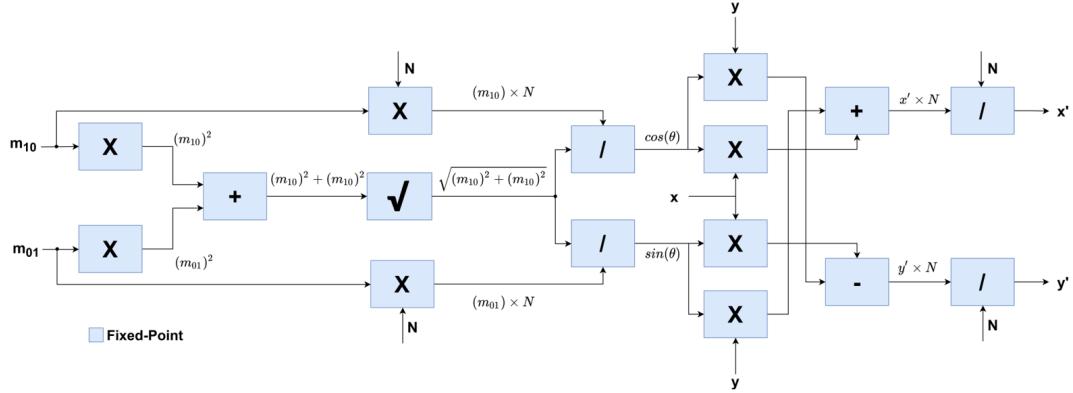


Figure 3.2.14: Diagram of the Hardware with Fixed-Point Numbers Approach

The only IP used in this approach is the Xilinx CORDIC and it is employed to perform the square root of the squared patch's moments sum

$$\sqrt{m10^2 + m01^2} \quad (3.3)$$

.The instantiation is done following the customisation. The two things worth mentioning in this customisation is the fact that the pipelining mode was set to “No Pipelining”, what drastically reduces its latency, and the input was set to 28 bits because that is, as it was deduced in Hardware Approaches, the maximum number of bits the input of this IP (sum of the patch's moments squares) requires to be stored. This IP, with the set customisation settings, consumes 317 LUTs.

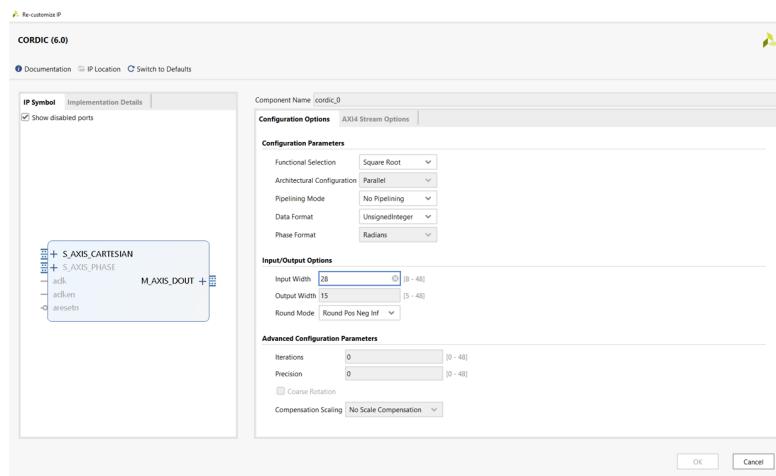


Figure 3.2.15: CORDIC IP Customisation Window

Besides this customisation, the flow control of the IP is also set to “non-blocking” so it does not block the execution and behaves as a wire. To instantiate this IP with the mentioned customisation, the template shown in the Code snippet 1 is employed.

Listing 3.1: Instantiation template of the Xilinx CORDIC IP with the desired customisation

```

1   cordic_o.sqrt(
2     .s_axis_cartesian_tvalid(1),
3     .s_axis_cartesian_tdata(input),
4     .m_axis_dout_tvalid(m_axis_dout_tvalid),
5     .m_axis_dout_tdata(output)
6 );

```

3.2.7 Software Refactored

The purpose of doing the refactoring is to improve the performance of the source code, in this case, is improve the OpenCV code to do a specific thing. That said, the refactored functions will do the same that the OpenCV functions do, but will be less generic:

Listing 3.2: create() Method

```

static Ptr<ORB>cv::features2d::create (int nfeatures=500, float scaleFactor=1.2f, int nlevels=8, int edgeThreshold=31, int
firstLevel=0, int WTA_K=2, int scoreType=ORB::HARRIS_SCORE, int patchSize=31, int fastThreshold=20)

```

This is the create function of the openCV, as it can be seen, it has a lot of parameters what can be excluded in the refraction to simplify the code. As in the global project in the ORB detection, it will be only used the FAST detector, in the refactoring Create function, it will only have the nFast as parameter to choose between the FAST-5, FAST-9, FAST-12.

Listing 3.3: detect() Method

```

virtual void cv::features2d::detect(InputArray image, std::vector<std::vector<KeyPoint>> &keypoints, InputArrayOfArrays masks=
noArray())

```

In the refactoring detect function instead of receiving the parameters image, keypoints vector, and the image where the keypoints will be afterwards, it will only have the image and the threshold as inputs.

State Chart. In the figure 3.2.16 the function orb_create the selected fast-n type will be selected according to the user's choice. As far as the detect function is concerned, the first step is to read the image. As the image is scrolled through it will classify each pixel as dark or bright depending on the chosen threshold. The second step is to check if the pixel that was classified as dark or bright is a keypoint. After the whole image has been scrolled, it is checked for neighbouring keypoints, either horizontally or vertically. If this is the case, the keypoint with the lowest score is suppressed and the keypoints are put in a linked list for the BRIEF stage.

For a better understanding, the figure 3.2.18 shows the general flowchart of the built detect function.

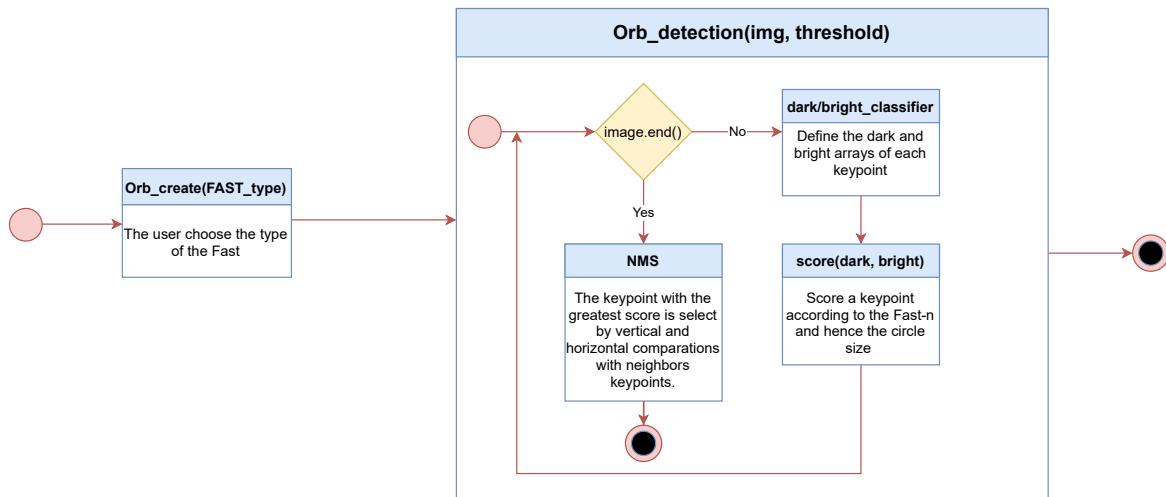


Figure 3.2.16: ORB - Software Refactoring State Machine

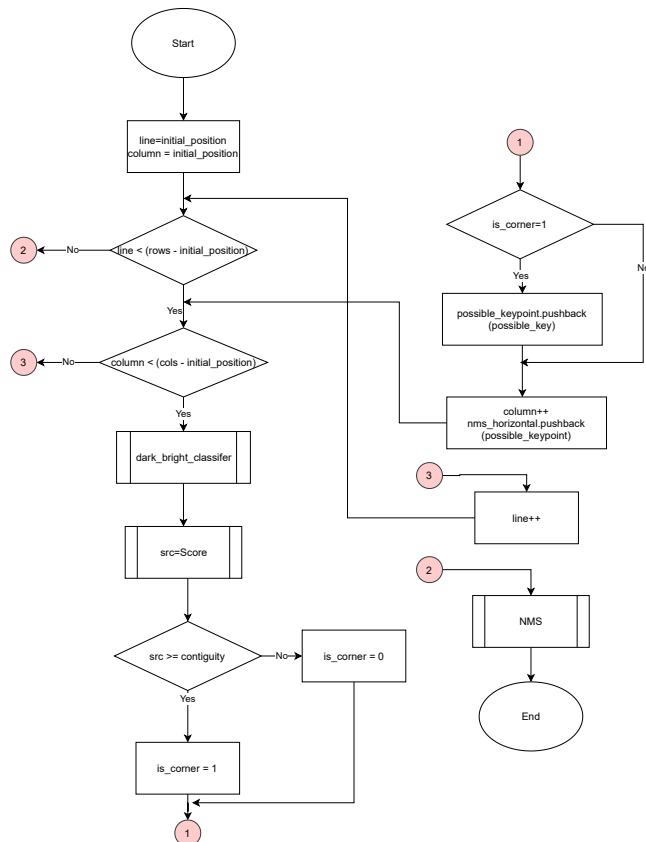


Figure 3.2.17: Flowchart detect function

Class Diagram The software refactoring will be implemented in C/C++ program language, so The class diagram shown below represents the relationship that exists between the ORB class, the structures point_t and nms_data_t, and the enumeration FAST_type.

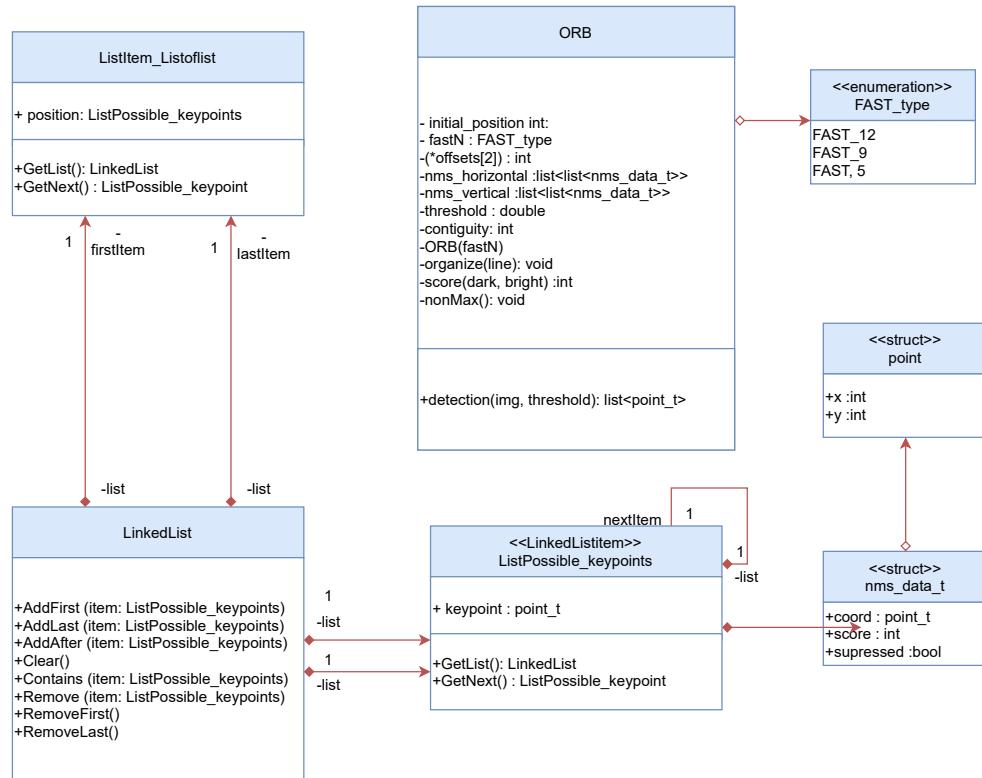


Figure 3.2.18: Flowchart detect function

3.3 BRISK Detection

The section will further specify the matters regarding the core of the detection module, the classifier. Furthermore, it will also approach the detection controller, which dictates the overall flux of data of the system.

3.3.1 Image Slicing

Regarding the image store within the PL fabric, one opted for a solution that doesn't need as many resources as storing the image as a whole in the hardware. The approach follows a simple sliding window concept as depicted in fig. 3.3.1. The intent is to operate on the first seven lines of image ($7 \times \text{width}$), represented by the letter A, sliding the 7×7 window through the seven lines horizontally, as seen in ①. Additionally, a fresh line is supposed to enter the system at each iteration, while the window still slides through the other seven lines. As one can see in ①, the first seven lines are processed while a new line (highlighted in darker grey) is added to prepare the second iteration. In ②, the window slides down one line, and proceeds to shift and process the new line set, whilst the next line of the image is loaded onto the spot left by the downwards sliding of the window. Moreover, the same idea applies for ③, and here one can really see that the system behaves as a circular buffer both vertically and horizontally, allowing for large amounts of data processing with less memory footprint. Its important to note that the 7×7 sliding window intends to represent the current Bresenham circle being analysed.

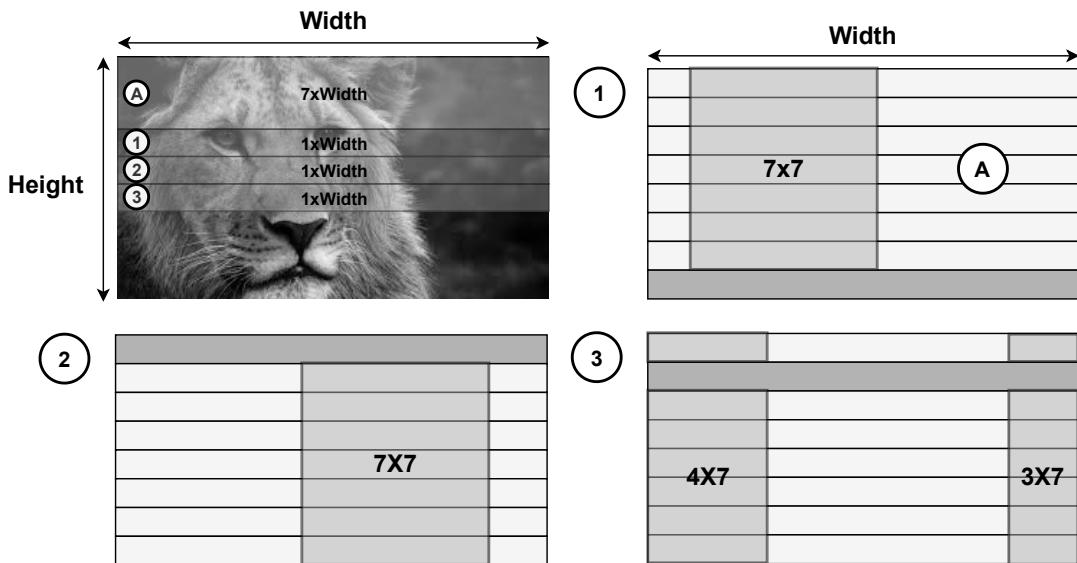


Figure 3.3.1: Image Slicing: Overview (No Limitation)

Window Limitation. The fig. 3.3.2, represents the exactly the same proposed solution for image analysis, but with the 4-column limitation sliding window. This is relevant because of the trade-off performed when designing the Block Random-Access Memory. This will be explained further in section 3.3.1 and section 3.3.1.

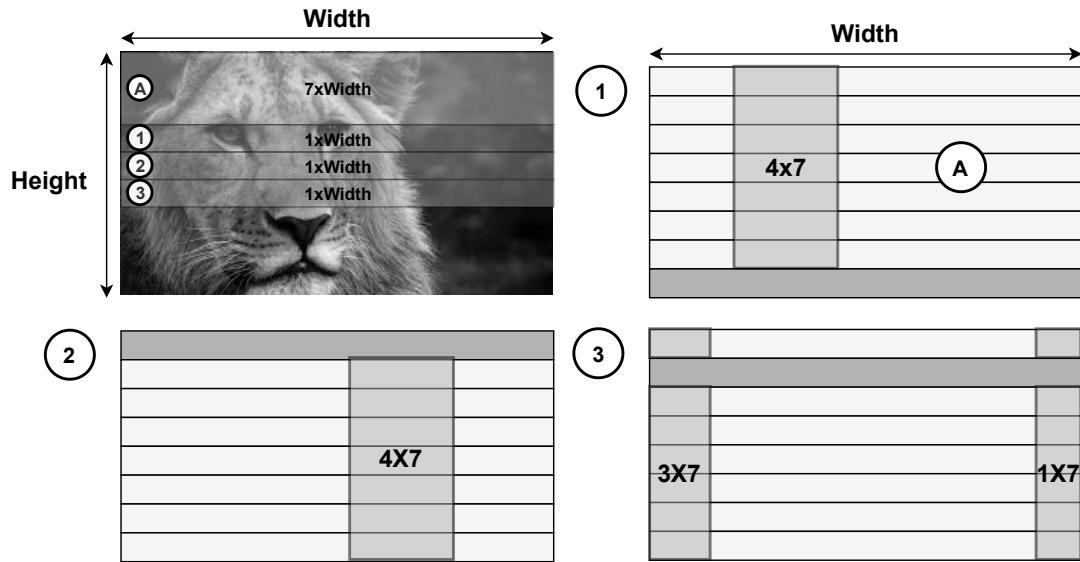


Figure 3.3.2: Image Slicing: Overview (Limited)

Reading

Memory read operations will be done in 32 bits per line in memory, to strike the best balance between resource usage and performance. This is explained further in section 3.3.3. This is done from right to left, instead of left to right, as illustrated in fig. 3.3.3, to simplify the design and testing process of the connected hardware units.

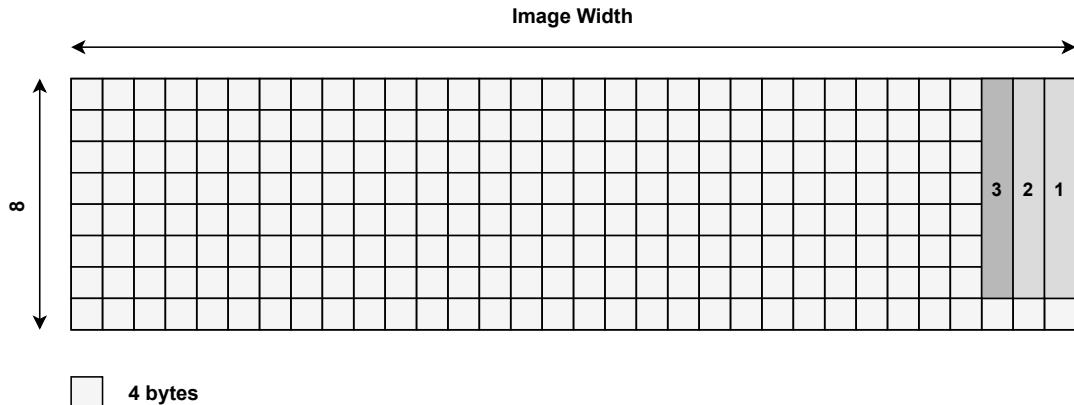


Figure 3.3.3: Memory Read Operation

Writing

Memory write operations to the image slice will also be done 4 bytes at a time, as that is the width of the bus. It will only be done in a single line at a time, though, as there is only one High-Performance AXI-Stream port dedicated to the Detector's memory slice. This is also explained further in section 3.3.3.

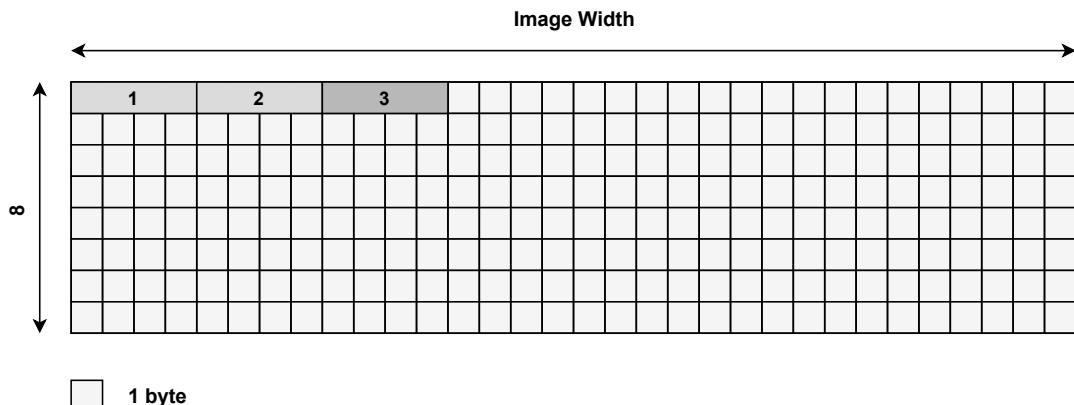


Figure 3.3.4: Memory Write Operation

3.3.2 Dynamic Mapping

In order to address and read from blocks present in any consecutive set of lines in the slice, mapping the output of each RAM block directly to the input of the Register File is not a functional strategy. Instead, each line in memory needs to be able to connect to any line in the input of the Register File. As such, the connection between the lines in these entities will be established dynamically, such that, when $\text{memory_out}[x] \rightarrow \text{rf_in}[0]$, $\text{memory_out}[(x+1) \& 7] \rightarrow \text{rf_in}[1]$, ..., until $\text{memory_out}[(x+6) \& 7] \rightarrow \text{rf_in}[6]$. This is illustrated in fig. 3.3.5.

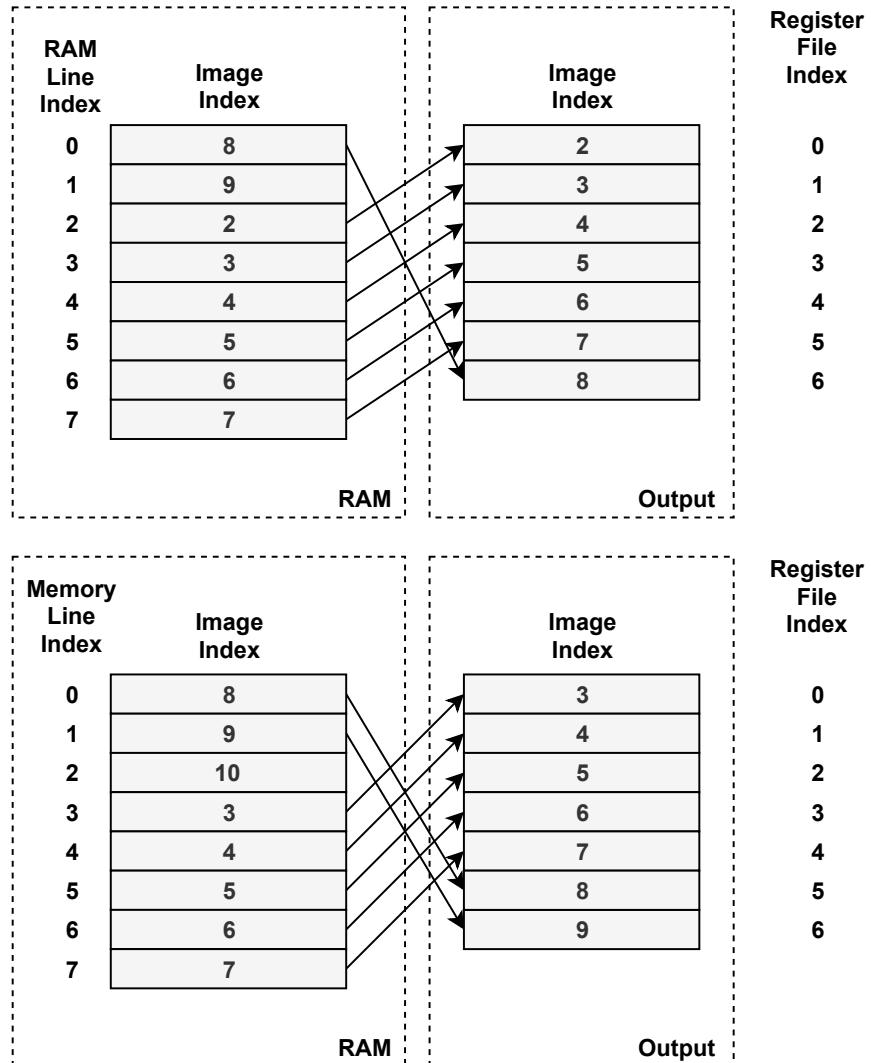


Figure 3.3.5: Dynamic Mapping Overview

3.3.3 Controller

In interfacing the Bright/Dark classifier with the local image storage, independent control functionality needs to be implemented to guarantee the readiness of the correct information at all times. For that, an independent controller was devised, as schematised in fig. 3.3.6.

As the controller plays an important role in controlling the access to memory, it serves as a common point for timing and controlling the flux of information. It serves the purpose of controlling what information goes into the classifier, through the coordinate system represented in a simplified way as `coord_fetch`. An interface was also created with the Non-Max Suppression stage that provides a calculation of the coordinates of the circle to whom the information arriving from the scoring stage belongs. In order to save resources and simplify the design of the further stages, instead of propagating the value of the coordinates throughout the stages, the value should be calculated based on the current coordinates at the controller's internal counters.

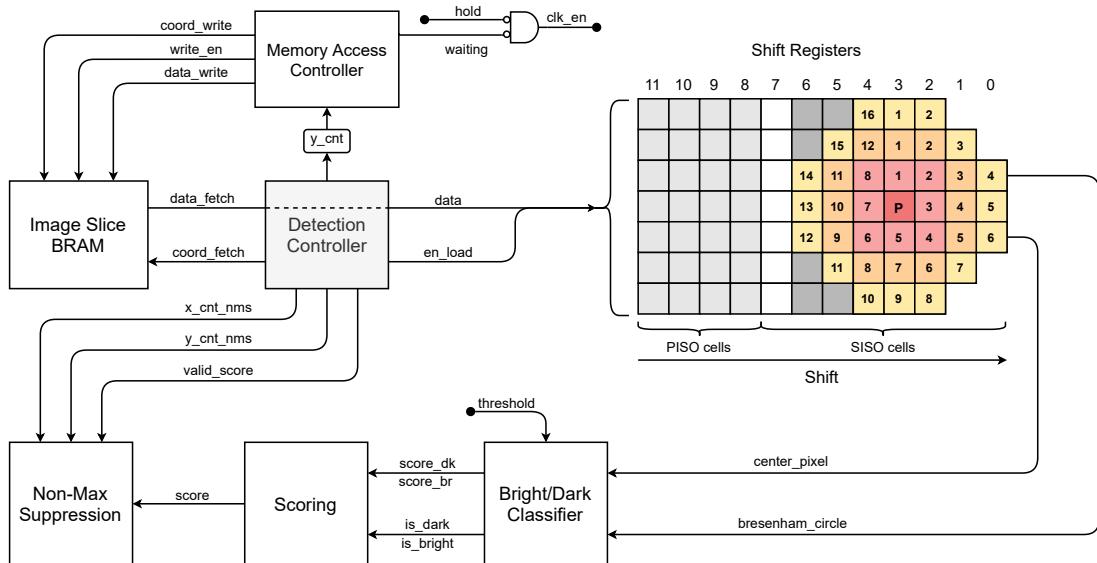


Figure 3.3.6: Buffer Controller Overview

The controller is also prepared to react to a global hold signal, to allow further stages to stall it in case of delays in processing.

In order to guarantee the most efficient use of information possible, avoiding unnecessary copies and simplifying the hardware design, the pixels of interest to the Bright/Dark classifier in the 7x7 frame will be directly mapped to the registers that are rightmost in a set of 7 shift registers. Most cells in those shift registers are Serial In Serial Out (SISO), but the leftmost cells are in a Parallel In Serial Out (PISO) configuration, as shown in fig. 3.3.6. This is to allow the information to be inserted in parallel but shifted right for then to be directed to the area where the Bresenham Circle is located. The 3 squares out of the Circle in the top and bottom right (6 in total) are not stored in order to optimise space utilisation as they never contain useful information.

In order to simplify mapping of the shift registers, always keeping the order in which the circles enter the bank, the scan is done from right to left.

In order to have a controller design that is as simple and predictable as possible, the register files of all the FAST variants will have the same number of columns.

Fig. 3.3.7 shows a first look at the intended behaviour of the Controller in *tandem* with the Bright/Dark Classifier's main logic, which acts as a consumer. It implies that the Controller is always favoured in arbitration when accessing RAM (in case of single-channel RAM blocks), to ensure the best timing with logic that is as simple as possible. It privileges filling the buffer to ensure the least possible latency, then inserting four bytes for each 4 cycles to ensure 1 byte per cycle throughput. This approach holds an essential disadvantage, which is that the insertions to the buffer need to be multiplexed, since they can be done in 3 different places for each bit. That also means that all the registers must be Parallel In Serial Out (PISO) ones. Since PISO cells require more logic, that could be a good place to start optimising. This is done in groups of 4 bytes to strike the best possible balance between hardware utilisation and dead time in reading.

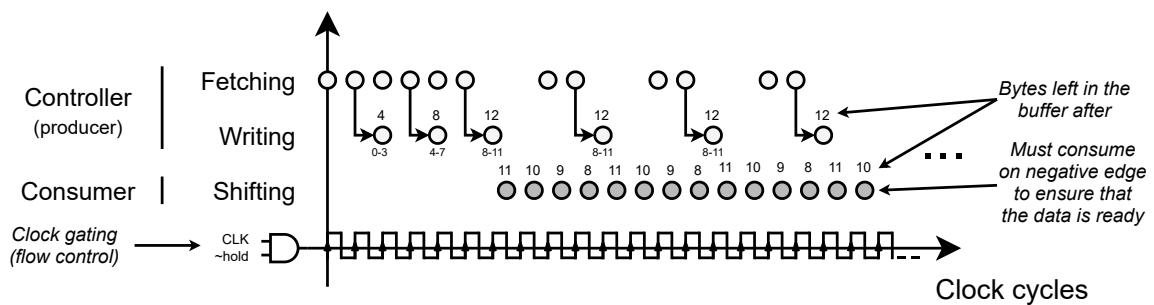


Figure 3.3.7: Simplified view of the data flow between the Controller and the Bright/Dark Classifier - 12 columns and 6 cycles of delay

An alternative approach, and the one chosen for implementation, is one that makes sacrifices the lowest latency possible in the already presented solution for improvements in hardware resources utilisation. By always inserting in the same cells and waiting for the information to be shifted before inserting the next bytes, the design can use PISO cells only in the leftmost 4 columns. The total latency at this stage is 10 cycles.

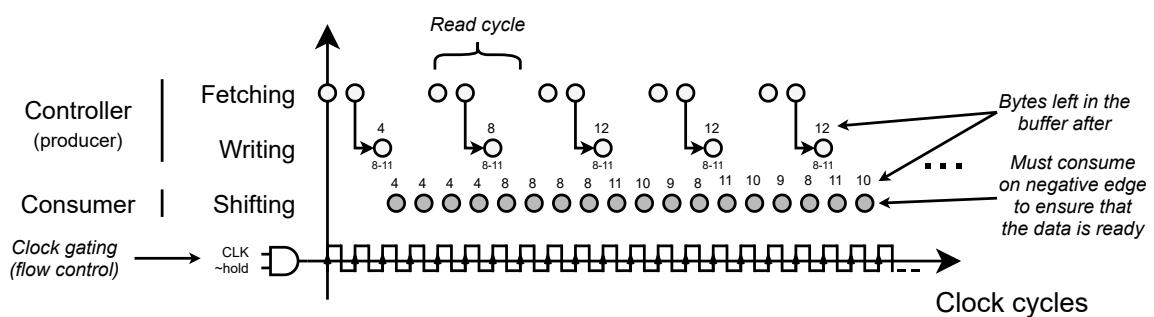


Figure 3.3.8: Simplified view of the data flow between the Controller and the Bright/Dark Classifier - 12 columns and 10 cycles of delay

The state machine for the controller is presented in fig. 3.3.9. It is based on a 4-clock edge cycle, where each edge in the cycle represents a different state. This was hinted at before in fig. 3.3.8.

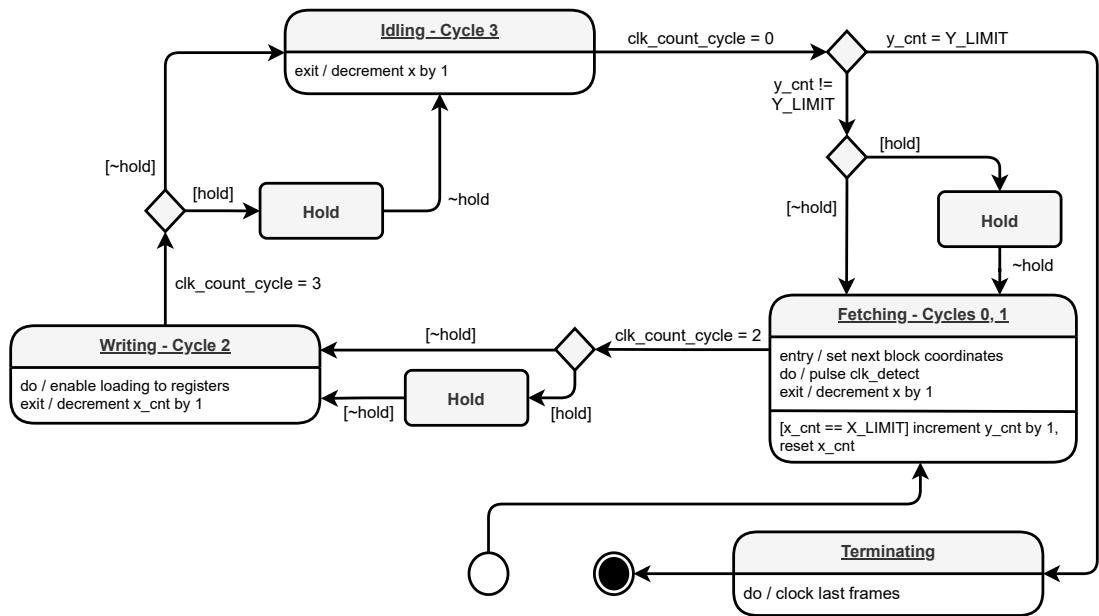


Figure 3.3.9: Controller State Machine

3.3.4 Bright/Dark Classifier

The bright/dark classifier block holds two sections as the name indicates, one for the bright comparisons and another one for the dark comparisons (figure 3.3.10). In both cases, the following sequence of steps is taken:

1. For the bright classifier, the intensity value of the centre pixel (P) is subtracted from the intensity value of each pixel on the Bresenham Circle. For the dark classifier, it is the value of each pixel in the Bresenham circle that is subtracted from the centre pixel instead;

$$OP1_{bright} = I_{p \rightarrow x} - I_p \quad (3.4)$$

$$OP1_{dark} = I_p - I_{p \rightarrow x} \quad (3.5)$$

2. The threshold value is subtracted from each of the n results from the first operation;

$$OP_2 = OP_1 - t \quad (3.6)$$

3. If OP_2 is greater than zero, the corresponding bit in the array (depending if it is the bright/dark section) will be set to one.

The scoring phase takes as input the result of the second operation if the third operation was true and or zero in the case where the third operation turned out to be false, because the result of the of the third operation is connected to the selector input. This requires n multiplexers and applies for both sections (bright and dark).

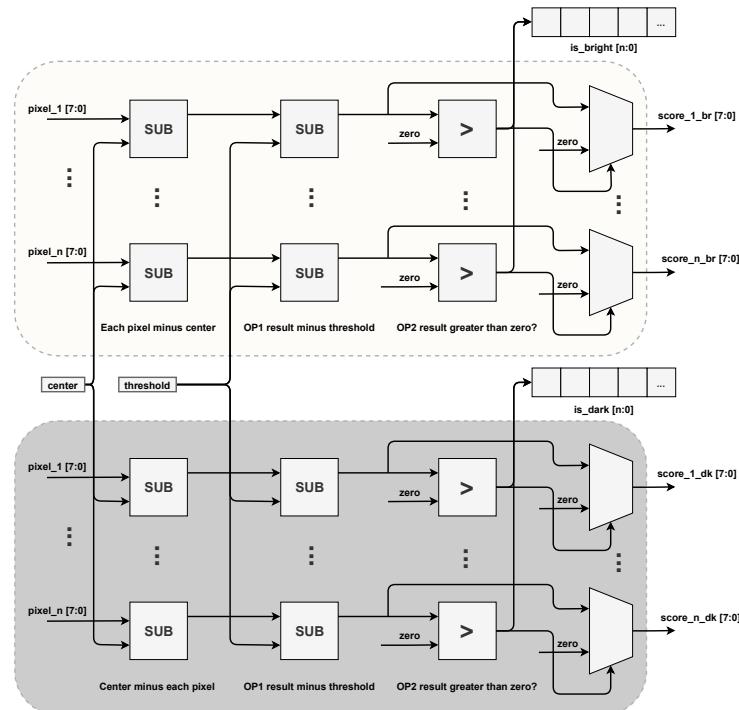


Figure 3.3.10: Hardware Architecture: Bright/Dark Classifier

3.3.5 Contiguity Test

The implementation of FAST point detection algorithm requires several comparisons for detecting contiguous and sufficiently brighter and dark pixels. Since the type of FAST(FAST-5, FAST-9, and FAST-12) dictates the number of pixels to be contiguous for the candidate corner to be a keypoint. It is extracted all possible states for 5,9 or 12 contiguous pixels within the input received by the Bright/Dark Classifier that is shown in the figure below accordingly with the type of FAST chosen previously.

This figure shows the possible states for FAST-5,FAST-9 and FAST-12 contiguous pixels in which the circled pixels denote sufficiently brighter or darker pixels and non circled pixels indicate ‘don’t-care’ pixels.

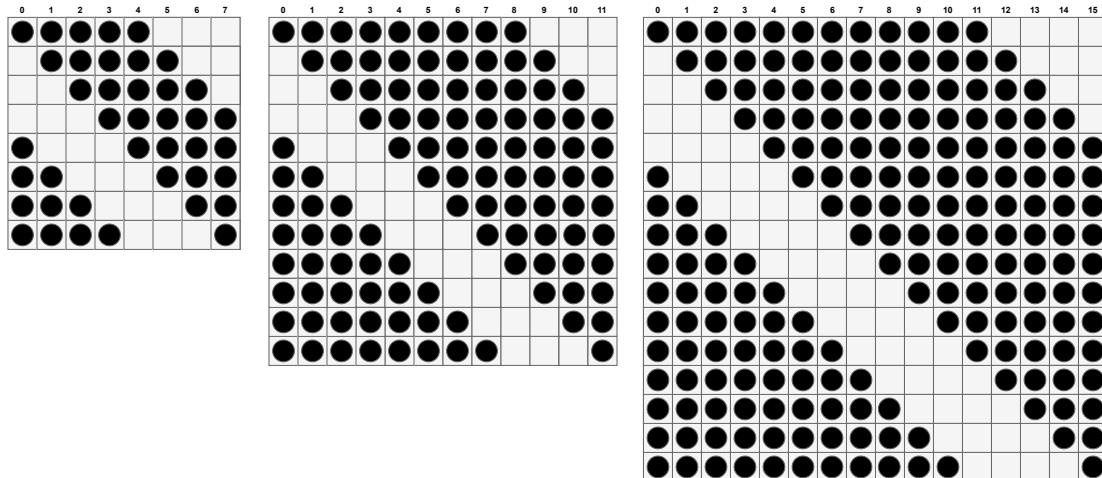


Figure 3.3.11: 16 Possible States for FAST-5, FAST-9 and FAST-12

The next figure illustrates the proposed parallel hardware architecture to extract FAST points by detecting one of 8, 12 or 16 possible states accordingly with the type of FAST-N. The architecture employs two banks compactors to compare the intensities of the pixels with $IP + T$ and $IP - T$ values. Then, the outputs of comparators are searched for one of the possible states of contiguous pixels using two combinational circuits and a logical OR to detect FAST points.

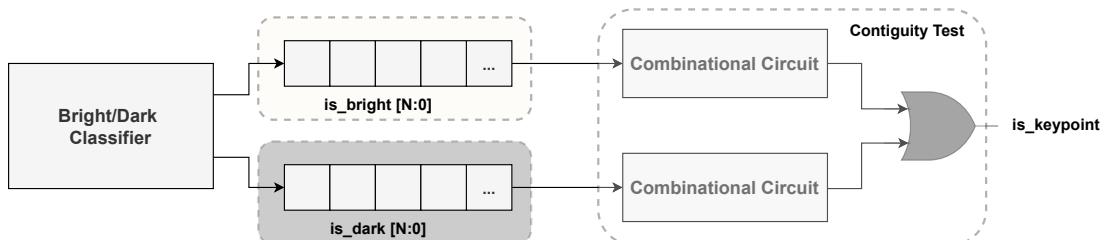


Figure 3.3.12: Parallel Hardware Architecture For Contiguity Test Where N Can Be 7,11 or 15

The first step in performing score computation is to split the two sets of 64,96 or 128bits received from the bright/dark classifier with the pixels classified as darker or lighter. With this division we get 8, 12 or 16 arrays of 8-bit each of the classification of how darker the pixels are and 8, 12 or 16 arrays of 8-bit each of how lighter pixels are.

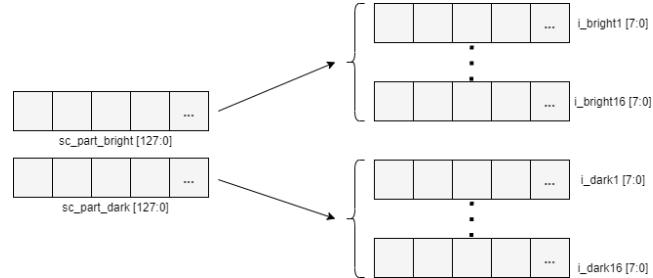


Figure 3.3.13: Deconcatenation of the Outputs from Bright/Dark Classifier in case its FAST-12

This outputs from the Bright/Dark classifier will be used in following equation to obtain the score the keypoint:

$$V = \max \left(\sum_{x \in S_{bright}} \left(|I_{p \rightarrow x} - I_p| - t \right), \sum_{x \in S_{dark}} \left(|I_p - I_{p \rightarrow x}| - t \right) \right) \quad (3.7)$$

$$S_{bright} = \{x \mid I_{p \rightarrow x} \geq I_p + t\} \quad (3.8)$$

$$S_{dark} = \{x \mid I_{p \rightarrow x} \leq I_p - t\} \quad (3.9)$$

The score computation will consists of two adder trees. Each adder tree will have the inputs for the corner score components from either the "bright" or the "dark" group of pixels. The adders will produce two values, the sum of the components of both of the aforementioned groups. Below is a representation of the adder tree for the "dark" group of pixels. In the example with the 16 sets of 8 bits each, the first sum will produce 8 results with 9 bits each. Then the second sum will produce 4 sets of 10 bits each. The third sum will have 2 sets of 11 bits each, and the final result will be a 12-bit value.

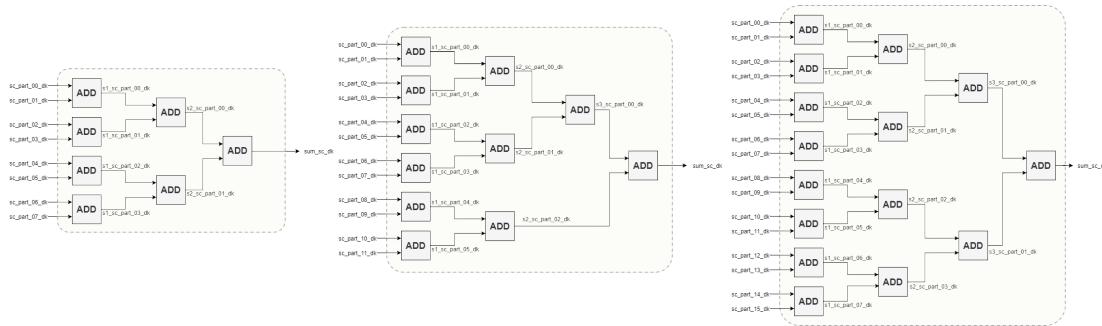


Figure 3.3.14: Score Computation for the Dark Group

After completing both sums in parallel the greater of those values is passed as the final value of the corner function as per equation 3.1.

Both dark and bright sums will go into the MUX and the result of the comparison will go in as well, to be able to select the maximum value between the dark and bright sums.

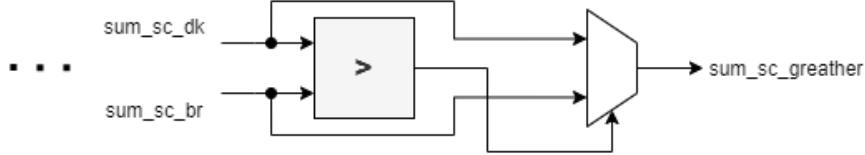


Figure 3.3.15: Comparison Between the Scores

The output, score, will have a variable size, depending on N. In the case of FAST-5, three chains of sums are performed, i.e., sum of 8-bit arrays that result in 9bits, which will be summed also with 9 bits and result in 10bits and, finally, the sum of 10bits with other 10, will result in 11bits, so the score array will be 11bits. For FAST-9 and 12, the score array will have 12 bits.

The value obtained in this phase will be passed as input to the next non-maximum suppression phase, so that the best points are selected and the least classified are discarded.

3.3.6 Non-maximum Suppression

High Level Synthesis is an automated design process that interprets an algorithmic description of a desired behaviour and creates digital hardware that implements that behaviour. Following with that idea, one designed some flowcharts to specify our module behaviour.

Once the module has all data at its disposal, it immediately starts by applying the horizontal sparse point algorithm. Then, the data structures (as the arrays containing pixel's positions and scores), are reorganised in order to bunch the information in such way we can initiate the next stage. When vertical NMS has finished, one completed the 2D filtering process, remaining only the comparison among layers. The fig. 3.3.16 represents the NMS filter.

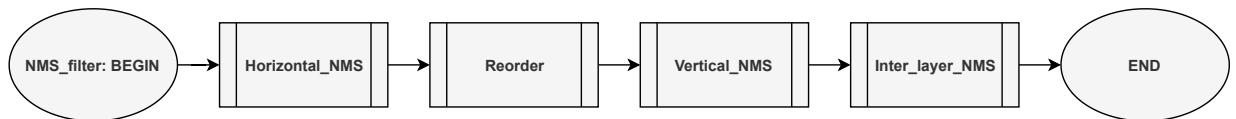


Figure 3.3.16: NMS Filter

2D Non-maximum suppression. Here in fig. 3.3.17, one presents the combinational logic circuit that resumes the horizontal and vertical stages. It consists of a tree with two branches, one for each comparison (with right and left pixels). Each branch starts with a comparator followed by a mux to select the great score and, in parallel, subtracts/adds a pixel to the input coordinate. The next clock cycle operates a logic and between those two outputs. Finally, we intercept the two verifications in order to generate the final stage output.

Data Handling. The basic filter's process has been described, but the overall data handling isn't that simple. It must be a background mechanism responsible for thinking about every possible situation when the pixels first start to come. For example, if some line has only one corner, the algorithm has to be capable of identifying it as such and put

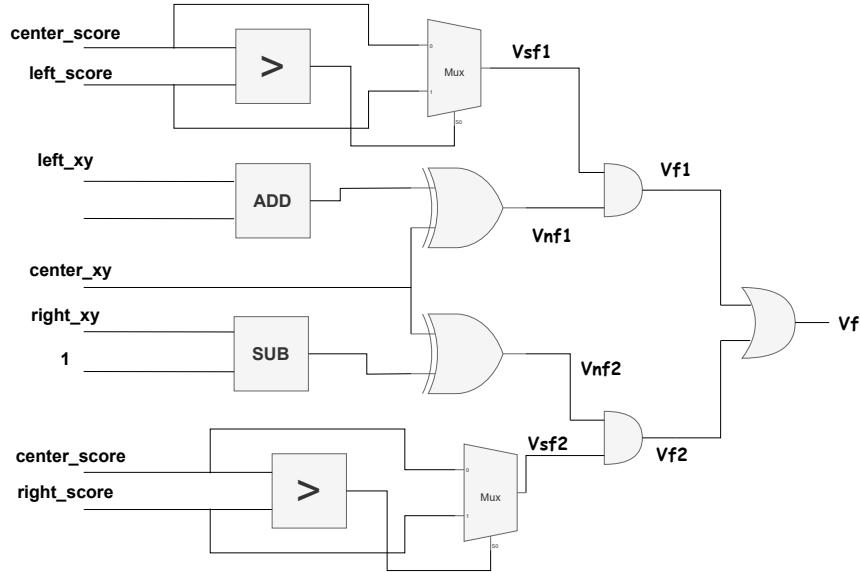


Figure 3.3.17: 2D Processing Logic

it in the correct position, which implies skipping a few steps. Only then, when everything is properly situated on the vectorised picture conception, the filter can be applied.

Horizontal stage. The next figure reflects the designed state machine to implement taking into account the following situations: the number of corners per line (one, two or three or more); and the corner's position (whether it is the first corner of the line, it is in the middle or the end). For each of these conditions, the system must respond accordingly.

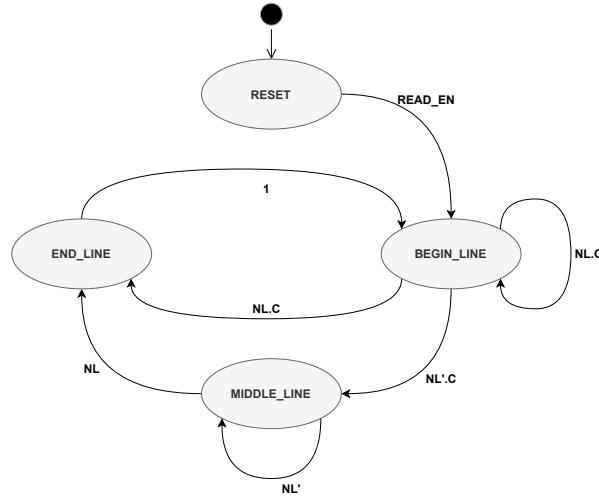


Figure 3.3.18: Horizontal NMS - State machine.

When the module starts receiving pixels, it automatically places itself on *begin_line* state. Unless there is only one corner in that line, the next iteration takes the system to the *middle_line* state and remains there until a pixel

from a new line arrives. Once that happens, the line's closed in the *end_line* state and returns right after to the *begin_line*.

These states are going to be considered in the filter code. When the state corresponds to the *begin_line*, the system discards the right pixel of the comparison because it has no valid information. The *end_line* does the opposite discarding the left pixel, and in the middle of a line, the system considers all three registers' information.

Stage Transition. The transition between horizontal and vertical stages is quite critical, and it demands detailed attention. The projects' point is to implement a hardware accelerator, but as wise engineers, we cannot implement that at any cost. This application requires resource concerns to follow the speed ones, which means that we must find a solution to execute the transition between these two states spending the minimum possible amount of memory.

That's why the system does not wait until the horizontal stage to finish to start the vertical one. That possibility would imply saving the whole picture corners, and it could be catastrophic from the hardware management standpoint. The solution is to reserve memory for only three pixels per column and applying the vertical filter as the new pixels are coming.

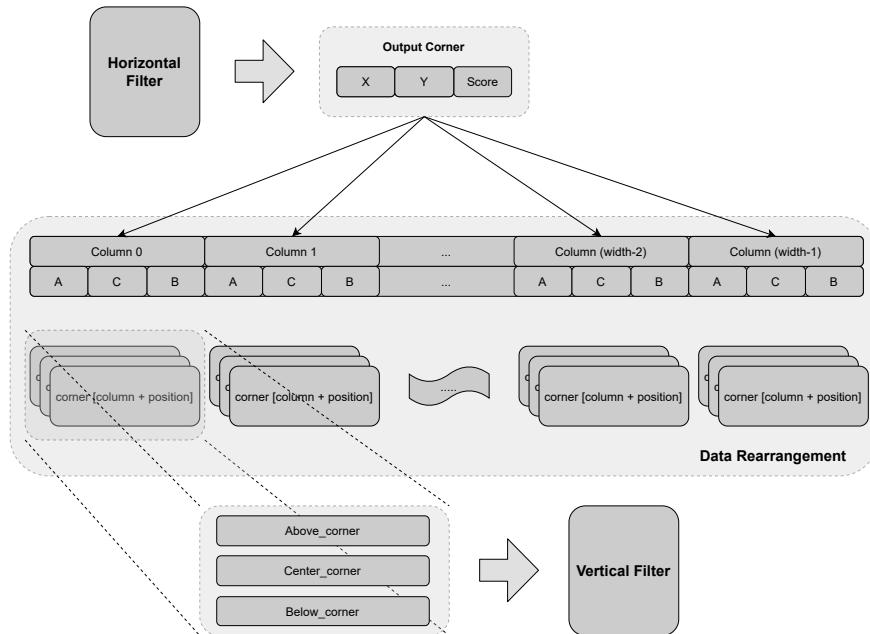


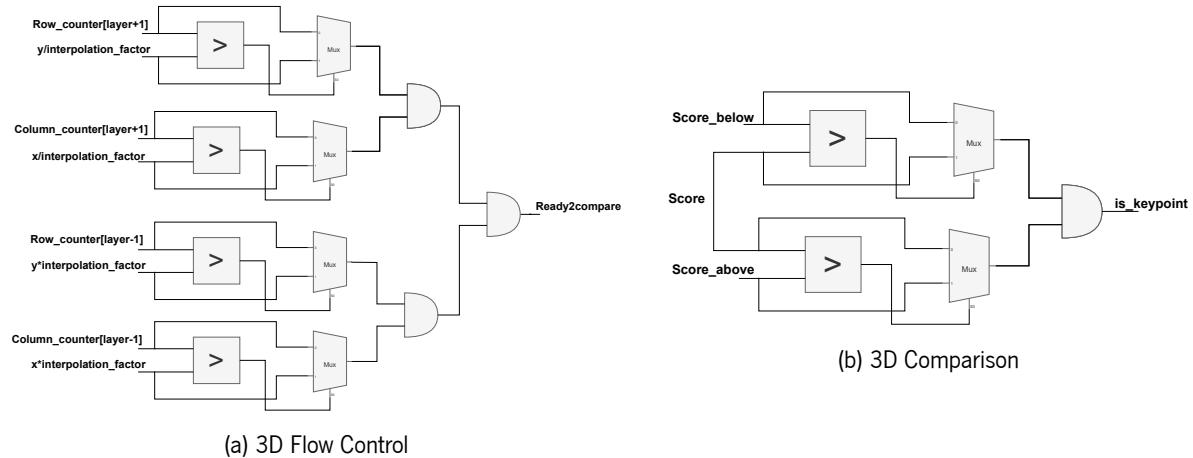
Figure 3.3.19: Data rearrangement.

So, every time there is an outcome from the horizontal filter, the output corner is placed in the respective column and position. For example, the corner whose X coordinate is '000000101' must be placed in the first position of the fifth column. Then, when the next corner of that column arrives, the second position takes the first one's value, and the system puts the new pixel on the top. At the same time, the scores and coordinates are compared to find whether the centre's corner is a keypoint or not, and we through the final output.

3D Non-maximum suppression. To describe the first part, one starts by getting the coordinates from the above and below layers. Besides, we need those layers row and column counters as entries of the circuit. It indicates if

those layers' pixels are already available for comparison, and we know that if the values of the counters are great than the actual coordinate. This process requires four comparators and four muxes, two for the coordinates of each layer. The following operations only assemble that outputs in a single control code. This NMS phase can be observed in fig. 3.3.20a.

The next step is simple and consists of comparing the corner score with the other layers' corresponding pixels score. After the muxes select the greater values, we join those two verification's in the final variable, which tells if a corner is still a corner or not. The fig. 3.3.20b shows this process in more detail.



3.3.7 Software Refactoring

State Chart. In the figure 3.3.21, is presented the State Chart of the BRISK algorithm. Since the implementation will be in C++, was created a object oriented line of thought.

Initially, the constructor will receive the image, the threshold value and the fast value. The function "*compute*" will compute all the pixels of the image and it will be there where all the processing of keypoints will be done. In the end, this function will return a list of keypoints that will be sent to the descriptor.

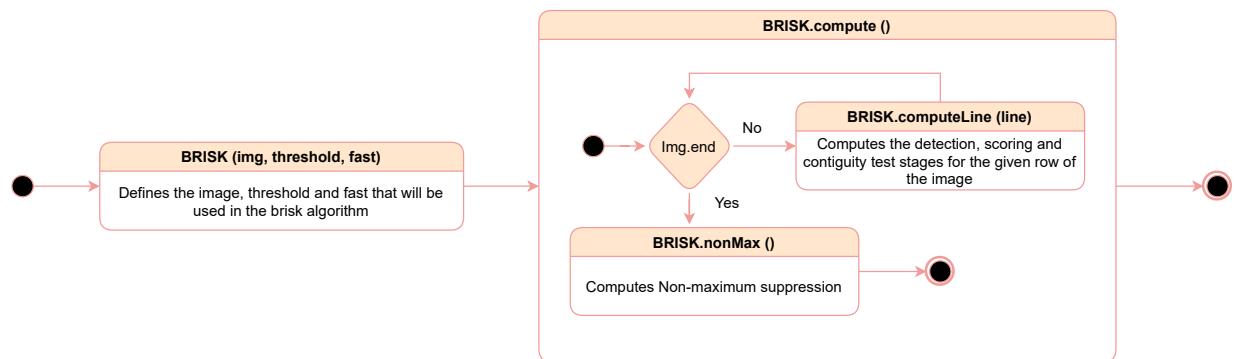


Figure 3.3.21: State Chart Diagram

Class Diagram. The figure 3.3.22 shows the class diagram where is presented all the parameters of the BRISK class.

It will be implemented two structs, one for the points which contains only the position of a pixel, and other for the potential keypoints, that contains a point as well as the score of the potential keypoint, and if it is suppressed or not in the NMS stage.

It will be used a linked list for the potential keypoints in order to have the position and the score of the detected potential keypoints and a linked list of lists, where every position of the list contains a list of potential keypoints.

Besides that there is an enumeration with the value of the FAST that can only be 5, 9 or 12.

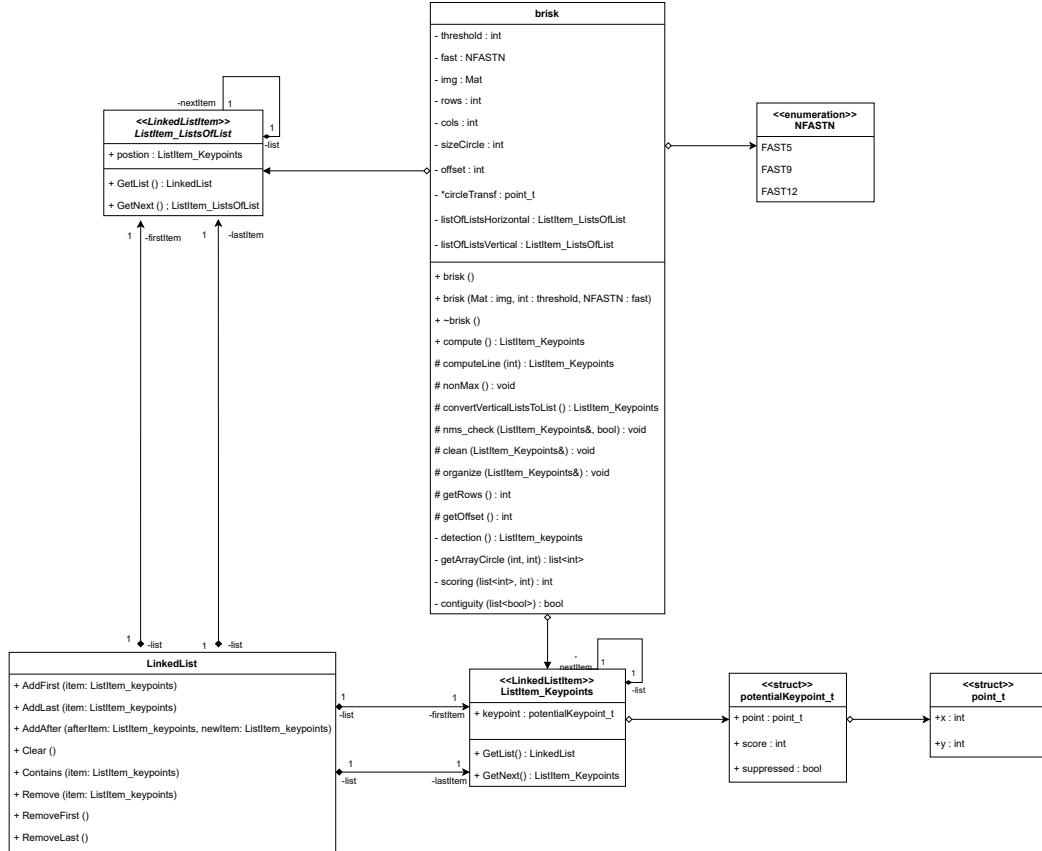


Figure 3.3.22: Class Diagram

3.4 Description

This section will concern the BRIEF description phase.

3.4.1 Memory Access

BRIEF's inputs are the keypoints' locations incoming from the detection module and the image under processing. This image is previously converted to greyscale so a pixel can be represented by only one byte and smoothed with a Gaussian filter to reduce eventual noise.

For each keypoint location's grabbed, a patch with a variable size of NxN pixels around the keypoint is created by acquiring the adjacent pixels from the image. Having this patch, a binary string describing the keypoint that regards the input's keypoint location is issued after running the BRIEF description algorithm.



Figure 3.4.1: Keypoint location retrieval and binary descriptor string storage flowchart

Patch Creation Module. In order to create a patch around a determined keypoint with location (x, y) , the address for the keypoint in the memory, where the image is stored, has to be calculated. Having the address of the keypoint, the address of the first pixel for each row of the patch is obtained and then each patch line is extracted from the image memory and stored in the register file.

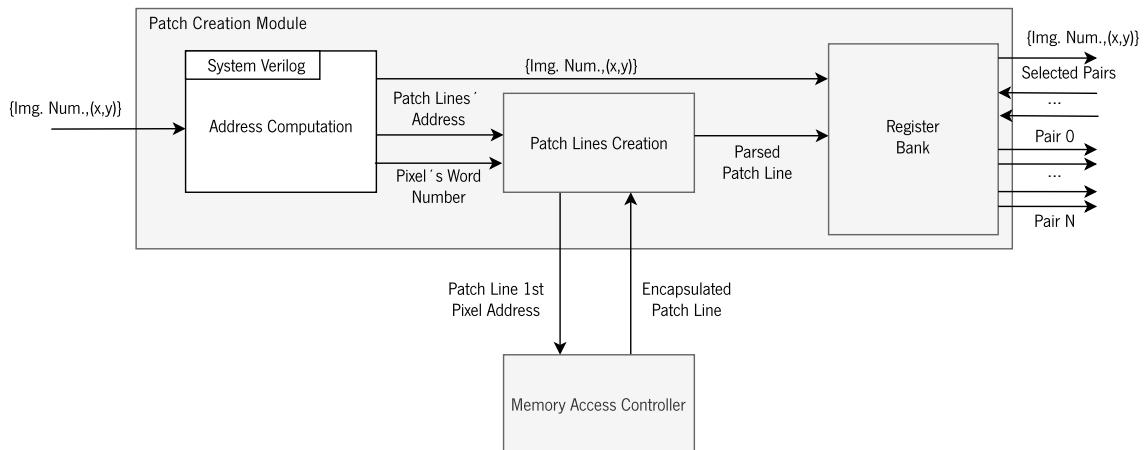


Figure 3.4.2: Patch creation module diagram

For a better understanding, supposing the patch has a size of 19×19 pixels, the address of the first pixel for each one of the 19 rows is calculated and from it are obtained the 19 pixels regarding that patch line. Each patch line is then stored in one of two 19×19 register files.

Two register files are used so that while one register file is being used for the description, the other one is being filled with the next patch for the next keypoint. Considering the description algorithm takes longer than the process of

making one patch, this method allows for immediate patch switching, which means the description module has always a patch ready for description of the corresponding keypoint.

Additional registers are used to store the corresponding keypoint coordinates and image number so that each patch has its information at all times. This greatly reduces synchronisation efforts.

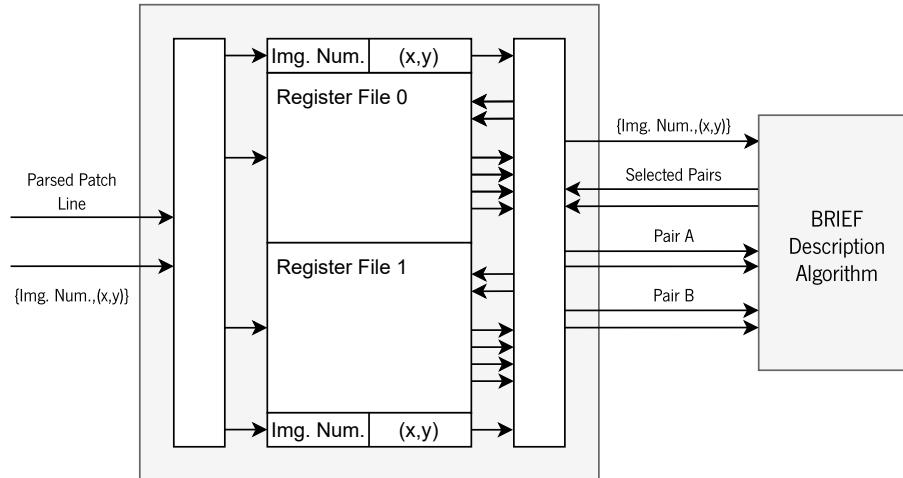


Figure 3.4.3: Register File diagram

Compute Patch Pixels Address Module. Given a keypoint location incoming from the detection stage, the address of the keypoint in the memory reserved to the image can be calculated using the equation 3.10 where x and y are the pixels' coordinates in the image and W is the image's width.

$$address = (x - 1) \left(\frac{W}{4} \right) + \frac{y - 1}{4} \quad (3.10)$$

For a better understanding, the representation of the fig. 3.4.4 simulating an image with a width of 20 pixels and a height of 10 pixels is used to visualise how the pixels are stored in the memory. For the purpose, it is considered that the image is stored in a way that 4 pixels are stored for each memory position.

1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8	1,9	1,10	1,11	1,12	1,13	1,14	1,15	1,16	1,17	1,18	1,19	1,20
2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8	2,9	2,10	2,11	2,12	2,13	2,14	2,15	2,16	2,17	2,18	2,19	2,20
3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8	3,9	3,10	3,11	3,12	3,13	3,14	3,15	3,16	3,17	3,18	3,19	3,20
4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8	4,9	4,10	4,11	4,12	4,13	4,14	4,15	4,16	4,17	4,18	4,19	4,20
5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8	5,9	5,10	5,11	5,12	5,13	5,14	5,15	5,16	5,17	5,18	5,19	5,20
6,1	6,2	6,3	6,4	6,5	6,6	6,7	6,8	6,9	6,10	6,11	6,12	6,13	6,14	6,15	6,16	6,17	6,18	6,19	6,20
7,1	7,2	7,3	7,4	7,5	7,6	7,7	7,8	7,9	7,10	7,11	7,12	7,13	7,14	7,15	7,16	7,17	7,18	7,19	7,20
8,1	8,2	8,3	8,4	8,5	8,6	8,7	8,8	8,9	8,10	8,11	8,12	8,13	8,14	8,15	8,16	8,17	8,18	8,19	8,20
9,1	9,2	9,3	9,4	9,5	9,6	9,7	9,8	9,9	9,10	9,11	9,12	9,13	9,14	9,15	9,16	9,17	9,18	9,19	9,20
10,1	10,2	10,3	10,4	10,5	10,6	10,7	10,8	10,9	10,10	10,11	10,12	10,13	10,14	10,15	10,16	10,17	10,18	10,19	10,20

Figure 3.4.4: Example image representation

Put this, the address of the keypoint with the location (6,10) is:

$$\text{keypoint add} = (6 - 1) \left(\frac{20}{4} \right) + \frac{10 - 1}{4} = 27.25$$

Where the decimal component reveals that the keypoint is the 2nd block of the 27th memory position. A null decimal component implies the pixel is in the 1st block of the memory address with the value regarding the integer component, while a decimal component of 50 implies the pixel is located in the 2nd block and a decimal component of 75 implies the pixel is located in the 3rd block.

Having the keypoint address, it is possible to get the address of the pixel regarding the beginning of the patch line it is inserted. This is effortlessly obtained using the equation 3.11, where N regards the size of the description patch.

$$\text{keypoint patch line add} = \text{keypoint add} - \frac{N - 1}{2} \left(\frac{1}{4} \right) \quad (3.11)$$

Taking advantage of the previous example, the keypoint patch line address for the keypoint with the location (6,10) is:

$$\text{keypoint patch line add} = 27.25 - \frac{5 - 1}{2} \left(\frac{1}{4} \right) = 26.75$$

Put this, the addresses of the first pixel of every patch lines are computed using the equation 3.12, where i goes from 0 to $N-1$ and W is the image's width.

$$\text{patch line add} = \text{keypoint patch line add} - \left[\left(\frac{W}{4} \right) \left(\frac{N - 1}{2} - i \right) \right] \quad (3.12)$$

For demonstration, the address of the first pixel of the fourth line of the patch can be obtained:

$$\text{patch 4th line add} = 26.75 - \left[\left(\frac{20}{4} \right) \left(\frac{5 - 1}{2} - 3 \right) \right] = 31.75$$

As the addresses of the first pixels of every patch line are computed, they are outputted to the memory multiplied by 100, so the decimal part is suppressed but the information is kept. This way, only this module has to be written in System Verilog to allow the use of real variables.

3.4.2 Pair Generation

After the keypoint detection stage, a certain number of keypoints are passed to the BRIEF descriptor and the image features are created. In this stage a patch is applied to each keypoint (as presented above) and multiple pairs of points are created inside that x by x pixels area, representing an individual fingerprint of each keypoints. This information is responsible for the feature description and is stored in the form of a binary string, with a size dependent on the selected number of generated pairs.

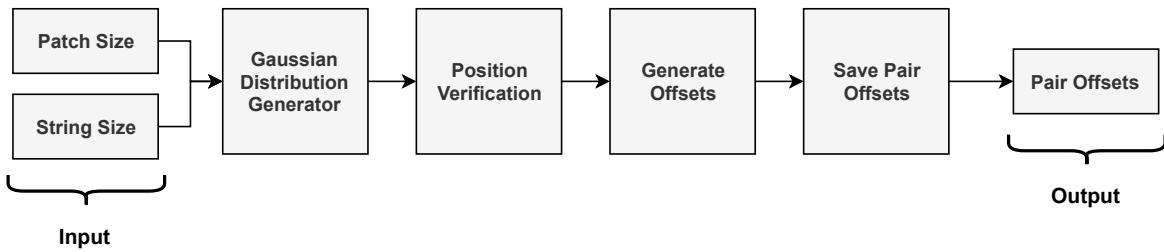


Figure 3.4.5: Pair Generation Overview

As explained in the section 2.1.3, from all the geometries, the fifth geometry is the only one that is not based on a random normal distribution design, since this method shows better results when compared with the non random symmetrical and regular strategy. For these reason we can already exclude this fifth option. Regarding the remainder, the results vary depending on the inputs, but in most cases the second algorithm shows the best results, enjoying a small advantage over the rest in terms of recognition rate. So this will be the chosen method for our sampling geometry.

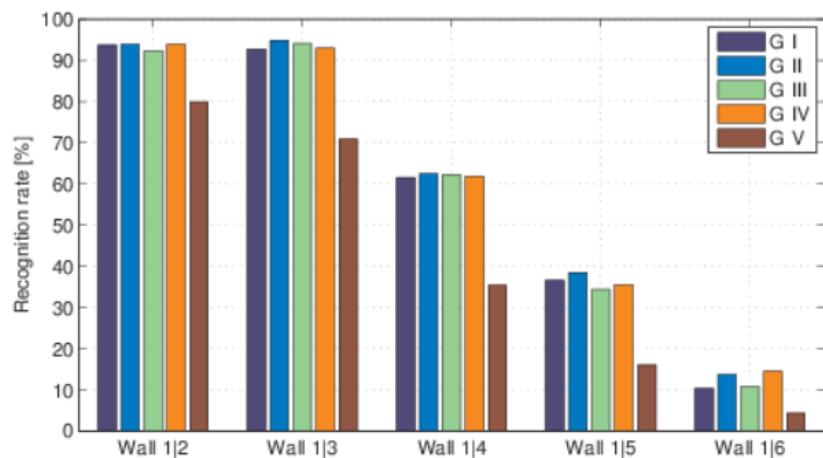


Figure 3.4.6: Recognition rate of the five different sampling geometries

As seen in the figure above, this tests were performed in order to demonstrate the different recognition rates of the five sampling geometries (on the y axis) in relation to an increasingly more difficult image to perform the match (on the x axis). In the chosen geometry both x and y pixels in the random pair are drawn from a Gaussian distribution or a spread of $0.04 * S^2$ around the keypoint.

During pair generation is fundamental to verify if all the coordinates of the detected pairs are contained inside the patch. This verification is important since it ensures that a certain feature belongs only to its respective keypoint. In the figure below, it's possible to analyse the different stages of the pair generation.

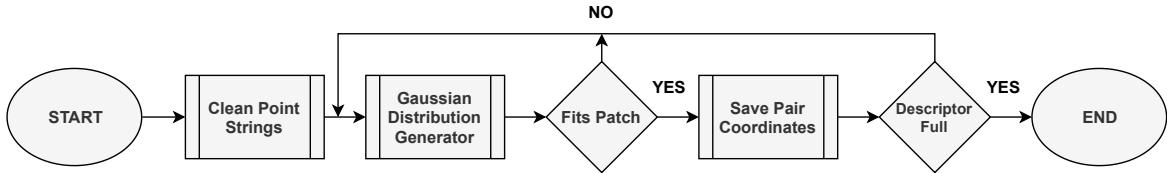


Figure 3.4.7: Pair generation function flowchart

So summing up, the pair generation starts cleaning the strings used to store the different pair's coordinates values. Then a Gaussian distribution is applied, in order to spread in a distributed manner all the different pairs inside the patch. Afterwards is verified if the points are actually inserted between the borders of the patch, and in case of success, the offsets to a central point are calculated and saved in two vectors. Otherwise, the pairs are discarded and new points are sampled. This process is repeated a number of times, depending on the BRIEF type, creating multiple pairs that will be used in the matching stage.

Implementation of pairs generation. It was decided to generate all pairs in MATLAB, using a sampling generator function that receives as input parameters, the type of generation (type of distribution of points), the size of the patch window, and the number of pairs to be generated, and returns two vectors with the offsets in relation to the central point of the patch. These vectors will be stored in non-volatile memory so that they can be used later.

3.4.3 Binary Test

Since the image received by the BRIEF module already comes in grayscale format, each pixel only uses 8 bits, whose values of intensity range from 0 to 255, with the highest value of intensity corresponding to the brightest pixel.

As the target descriptor length is n , 4 parallel comparators are used to compute the complete binary vector in $n/4$ clock cycles. The resulting binary vector is stored in a n -bit register as shown in figure fig. 3.4.8.

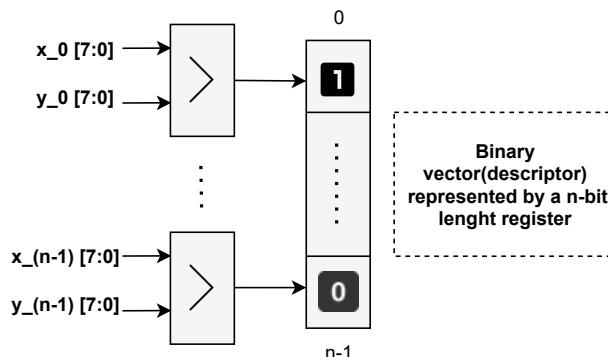


Figure 3.4.8: Binary Vector Computation Module Diagram

Algorithm Overview. The flowchart below represents the algorithm that executes on every positive edge of the clock. The first step is to check the *patch_available* flag, which indicates that an image patch is available in the register file. If this condition is true, the memory addresses for four patch-pairs will be acquired and used as an input to the register file.

Once the memory addresses for the first four patch-pairs have been computed, the *begin_desc* flag is activated and the description process can start by comparing four patch-pairs in each clock cycle. The description itself will be one clock cycle behind the computation of patch-pairs, because it is not possible to do both in the same clock cycle, and the *begin_desc* flag works to this effect.

Every time the *desc_complete* flag is active means that the descriptor string is finished and the indexes for both the pair addresses index and the descriptor itself can be reset.

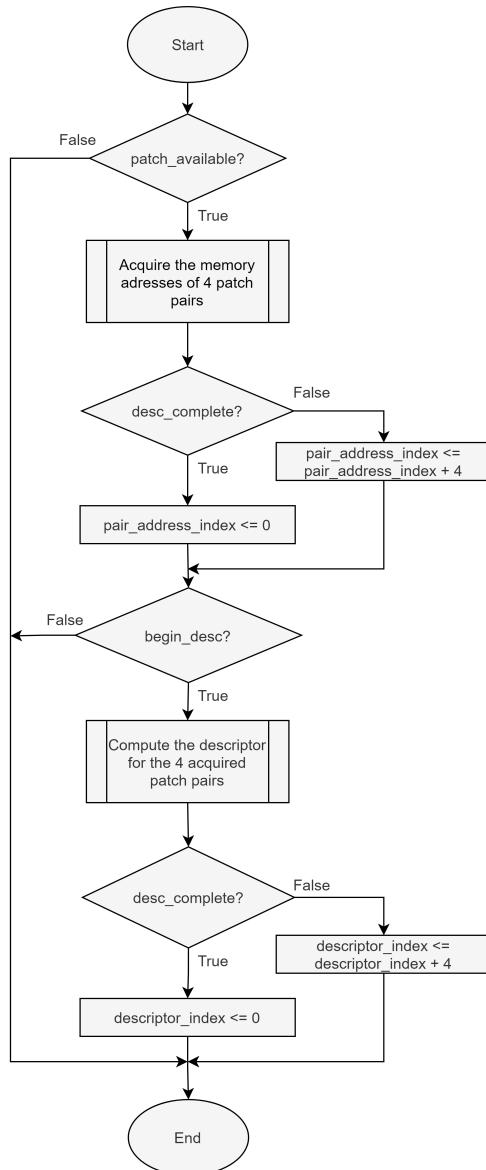


Figure 3.4.9: Flowchart for the binary test

3.4.4 Software Refactoring

The purpose of a software refactoring is not necessarily to improve the internal structure of an existing program's source code, but to make more understandable and perform better given a specific situation. Taking this into account the source code which we will base on, will be the OpenCV, more speciously:

Listing 3.4: create() Method

```
static Ptr<BriefDescriptorExtractor> cv::xfeatures2d::BriefDescriptorExtractor::create(int bytes = 32, bool use_orientation = false
)
```

Where, bytes is the length of the descriptor in bytes, valid values are: 16, 32 (default) or 64 and use_orientation is the usage of sample patterns using keypoints orientation, which is disabled by default.

- It is important to notice that BRIEF is by norm, a rotation variant descriptor, which is why after the refactoring the created function will only allow the choice of the descriptor size;

Listing 3.5: compute() Method

```
virtual void cv::Feature2D::compute(InputArray image, std::vector<KeyPoint> & keypoints, OutputArray descriptors)
```

As the name suggests this function computes the descriptors for a set of keypoints detected in an image and places that said descriptor in an array.

- After the refactoring, as previously stated the user can chose the descriptor size but also the patch size, this choice will make the external behaviour change for this specific project, complying with the previously defined requirements;

So with the use of the software refactoring, the benefits that are expected, when comparing with other source code are:

- making the code more understandable;
- favouring the reusable design elements (e.g. the design patterns) and code modules;
- improving the objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance;
- allowing the developer to have a better understanding of the decisions made, in particular in the context of this application;

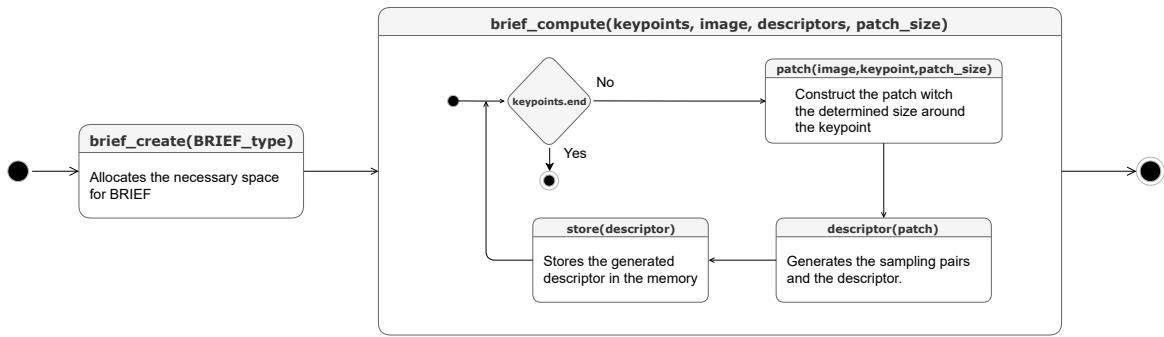


Figure 3.4.10: Software Refactoring State Machine

State Chart. As previously stated in chapter 2.1.3 BRIEF follows a simple line of tough, exemplified in fig. 2.10.2 (where BRIEF will first create a patch, followed by the pattern generation and finally the computation of the descriptor).

In fig. 3.4.10 similarly to fig. 2.10.6 and to OpenCV approach, the function `brief_create` the selected space will be correctly allocated, depending on the user option. The function `brief_compute` can also be seen in greater detail in fig. 3.4.10, where the first step is verify if all the keypoints where processed (these will be received as a linked list or a vector), if not the descriptor will be computed and stored following the previously presented line of tough. Otherwise if all keypoints where processed then the `brief_compute` function will end.

Since the implementation will be done in C++, a more object oriented line of thought was considered in which the following changes were made to the state chart, demonstrated in figure 3.4.11. The constructor will then receive the patch and descriptor size, the image and where to store the descriptor, while the compute function only receives the list of keypoints and generates and stores the descriptor for each one.

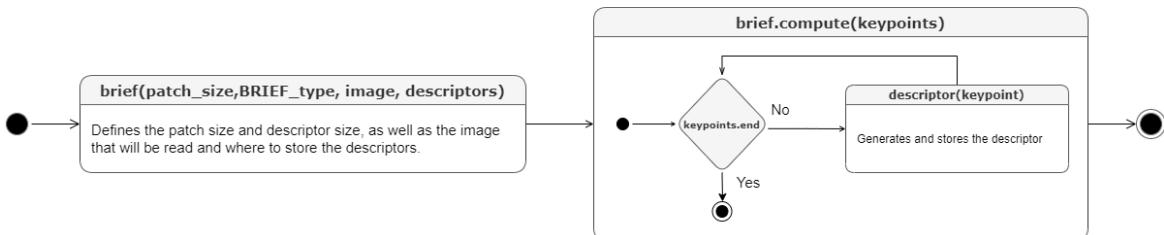


Figure 3.4.11: Software Refactoring State Machine

Class Diagram. The software refactoring will be implemented using C/C++ programming language. Taking this into account the class diagram presented in fig. 3.4.12 shows the class construction, where both the keypoints and the respective descriptor will be implemented via a linked list.

Also in fig. 3.4.12, the linked list for the descriptor will contain not only the descriptor but also the keypoint location for that said descriptor. The enumeration designated by **BRIEF_type** has three methods, which will inform the size of the descriptor for the allocation and computation of the descriptor, where **BRIEF_8** will be a descriptor with a 64 bit, **BRIEF_16** a 128 bit descriptor and **BRIEF_32** a 256 bit descriptor.

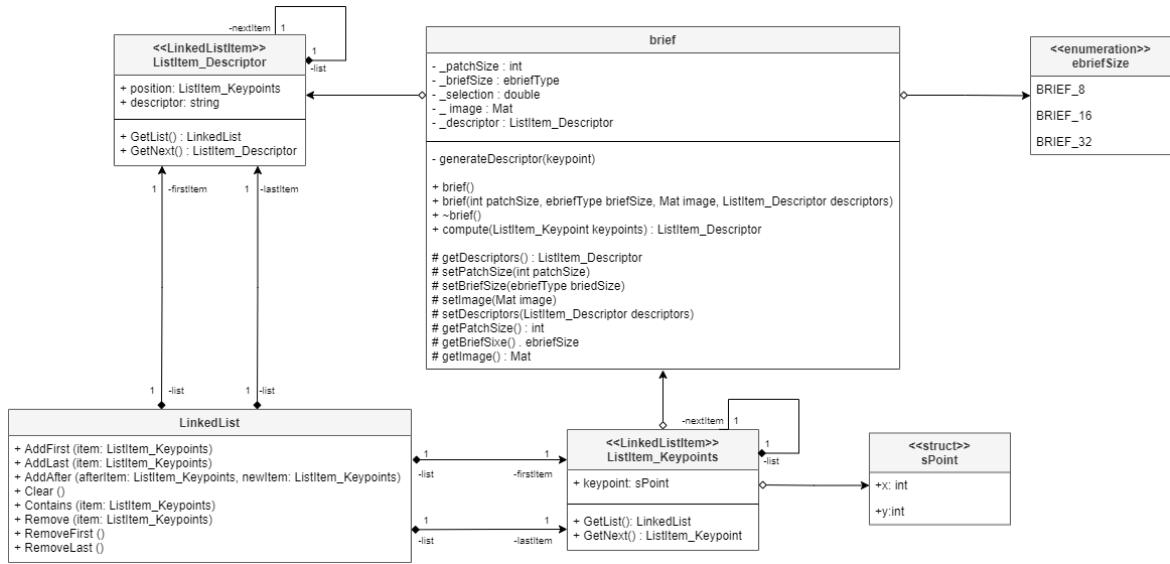


Figure 3.4.12: Software Refactoring Class Diagram

Sampling Pattern. For the implementation of the sampling pattern, the pattern was generated using already implemented MatLab script (which the result is shown in figure 3.4.13). [?] The pattern was then transferred for a .txt file allowing us to use it in the implementation. Although, there was the opportunity to generate each run the pattern utilising a pre-determined seed, it was not feasible to expect the same result in different platforms. The method chosen allows for a simple and efficient way to save and use the desired pattern.

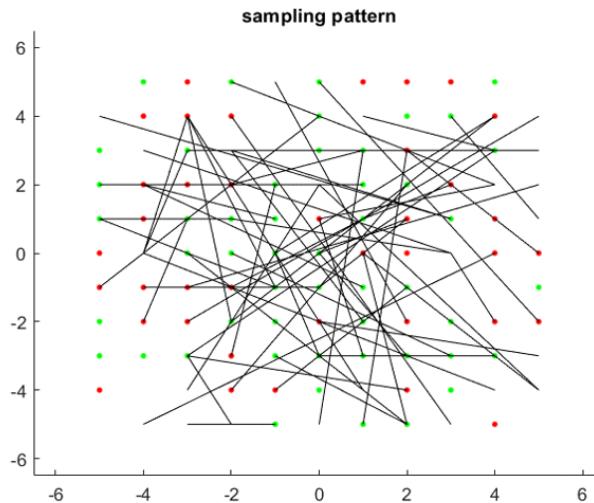


Figure 3.4.13: Sampling pattern generated by Matlab with the descriptor size as 64 and the patch size as 11

As previously stated the keypoint is the centre of the pattern, so for each pair of points (sampling pairs) there is a line, and a necessity to locate each point position relative to the keypoint. Taking this into account a Cartesian plane was defined with keypoint as its centre, shown in figure 3.4.13.

A concern raised by the implementation is the necessity of different txt files according to the size of the patch and the descriptor. Since when implementing the code there are 3 choices of size for each parameter (descriptor and the patch) there were created 9 different files where their name would be given by the following formula:

$$\text{"brief_"} + \frac{\text{DescriptorSize}}{\text{PatchSize}} + \text{".txt"} \quad (3.13)$$

Where the division of the descriptor size for the patch size is approximated to the unity.

Descriptor Generation. As previously stated BRIEF follows a relatively simple line of thought (exemplified in chapter 2.10.1). The implementation in software will differ slightly since the pattern will be previously saved as offsets from the keypoint and the totality of the image is available at any time, both these factors removed the need for the patch construction. With that mind the BRIEF implementation followed the flowchart shown in the figure 3.4.14.

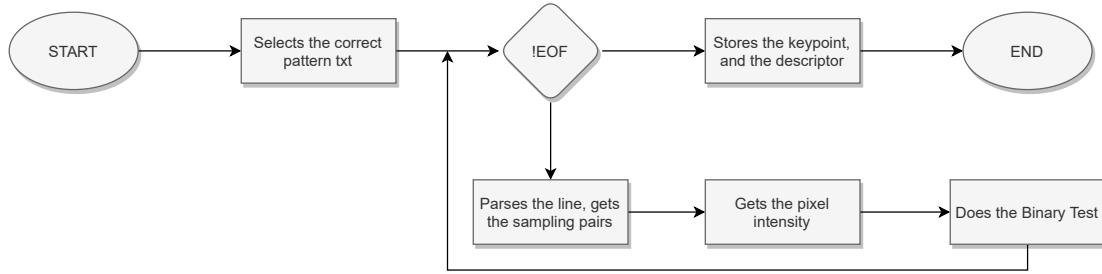


Figure 3.4.14: Flowchart for the descriptor generation in software

The generation in software will then follow the diagram presented in figure 3.4.14, where first the correct file is open according to the descriptor and patch size selected, after that each line of the file will be read, the sampling pairs obtained from each line and their binary test made, when the file ends the descriptor will have the selected size and made the description according to pattern previously generated in Matlab.

3.5 Matching

This section regards the matching phase.

3.5.1 Matching Hardware Specification

Memory Access. There is a need to store and read the descriptors, the inputs of the matching system. This section of the Design is dedicated to approaching the memory system and controller. The Requirements for the design are for it to store a considerable amount of data and a fast and accessible operation of reading this data.

Since the number of descriptors in each image is considerable and the amount of memory that these descriptors are gonna take is proportionally considerable, it is necessary to evaluate what kind of memories exist in an FPGA and which can solve this problem.

The memory elements available to use are the Random-Access Memory (RAM), Read-Only Memory (ROM), or shift registers. These elements are RAMs, LUTs, and shift registers. The type of memory that will be used is the random-access memory.

The two types of random-access memory considered to solve the problem at hand were the BRAM and the DRAM. Looking further into the characteristics of each one, there is a major handicap in using distributed RAM considering the board limitation in resources, which is the fact that it is constructed using LUTs. However, DRAMs are asynchronous whereas block rams are synchronous, which presents itself as an advantage in favour of DRAMs, since synchronous reading and writing is not only slower, but also brings forward a new design constraint to control the memory.

Considering the hardware limitations in resources and doing a trade-off, the best option is to use the BRAM instead of DRAM because the LUTs are precious resources for the board.

Controller and BRAMs. The descriptors coming from the BRIEF stage will be stored in two different single port BRAMs, one for each image. To transport the different descriptors to the matching cores and start the matching process, a controller will be used. This controller will allow the BRIEF stage to write descriptors in the BRAMs inside the module and count how many descriptors exist for each images, which is outputted through NumDesclImage1 and NumDesclImage2. This will be useful to inform the matching stage about how many descriptors of each image are left.

The controller reveals to be an important feature because it is responsible to start and reload N patterns to the matching cores. Two clock cycles are required to read the first address after the write_enable2 signal is active. This will happen when all descriptors of the first image are stored in the first memory and outputted through the CommonPart, and the number of descriptors from the second image equals the number of matching cores. To reload the patterns, it is necessary that the matching stage informs the controller that the comparisons between the descriptors are already finished, which is done by using the result_ready signal. In figure 3.5.1 it is possible to see the inputs and outputs of the system.

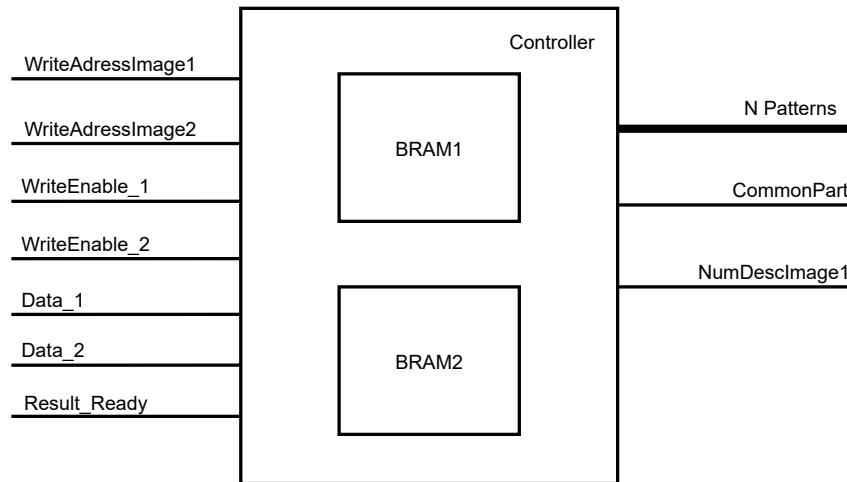


Figure 3.5.1: BRAM design for storing and getting the descriptors

Brute-Force Hardware Overview. The diagram present in figure 3.5.2 globalises every algorithm of the brute-force phase, summarising the different stages of the brute-force process. Firstly, the two memories receive the descriptors and the first operation to do is the XOR operation, with one descriptor from each image. Next, the hamming distance is calculated, using the result of the XOR operation. The distances calculated are used in the different iterations that are done afterwards, being the minimal distance the best match for the descriptor.

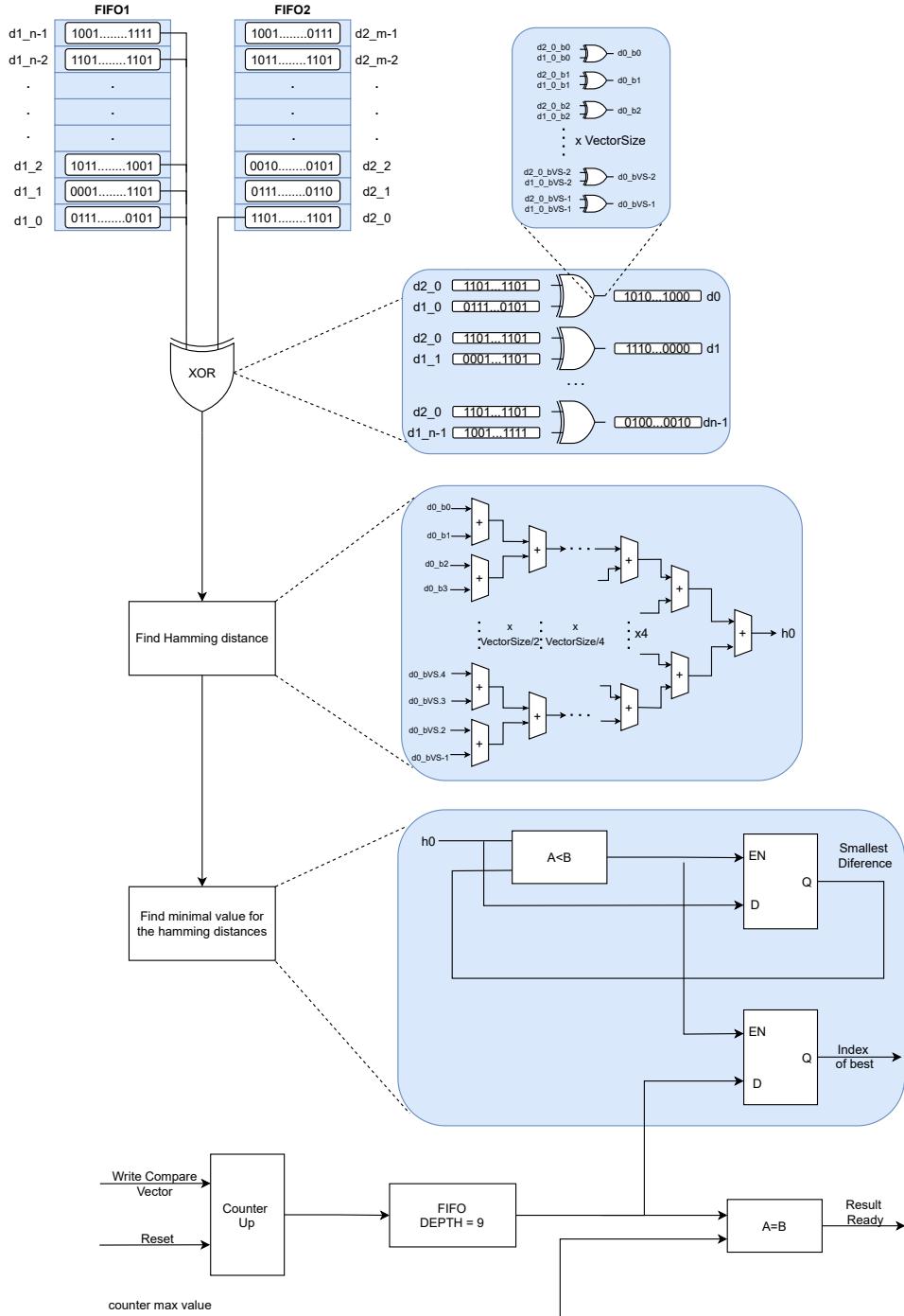


Figure 3.5.2: Matching Hardware Overview diagram

1. The descriptors provided by the BRIEF stage may change in the vector size, being able to take the values of 64, 128 or 256 bits, represented in this diagram as *VectorSize* which inherently changes the number of clock cycles and adders required for calculating the hamming distance. The descriptors of the first image and the

second image are stored in FIFO1 and FIFO2, respectively. The number of descriptors can also vary from image to image depending on how many keypoints have been detected by the BRISK or the ORB algorithms, which is represented in this diagram with the values n and m .

2. The first "XOR" operations of the first descriptor in both images are implemented in parallel and its used to exploit the bits that vary between the two descriptors. When the comparison between the first descriptor of both images is done, the second "XOR" operation will be done between the first descriptor of the second image and the second descriptor of the first image, going like this until all descriptors from the first image have been compared to the first descriptor of the second image. At that point, the same operation will be done with the second descriptor of the second image, incrementally, until all descriptors from image two have also been compared.
3. After doing each "XOR" operation, the immediate thing to do is calculating the hamming distance. This is done by using adders between all the bits of each vector coming from the "XOR" operations. It can be seen in the diagram in the Find Hamming Distance block, where from $d0_b0$ to $d0_bVS-1$ adders will be implemented until there is only one value, $h0$, which represents the hamming distance. This will also be done for all vectors from $d0$ to $dn-1$.
4. These values of hamming distances will then be compared in order to check if this new hamming distance is the minimal distance or if it can be despised, which can be seen in the find minimal value for the hamming distances block. If the value coming from the find hamming distance block is lesser than the previous smaller hamming distance value, this new value will then be stored.
5. Saving the smallest difference is not the only thing that must be saved, as it is also necessary to save the index corresponding to this smallest difference. For this we have the module present at the end of the diagram. Where the index value of the hamming distance being compared being saved when this value is smaller than the previous one. When the index value of the descriptor from image one being compared equals the counter max value, the result is ready and the next descriptor from image two can then start being compared to all descriptors of image one.
6. Once the minimal value for the hamming distance is found, it is necessary to output the pairs of descriptors that are the best matches.

Matching Core Diagram. One of the chosen methods to accelerate the matching process consists on using several matching cores. Each matching core will compare one descriptor from image two with one descriptor from image one, being the descriptors from image 1 common to all matching cores, since the descriptors from image one will be all stored first in a BRAM. This technique is a trade-off between resources and speed, allowing the use of fewer resources and guaranteeing an increase in performance. It is also important to mention that each matching core will implement a pipeline technique, thus providing an increase in speed when comparing descriptors. The use of multiple matching cores (parallelism) allows the comparison of multiple descriptors simultaneously, speeding up the matching process. This process becomes even faster due to the application of the pipeline in each matching core. The matching core diagram is represented in figure 3.5.3.

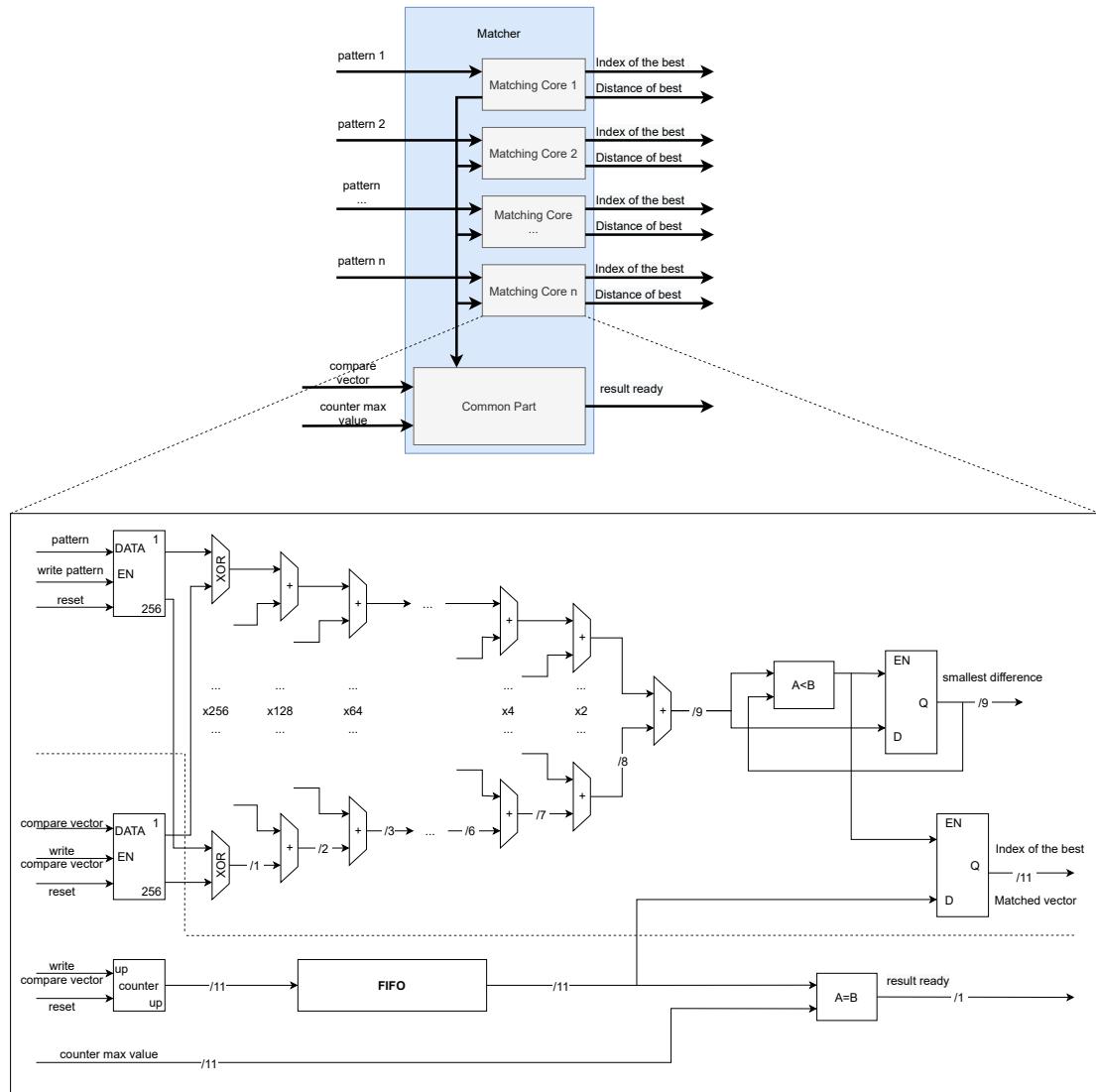


Figure 3.5.3: Matching Core Diagram

Pipelining and Parallelism. In order to achieve an even higher processing speed, extensive pipeline is employed to compute the Hamming distance between the two descriptors. With pipeline, if a descriptor has 256 bits, the first calculation of the hamming distance between two descriptors takes nine clock cycles to be computed (taking eight or seven clock cycles if the descriptors have 128 or 64 bits, respectively). Each clock cycle following the ninth clock cycle will have already calculated the next hamming distance, not requiring nine clock cycles for each comparison that will follow, thus speeding up the process considerably.

The first stage receives one descriptor for each clock cycle, computing the XOR operation between the descriptors. The result is passed to the next stage on the rising edge of the next clock cycle and the next stages will be the sums of the result of the first stage.

The next diagram 3.5.4, explains in more detail how pipeline works in the system.

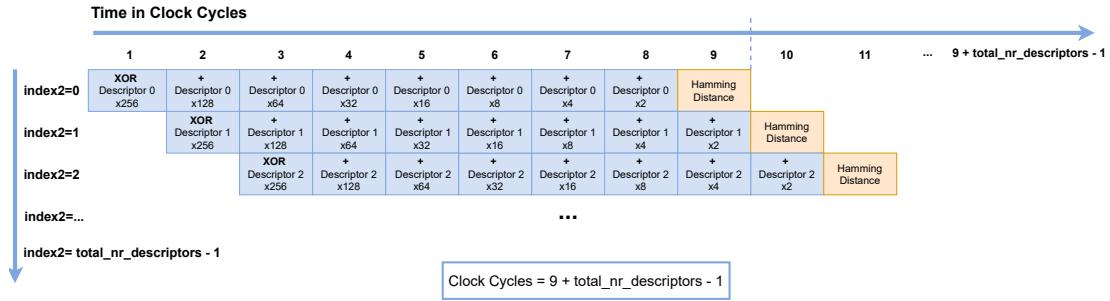


Figure 3.5.4: Pipelining for each matching core

Along with pipelining, there is parallelism in the matching process, as was referenced before. All descriptors in memory 1 are compared in pipeline with one descriptor in memory 2. This process happens at n ($n=6$ in the figure below) matching cores at once. When the comparisons with all the memory 1 descriptors is finished, the matching cores will load new values of descriptors from memory 2 and the pipelining process starts again with all descriptors from memory 1, which is the common part. Figure 3.5.5 helps to fully visualise this parallelism and pipelining process in the matching phase.

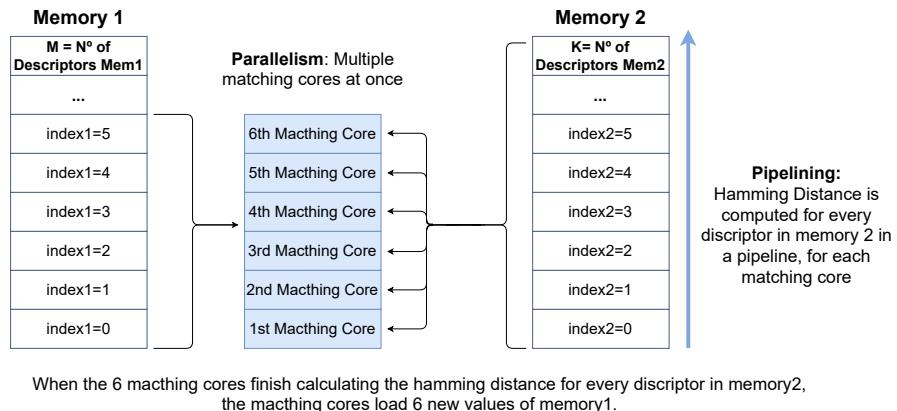


Figure 3.5.5: Parallelism and Pipelining in Matching

Cross Checking. The last part of the matching phase implemented in hardware is the cross checking technique, which is done when all the best matches for the descriptors from image two are calculated.

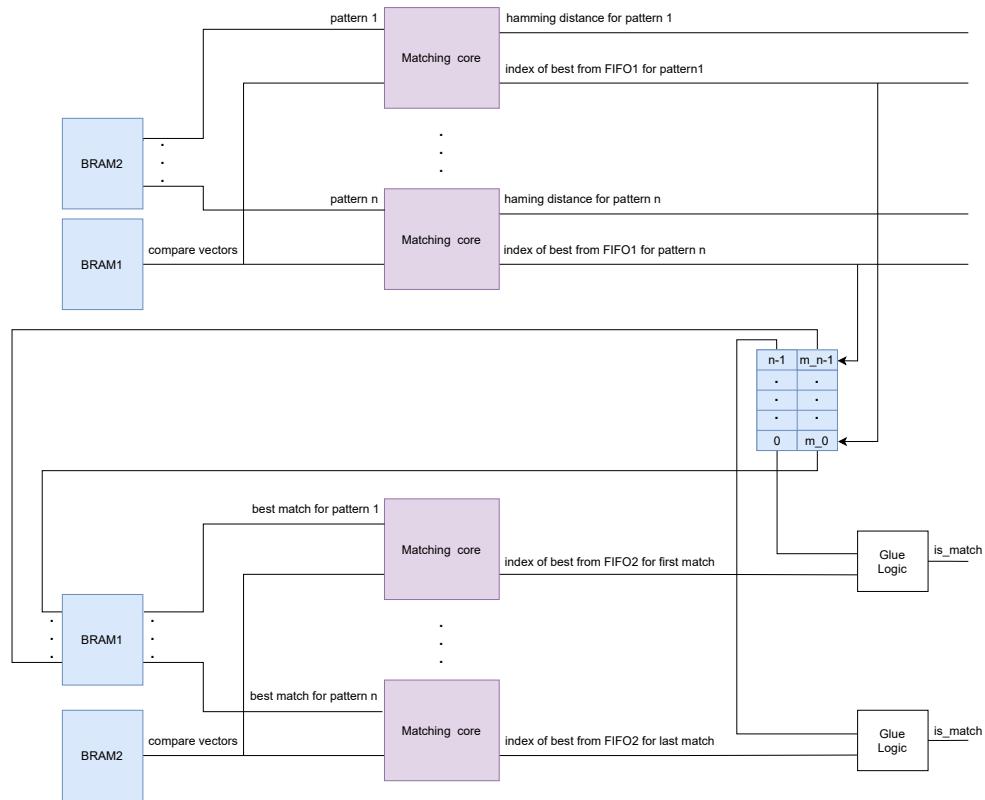


Figure 3.5.6: Cross Checking diagram

At the top part of this diagram present in figure 3.5.6 the simple process of the brute force algorithm is described in order to easily understand how the cross checking algorithm will be done.

The cross checking algorithm will only be implemented once the matches for all patterns of BRAM2 are found, since to do the inverse process of comparing one descriptor stored in the BRAM1 will all descriptors from BRAM2, all keypoints from the second image must be already described and stored in BRAM2 by BRIEF.

As can be seen from the diagram above, the best match from image one, for each descriptor from image two, will be stored so that it can be accessed later, once the brute-force algorithm has been applied to all descriptors from image two. When this is done, the cross checking algorithm can start by indexing BRAM1 (corresponding to the first image), with the best match for the first descriptor in BRAM2 (from the second image). Then, this descriptor from image one will be compared to all descriptors from image two, basically doing the inverse process that was done before. The output from this comparison will be an index from a descriptor from image two, which will then be compared to zero, represented in the diagram by the glue logic block, to check if they are the same index and see if the cross checking was successful. If the indexes are the same, then there is in fact a match. If not, the match can be discarded. This process will be repeated to all the best matches for image two that were previously stored, implementing the whole cross checking algorithm.

3.5.2 Software and Algorithm Specification

As far as software is concerned, everything must be sequential, which means that each of the descriptors of image 1 will have to be compared with all the descriptors of image two, one by one, it is expected that this method will be far slower than its hardware implementation that is described in the next section. Figures 3.5.7 and 3.5.8 describe, sequentially, the steps to implement such an algorithm and are self-explanatory to an extent, each descriptor associated with image one is compared (through the use of a XOR operation and the comparison of the Hamming distance) to each descriptor associated with image two, and afterwards the mismatches are eliminated; two details will also be specified: the **XOR** process exists merely because the XOR operator is bitwise and some manipulation will be required to perform a XOR between two descriptors; the **calcHammingDistance** process consists of merely counting the number of 1s that exist in the output of the XOR as such it was regarded as a simple algorithm whose detailed implementation that would only increase the visual complexity of the flowcharts.

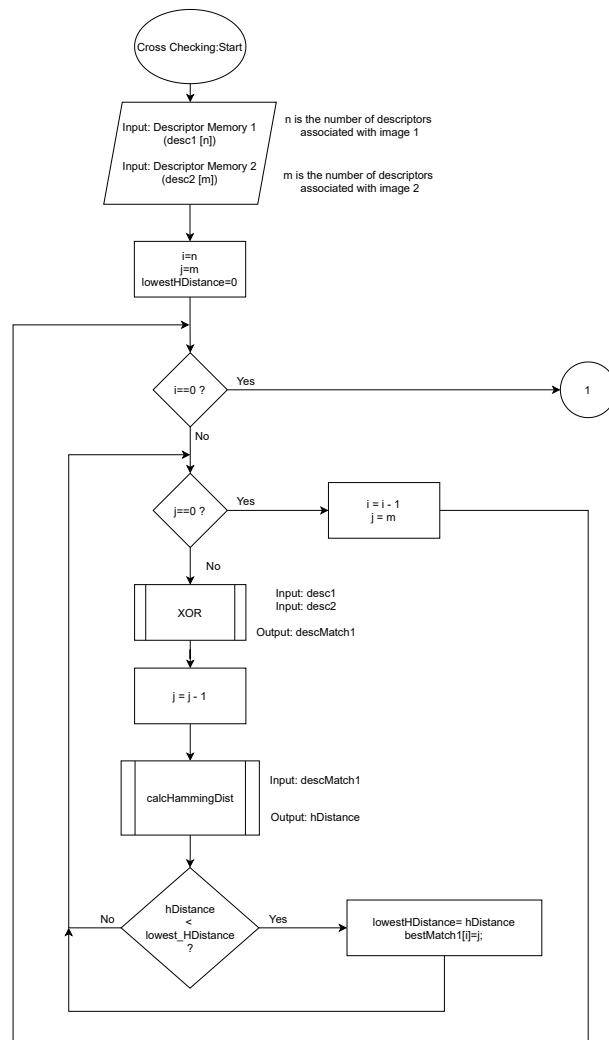


Figure 3.5.7: Cross-Checking Flowchart part 1

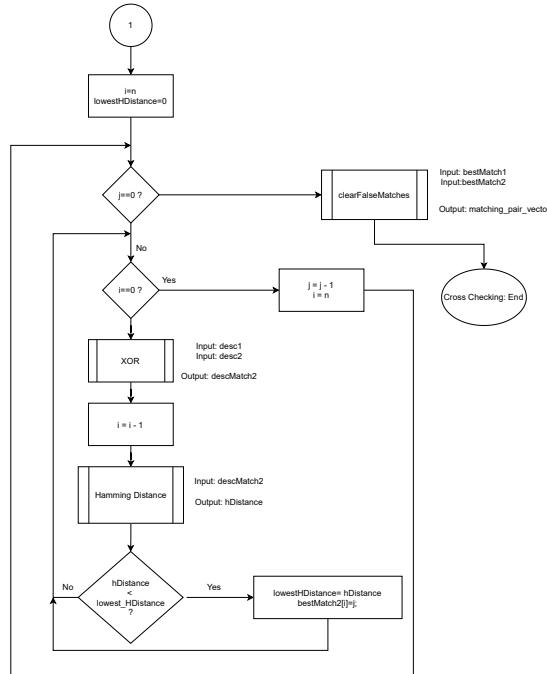


Figure 3.5.8: Cross-Checking Flowchart part 2

Figure 3.5.9 consists of the flowchart containing the most important step in cross-checking, namely removing all matches that are not coherent, for instance if the best match for descriptor number one of image one is descriptor two of image two, but the best match of two of image two is descriptor five of image one, there is a lapse in coherency and as such both matches are considered false/irrelevant. This step is the entire point of the cross checking and allows the use of a simpler approach when it comes to the outlier removal, since cross-checking will already remove a large portion of said outliers.

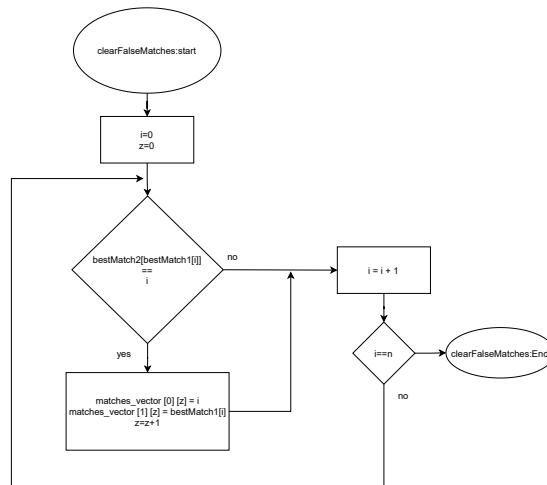


Figure 3.5.9: Cross-Checking Flowchart part 3

The slope-based rejection aims to filter out **incorrect matches** if their **slope** is outside of the mean of all slopes of matches in both images, within a threshold tolerance. Firstly, one receives as input the FIFO containing the coordinates of the key points of both images (coord_fifo1, coord_fifo2) from BRIEF and a 2D array containing the index of the matched keypoints (matches_vector) from the cross-checking stage. Then, it appends on the matching_pairs_vector, in each position, the rows and columns of the two matched key points to calculate slope, **using equation 2.10**. After that, it only appends on the final vector (correct_match_vector) the elements of matching_pairs_vector whose slope verifies the condition stated above in the first statement of this paragraph.

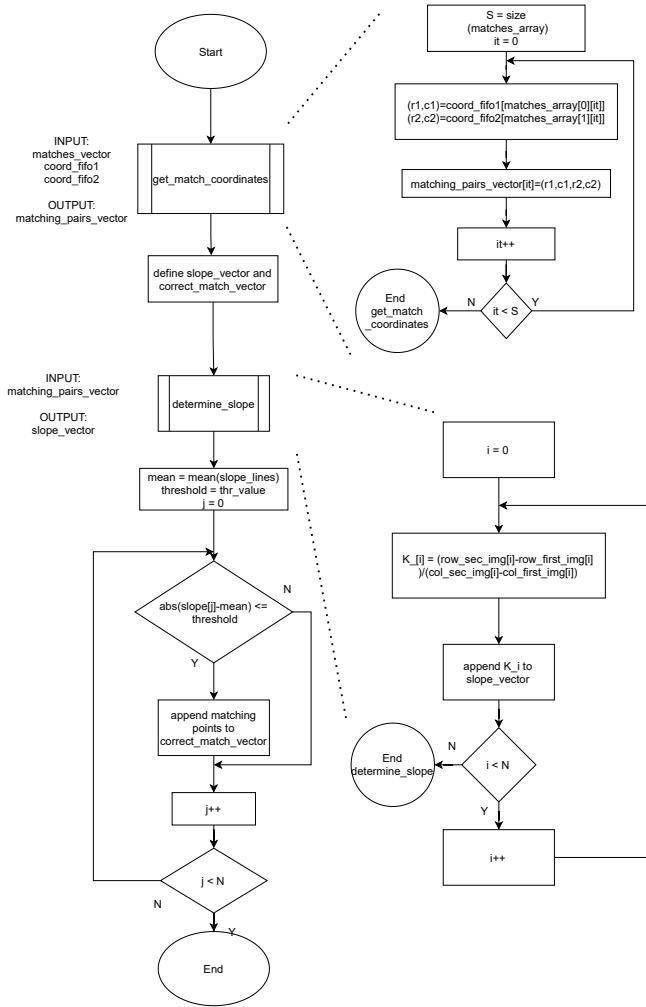


Figure 3.5.10: Slope-based Rejection Algorithm

3.5.3 Class Diagram

The class diagram is defined in the figure 3.5.11 below. The diagram contains one class descriptor and one class keypoint, where both are elements of the main (matching) class. This class in turn contains the methods that will be used in the matching process.

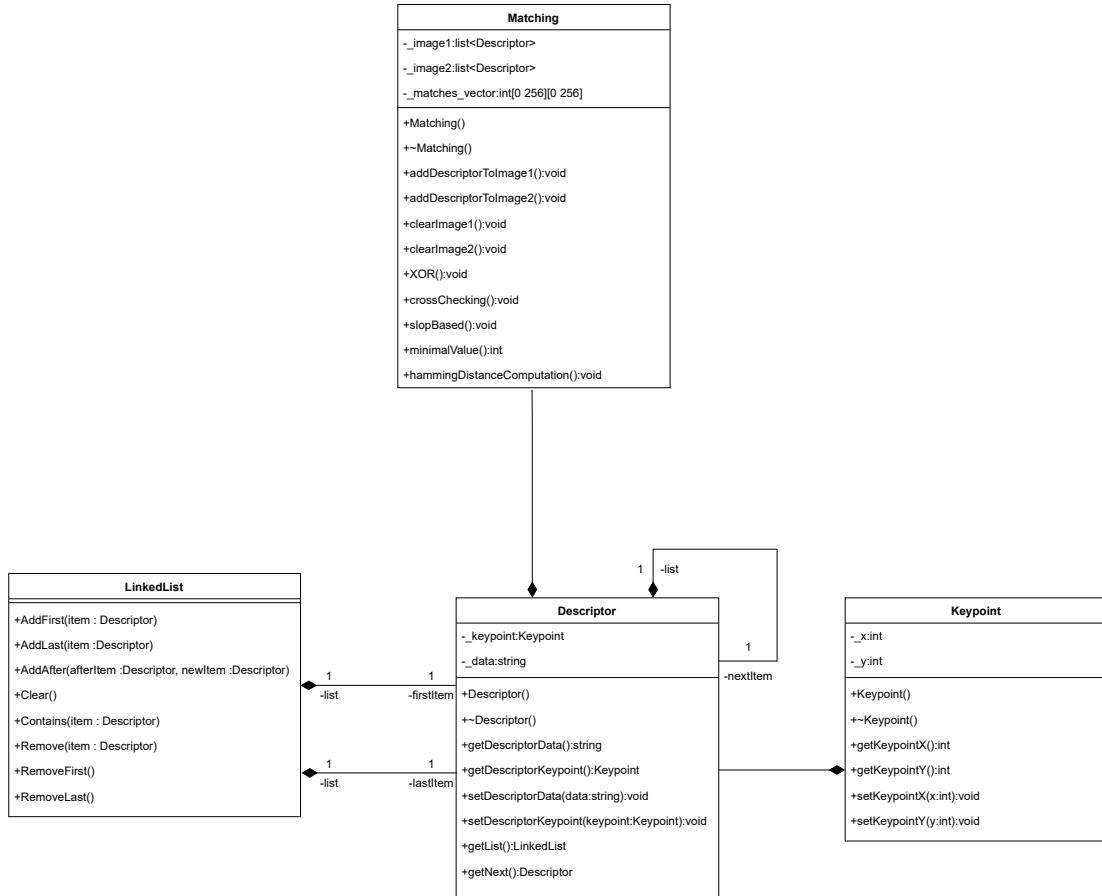


Figure 3.5.11: Matching Class Diagram

3.6 DSL Design

3.6.1 Syntax

After the definition of all the variables and keywords that will be used, in the design phase is time to describe the syntax. Below are presented the syntax rules for the DSL. The syntax that will be implemented has a certain abstraction with an intuitive nomenclature, in order to simplify the adaptation of the user to the DSL.

The syntax will be implement in a JSON format in order to be simpler to the user to understand the script organization.

Listing 3.6: DSL Syntax

```

Model -> ProgramType
ProgramType -> System | Generation
4
System -> "System" ID "{" OperationMode DetectorType Descriptor "}" ";"
OperationMode -> "OperationMode" ":" ("SW" | "SW-Refactored" | "Hybrid") ";"
```

```

9  DetectorType -> ORB | BRISK
10
11  ORB -> "ORB" "{" NFAST_N Threshold "}"
12
13  BRISK -> "BRISK" "{" NFAST_N Threshold "}"
14
15  NFAST_N -> "NFAST_N" ":" INT ";"
16
17  Threshold -> "Threshold" ":" INT ";"
18
19  Descriptor -> "BRIEF" "{" PatchSize StringSize "}" ";"
20
21  PatchSize -> "PatchSize" ":" INT ";"
22
23  StringSize -> "StringSize" ":" INT ";"
24
25  Generation -> " initialize " ID ";"

```

3.6.2 Validation Rules

The parameters of the system have a limited range because of their meaning in the algorithms. That said, there is a need to define some validation rules to define the ranges that the values can have.

The IDE that will be used allows a constant real-time validation of the code that the user is writing. If the user gives a value to a variable out of the range it will be displayed an error message to warn the user that the chosen value is prohibited, and for that reason it will be impossible to generate code if the user doesn't change the parameter's value.

Production	Rule
ORB{ N-FASTN Threshold }	N-FASTN = 5 9 12 0 <= Threshold <= 255
BRISK{ N-FASTN Threshold }	N-FASTN = 5 9 12 0 <= Threshold <= 255
BRIEF{ StringSize PatchSize }	StringSize = 64 128 256 PatchSize = 11 15 19
OperationMode	OperationMode=Sw SwRefactored Hybrid

Table 3.1: Validation Rules

3.6.3 Code generation approaches

Code generation is the last step of a DSL, responsible to generate an executable code in accordance to the the customisations chosen by the DSL user. Those configurations are introduced in the user script and are combined with the non-executable annotated files inside the code repository. In order to do so, different approaches can be selected. The most common and intuitive one is to annotate the code in the repository with some special character not used by the language at hand. The special character that will be used is '@', and using this character at both sides we can define a section as an annotation, i.e. @something@. Then, a generic function that receives a file, the special character and the system to access the values the user desires will parse the file and look for the special character. Once two of this special character is found, the string inside it will be compared with an array with all the possible variables and if the annotation corresponds to an existent variable, then the annotation will be replaced by the value of the variable inserted by the user that corresponds to that annotation. On the other hand, if the variable inside the special characters does not exist then nothing will happen.

This implementation is straightforward and very good to implement, since it is secure and will one hundred percent replace the annotation with the desired values if it is well implemented. However, sometimes other approaches are needed. For example, if the code of a file is to have many annotations or the modification of a variable implies the modification of many other variables or different implementations of an algorithm, then maybe it is better to go for other approaches. For this, one thought of other implementations, such as:

- Use the #ifdef directive
- Do selections inside an annotation
- Choose a file from different files inside the repository according to the parameter chosen by the user

Since the #ifdef directive is sometimes linked to the problem known as the #ifdef hell, this option was discarded and it was decided to go with the other two possibilities depending on what is more beneficial in the case at hand. The third approach is very simple since no annotations have to be made and is better for larger scripts, and the second approach is harder to implement but is better for smaller scripts. In the following fig. 3.6.1 it can be better seen how this second approach works.



```

#include <stdio.h>
void dsl_test()
{
    int test_var = 1;
    int descriptor_size = @StringSize@;
    int patch_size = @PatchSize@;
    @IF StringSize == 64@
        test_var = 2;
    @ENDIF@
    @IF StringSize == 128@
        test_var = 3;
    @ENDIF@
    @IF StringSize == 256@
        test_var = 4;
    @ENDIF@
    @IF PatchSize == 11@
        test_var = 5;
    @ENDIF@
    @IF PatchSize == 15@
        test_var = 6;
    @ENDIF@
    @IF PatchSize == 19@
        test_var = 7;
    @ENDIF@

    test_var = @@;
}

```

```

#include <stdio.h>
void dsl_test()
{
    int test_var = 1;
    int descriptor_size = 256;
    int patch_size = 11;
    test_var = 4;
    test_var = 5;

    test_var = @@;
}

```

Figure 3.6.1: Second approach

On the left side is the annotated code, and in the right is the generated code, and according to what was specified in the script, the desired block of code will be selected. This verification is simple and goes like this: @IF *variable* *operator* *value to compare*@ block of code @ENDIF@. If the value to compare is equal to the one defined in the script, then that block of code will be the one selected, if not, that it will be deleted from the file. It is important to note that these blocks of code can also have simple annotations inside them, as can be seen in the figure, but if the content of this annotation does not correspond to any valid annotation that needs to be replaced, then it will still remain in the final code, generating an non-executable code, as can be seen in the last line of the file, so it is very important to implement the annotations correctly. The following flowcharts help to understand how this second approach works.

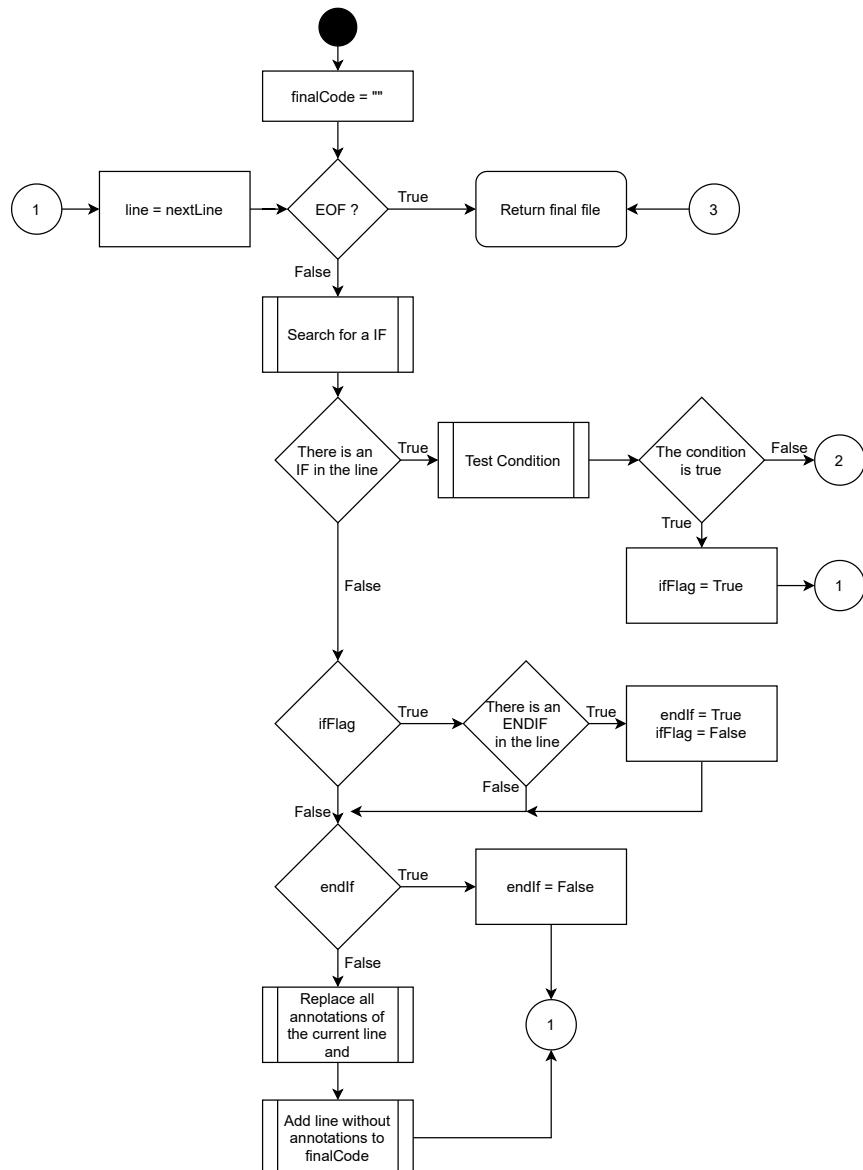


Figure 3.6.2: Flowchart of the generic function to replace annotations

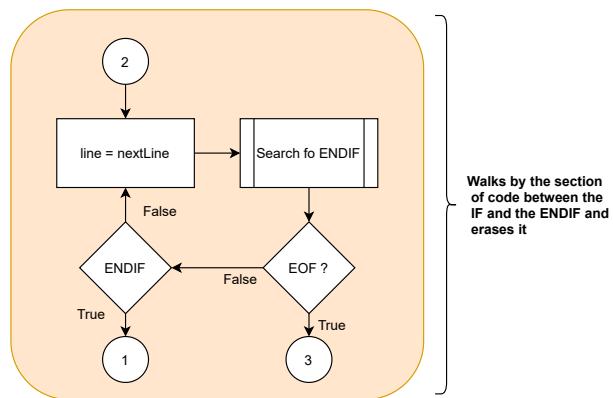


Figure 3.6.3: Flowchart of how the code inside false condition is deleted

3.7 Profiling

This section regards the design for the profiling part of the project.

3.7.1 Tool choice

To profile the OpenCV we choose VTune and OProfile and below is a table that demonstrates a gap analysis between the two.

Parameter	Instrumentation Profiling - VTune	Sampling OProfile	Profiler -
Overhead Level	High	Low	
Code Injection Level	High	Low	
Multiplatform	No	Yes	
Hotspot Detection	Yes	Yes	
Thread Tracking	Yes	No	
Memory access Tracking	Yes	No	
Anomaly detection	Yes	No	

Table 3.2: Gap analysis table between VTune and OProfile

3.7.2 Test Design

Oprofile Test Design

Image processing applications present an ideal type of application to tune with OProfile. The image processing applications are CPU intensive.

The first step in reducing the amount of time a program takes to execute is to find where the program spends time. Reducing the program hot spots will have the greatest impact on performance. By default OProfile uses the time-based metrics. The fig. 3.7.1 shows the time-based events for various platforms.

Processor	Default Event for Counter	Description
AMD Athlon and AMD64	CPU_CLK_UNHALTED	The processor's clock is not halted
AMD Family 10h, AMD Family 11h, AMD Family 12h	CPU_CLK_UNHALTED	The processor's clock is not halted
AMD Family 14h, AMD Family 15h	CPU_CLK_UNHALTED	The processor's clock is not halted
Applied Micro X-Gene	CPU_CYCLES	Processor Cycles
ARM Cortex A53	CPU_CYCLES	Processor Cycles
ARM Cortex A57	CPU_CYCLES	Processor Cycles
IBM POWER4	CYCLES	Processor Cycles
IBM POWER5	CYCLES	Processor Cycles
IBM POWER8	CYCLES	Processor Cycles
IBM PowerPC 970	CYCLES	Processor Cycles
Intel Core i7	CPU_CLK_UNHALTED	The processor's clock is not halted
Intel Nehalem microarchitecture	CPU_CLK_UNHALTED	The processor's clock is not halted
Intel Pentium 4 (hyper-threaded and non-hyper-threaded)	GLOBAL_POWER_EVENTS	The time during which the processor is not stopped
Intel Westmere microarchitecture	CPU_CLK_UNHALTED	The processor's clock is not halted
Intel Broadwell microarchitecture	CPU_CLK_UNHALTED	The processor's clock is not halted
Intel Silvermont microarchitecture	CPU_CLK_UNHALTED	The processor's clock is not halted
TIMER_INT	(none)	Sample for each timer interrupt

Figure 3.7.1: OProfile CPU-CLK-UNHALTED

Because the experiments are being performed on a Intel iCore i7, we will use the CPU-CLK-UNHALTED.

After the choice of the event for counter, and the oprofile correctly installed, we can start profiling. In order to do that we must write on command prompt the `opercf` command, indicate the event for counter chosen before, the count and the name of the executable, like exemplified below in listing 3.7.

Listing 3.7: Execution of the `opercf` command

```
opercf --append --event CPU_CLK_UNHALTED:100000:0:0:1 ./program_exe
```

The program will be sampled, now we want to see the report, the results of that sampling. There are numerous ways to see the results, but in this case, we considered that the most adequate is with a "Symbol summary for a single application including libraries". In order to obtain that, the following command must be insert listing 3.8.

Listing 3.8: Execution of the `opreport` command

```
opreport --demangle=smart --symbols 'which lyx'
```

With this report we are able to see every function and respective library executed by the application, how many samples the function collected and the percentage of CPU used.

Finally, to add more certainty and credibility to our tests we will implement a bash script that receives as inputs the count, the application name and the number of executions we want to sample and outputs a report with the samples taken from all of the program executions. For example, in one single execution an hotspot may not be clearly noticed, but it will be, for sure, noticed in 1000 executions.

Intel VTune Test Design

To profile using VTune tool the group proceeded to install both visual studio and the OneApi package from Intel. The OneApi package comes with many different tools to help the user set up a programming environment yet the only one that matters to the group is the VTune profiler and a few dependencies. The operating system chosen was Windows, because Intel's support is much better for this operating system than for other open source OS's, such as Ubuntu. After setting Up the tools the profiling process is really simple and intuitive. The following steps show how to profile any program with VTune.

First click the button as shown in the fig. 3.7.2 to start VTune.

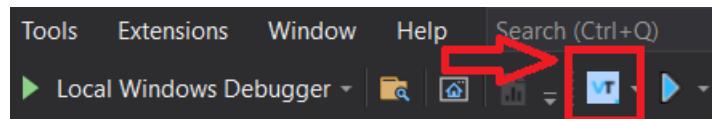


Figure 3.7.2: VTune Profiling 1

Then, click configure analysis to start a new profiling session and now it is possible to configure the analysis that will be done. The WHERE window allows the user to select where the program will be profiled, as seen in fig. 3.7.3a. Additionally, there's also a window that allows the user to select what he wants to profile (fig. 3.7.3b).



(a) VTune: Selection of the host type



(b) VTune: Selection of the application to profile

Finally, the How window allows the user to select certain parameters of the analysis regarding algorithms, parallelism, microarchitecture and accelerators, as seen in fig. 3.7.4. After pressing the play button the program will be executed until either the user stop the program or the program finishes, during this time the profiler will be

retrieving data. When the program finishes the execution the profiler will show a screen to navigate through the multiple tabs to be able to see different types of analysis of the program.

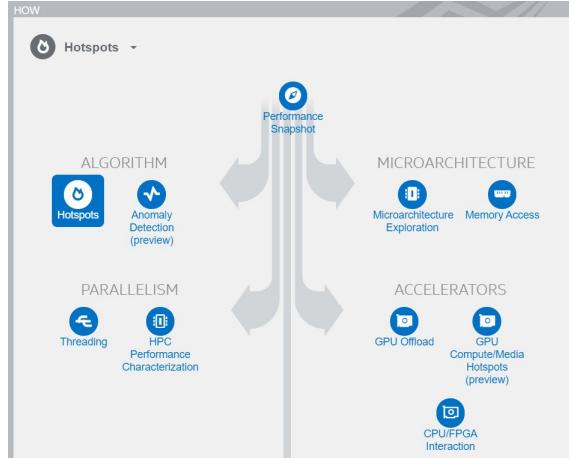


Figure 3.7.4: VTune: Parameter Configuration

3.8 Test Cases

This section will present the project's test cases, stipulating what needs to be done to ensure the requirements and constraints of section 2.2 are met.

3.8.1 DSL Unit and Integration Tests

In the tables below, table 5.17 and table 5.18 are presented all the tests that will be executed in order to verify that the DSL is working like expected. It will be used some scripts with different error as input to make sure that all the errors (lexical, syntactic and semantic) are detected. Then, is also needed to verify if the code was generated and all the variability points of the annotated code were changed with the value given in the input script. Finally, it will be tested the final version where it will be used the annotated code of the BRISK algorithm, and it will be tested the generated code with the values from the input script.

Unit Test	Expected Results
Test all instructions individually	All instructions function correctly
Use DSL code to check for lexical errors	All lexical errors were detected
Use DSL code to check syntactic errors	All syntactic errors were detected
Use DSL code to check semantic errors	All semantic errors were detected
Use DSL code to check the code generator	The C++ and verilog code was generated

Table 3.3: DSL Unit Tests

Integrated Test	Expected Results
Use configurated system written in DSL to generate code	The code was generated correctly

Table 3.4: DSL Integration Tests

3.8.2 ORB Detection Unit and Integration Tests

The unit and integration tests for ORB are presented in table 3.5.

Unit and Integrated Tests	Expected Result	Real Result
Verify the patch generation	Correctly generate the patch	
Invalid data signalling	Correctly identify edge cases in patcher	
Verify dark/bright classifier stage	The potential keypoints were classified as expected	
Verify contiguity test and scoring stage	The keypoints were all detected	
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	
Verify lists organisation	The lists were organised as expected	
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	

Table 3.5: ORB Unit and Integration Tests

Unit and Integrated Tests	Expected Results	Real Results
Verify detection stage	The potential keypoints were detected as expected	
Verify contiguity test and scoring stage	The keypoints were all detected	
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	
Verify lists organisation	The lists were organised as expected	
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	
Compare execution times	Lower execution time	

Table 3.6: ORB - Test Cases for Software Implementation

3.8.3 BRISK Detection Unit and Integration Tests

The unit and integration tests for BRISK detection are presented in table 3.7.

Unit Test	Expected Result	Real Result
Request image fetch	Image is fetched and written to the internal buffer	
Request detection	Pixels correctly identified as bright or dark	
Program image location and size	Configure memory access controller	
Measure the detector's throughput and latency with Vivado	Low latency and high throughput	
Compare BRISK detection in Sw and Hw	Less latency on the Hw	
Integrated Test	Expected Result	Real Result
Trigger detection	Pixel classification and other scoring and NMS inputs	

Table 3.7: BRISK Detection Unit and Integration Tests

3.8.4 BRISK Scoring and Contiguity Unit and Integration Tests

The unit and integration tests for BRISK scoring are presented in table 3.8.

Unit Test	Expected Result	Real Result
Input a contiguous state	Valid keypoint	
Input a non contiguous state	Not a keypoint	
Input bright/dark pixels	Get score value	
Compare pixels scores	Get maximum score value	
Test throughput and latency	Less latency on the Hw	
Integrated Test	Expected Result	Real Result
Combining Detection and Scoring Stages	Get Score and Contiguity Test at the same time	

Table 3.8: BRISK Scoring and Contiguity Test Unit and Integration Tests

3.8.5 BRISK NMS Unit and Integration Tests

In the table 3.9 are the NMS tests that will be needed to verify if the algorithm is working as expected.

Unit and Integrated Tests	Expected Result	Real Result
Verify detection stage	The potential keypoints were detected as expected	
Verify contiguity test and scoring stage	The keypoints were all detected	
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	
Verify lists organisation	The lists were organised as expected	
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	
Compare execution times	Lower execution time	

Table 3.9: BRISK NMS Unit and Integration Tests

3.8.6 BRIEF

As we can see in section 3.8.6, the purpose of these test cases is to verify the different stages of the BRIEF descriptor, and here will be described the main ones. Firstly the reception of the keypoint is confirmed from the detection stage. After that, is certified the pair generation, generated accordingly to the geometry chose, and if the

descriptor computation saves the result correctly in the binary string after the system reads the pixel intensity. Finally, the descriptor dispatches correctly to the matching stage.

Unit and Integrated Tests	Expected Results
Keypoint location reception	Receive keypoint location from detection stage
Pair generation	The patch-pairs are successfully generated
Descriptor computation	The binary string for the descriptor is generated from the binary tests
Verify the space allocation	The correct space was allocated
Verify the patch	The correct area was selected with the keypoint at the centre
Verify the pattern generation	The pattern was generated has intended
Descriptor dispatch	The descriptor is sent to the matching stage
Descriptor repeatability	Returns the same vector

Table 3.10: BRIEF Unit and Integration Test Cases for Hardware Implementation

Unit and Integration Tests	Expected Results	Real Results
Verify the pattern generation	The pattern was generated has intended	
Verify the Descriptor Computation	Generates the expected descriptor	
Descriptor Repeatably	Returns the same string	
Compare execution times	Lower execution time	

Table 3.11: BRIEF Unit and Integration Test Cases for Software Implementation

3.8.7 Matching

The test cases for the matching can be consulted in tables 5.27 (Hardware) and 5.28 (Software) and almost all were successful, with the connection between PS and PL not working, due to AXI problems.

3.8.8 Hardware

Unit and Integrated Tests	Expected Results	Real Results
Matching Core best descriptor detection	Best descriptor detection	
Reading and writing in the BRAM	Read and write data in the BRAM at the correct address	
Coordination between the controller and BRAM	Connection between the controller and the BRAM	
Information exchange between the controller and 1 matching core	Load the descriptors to the matching Core and receive the best matches and signals in the controller	
Information exchange between the controller and the 3 matching cores	Load the descriptors to the 3 matching Core and receive the best matches and signals in the controller	
Cross checking	False Matching signalled	
Applying more than one matching Core	Reduction of time in the matching process	
Connection between the PS and the PL	Exchange information between the PS and the PL	

Table 3.12: Matching Unit and Integration Test Cases for Hardware Implementation

3.8.9 Software

Unit and Integrated Tests	Expected Results	Real Results
Load linked list descriptors	Correct reading of all descriptors	
Slope based rejection	Remove false matches	
Comparison of the results obtained with the implemented algorithm and the OpenCV	High accuracy	
Brute force	Find the best matches	
Cross Checking	False Matching removed	

Table 3.13: Matching Unit and Integration Test Cases for Software Implementation

4 | Implementation

This chapter will present the implementation guidelines one must follow to achieve the intended results, attending to what was stipulated in the design phase. All the steps will be described in detail to ensure rich project documentation. This matters, as the document might be used for future reference. Specifically, this section will approach some important HDL code, whilst discussing some implementation concerns.

4.1 ORB

4.1.1 Corner Score Computation

Scorer. The inputs and outputs of the full score module are in the list 4.1.

Listing 4.1: Scorer Inputs and Outputs.

```
module full_scorer (
    input clk,
    input rst,
    4   input [15:0] is_dark,
    input [15:0] is_bright,
    input in_valid,
    input stall,
    9   output [4:0] score,
    output is_corner,
    output out_valid
);
parameter fastN = 12;
```

In order to a better identification of all system signals, input and output was establish the followed list with their meaning.

- *clk* - clock signal,
- *rst* - negative logic signal to reset the module,
- *is_dark* - array with the dark bits,
- *is_bright* - array with the bright bits,
- *in_valid* - signals that the inputs are valid, and should start operating,
- *stall* - signals that the module's pipeline must stop/stall,
- *score* - the score of the keypoint,
- *is_corner* - signal that classifies, the keypoint as a corner or not,
- *out_valid* - signal's that the data displayed by *score* and *is_corner* are valid.

The *reset* signal will be the highest priority one, followed by the *stall* signal and finally *in_valid*. This meaning, a higher priority signal will overshadow any of the operations that correspondent to a lower priority signal. Plus, the parameter *fastN* will be change accordingly to the desired situation by the DSL. Thus enabling our system to perform the score using different FAST version - them being 5,9 or 12.

Contiguity Counter. The inputs and outputs of the contiguity counter module are in figure 4.2.

Listing 4.2: Contiguity Counter Inputs and Outputs.

```

1  module contg_count (
2    input clk,
3    input rst,
4    input [15:0] A,
5    input [15:0] B,
6    input in_valid,
7    input [4:0] n,
8    output [4:0] scoreA,
9    output [4:0] scoreB,
10   output reg ovalid
11 );

```

In order to a better identification of all system signals, input and output was establish the followed list with their meaning.

- *clk* - clock signal,
- *rst* - negative logic signal to reset the module,
- *A* - array with the dark bits,
- *B* - array with the bright bits,
- *in_valid* - signals that the inputs are valid, and the module should start operating,
- *n* - the signals indicates the choosen fastN,
- *scoreA* - the score of the array A,
- *scoreB* - the score of the array B,
- *outvalid* - signal's that the data displayed by *score* and *is_corner* are valid.

The signal priority scheme is as said previously.

4.1.2 Modules implementation

Score Implementation. The Scorer module is responsible for forwarding the new data, when signalled, to the Contiguity Counter module. When this one finishes, its output will be used to perform the contiguity test and compute the score, and finish by signalling that new outputs are available/valid.

Firstly, the code within the *always @* block, present in the listing 4.3, is accessed whenever there is a positive rising edge of the clock. If the *stall* signal is not active, it will proceed to forward the data between each pipeline

stage. If stalling is required, it won't forward the data along the pipeline, thus keeping the current information, and only resume its operation when the signal goes back to 0.

By feeding the data through registers to the counter module, one can store its inputs and avoid an inaccurate output by having the inputs changing while acting upon them. The 3 registers (*is_en*, *bright*,*dark*) are directly connected to the counter module as will be discussed in the module integration section.

The first 3 wires(*is_corner*, *score*, *out_valid*) are connected to the contiguity counter output, and provide the necessary and simple logic to perform the contiguity test and compute the score. This result is to be considered valid only when *out_valid* signal is active. When stalling this signal should be driven to 0, to avoid any possible data loss on the receiver's end.

With the contiguity counter latency being of a 1, and only afterwards the scoring and contiguity test is performed, along with *out_valid* assignment, our module's pipeline is classified as 2-stage.

Listing 4.3: Scorer implementation.

```

3      assign is_corner = (score_bright >= fastN || score_dark >= fastN) ? 1:0; // Contiguity Test
4      assign score = (score_bright >= score_dark) ? score_bright:score_dark; // score
5
6      assign out_valid = (~rst || stall) ? 0:ovalid;
7
8      always @ (posedge clk) begin
9          if(~stall) begin
10              in_en <= in_valid;
11              bright <= is_bright;
12              dark <= is_dark;
13          end
14      end

```

The RTL Analysis schematic is presented below. It shows a loyal representation of what was expected from our design, besides some implementation details.

Figure 4.1.1: RTL Analysis Schematic.

Contiguity Counter. In order to correctly perform the count of contiguous bit, one would have to use sequential logic, therefore the use of a blocking assignment inside the for loop at the *always @* block.

As the flowchart 3.2.4 demonstrates the array must be crossed twice the circle size, thus it is used a for loop, which in an EDA will duplicate the hardware in order to perform the operation while meeting the time constraint of 1 cycle. This is where most resources will be used.

The reset condition is required for the system to initialize/boot-up.

Listing 4.4: Contiguity Counter implementation

```

1      always @ (posedge clk) begin
2          if (~rst) begin
3              countA <= 0;
4              maxA <= 0;
5              countB <= 0;
6              maxB <= 0;

```

```

    i <= 0;
    ovalid <= 0;

11     end
else begin
    ovalid <= 0;
    if (in_valid) begin
        maxA = 0;
        maxB = 0;

16        for(i = 0 ; i < max ; i = i+1) begin

21            countA = (A[( i > (n -1)) ? i-n:i]) ? countA +1:0;
            maxA = (countA>maxA) ? countA : maxA;

            countB = (B[( i > (n -1)) ? i-n:i]) ? countB+1:0;
            maxB = (countB>maxB) ? countB : maxB;

26        end
        countA <= 0;
        countB <= 0;
        maxA <= (maxA > n) ? n:maxA;
        maxB <= (maxB > n) ? n:maxB;
31
        ovalid <= 1;

        end
    end
36
    end
    assign scoreA = maxA;
    assign scoreB = maxB;

```

4.1.3 Modules integration

For the Corner Score Computation perform correctly the modules integration has to be done. This is done by instantiating the contiguity counter module and making the correct assignment.

Both modules will share the same *clk,rst* and *out_valid* signals. As far for the other signals, except *n*, will be connected to pipeline registers.*n* is generated by the parameter *fastN* which indicates, as the name suggest, which FAST approach to take.

Listing 4.5: Contiguity Counter instantiation.

```

assign circle_size = (fastN == 5) ? 8: (fastN == 9) ? 12: (fastN == 12) ? 16:16;
contg_count count(
    .clk(clk),
    .rst(rst),
5     .A(bright),
    .B(dark),
    .in_valid(in_en),
    .n(circle_size), // n(fastN)
    .scoreA(score_bright),
10   .scoreB(score_dark),
    .ovalid(ovalid)
);

```

4.1.4 NMS

In this section, the implementation of the design is presented along with the discussion of some implementation details. Only important snippets of the code are shown to help contextualise and understand each implementation.

Hybrid Approach. The implementation of this approach in C language is very straightforward, for the reasons stated in the Hybrid Approach section. As it is necessary to calculate square root, the library math.h is used to provide the method sqrt() and, for that, the linker must be informed that the math library is being used, as it can be seen in the figure below.

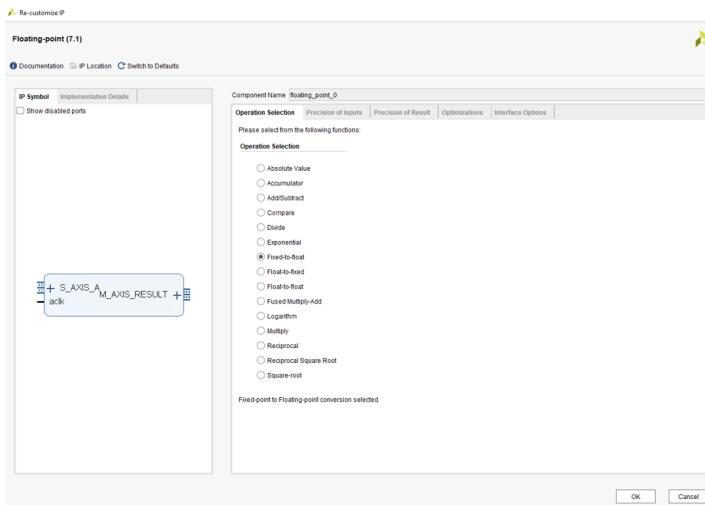


Figure 4.1.2: Linker Settings Configuration.

In order to prevent repetitive computations, the square root, sine and cosine are calculated separately so that they can be reused. This proved to be a better solution as it reduced the latency by 10 clock cycles. The latency is measured recurring to the xtime_1.h library which provides methods such as XTime_GetTime() used to interact with the global timer counter register. The execution time is obtained by the difference of time between the time at the start of the execution and the time at the end of the execution. The clock frequency of the global timer, which is always clocked at half of the CPU frequency, is then used to calculate the latency in clock cycles.

Listing 4.6: Usage of the global timer counter register for execution time calculation

```
//set up timer
XTime_GetTime(&tStart);
3
//execute code here
...
8
//end timer
XTime_GetTime(&tEnd);
```

4.1.5 Hardware Approaches

Hardware with Floating-Point Numbers Approach. Considering that this approach is implemented with a pipeline architecture, this section is split into each stage of the pipeline. The final implementation RTL schematic can be seen in the figure below, where it is also possible to see each stage and respective pipeline register with the corresponding inputs/outputs. As previously mentioned, the IP core used for all the instances is the Floating-point (7.1).

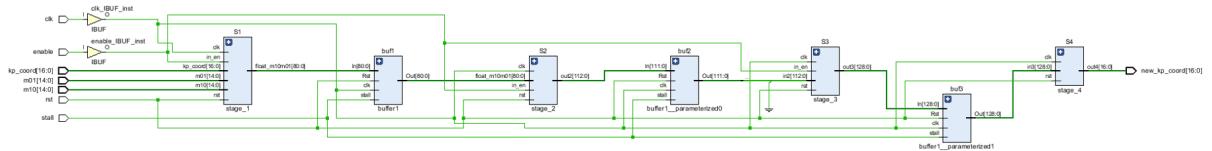


Figure 4.1.3: Hardware with Floating-Point Numbers Approach RTL.

Pipeline Registers. The pipeline registers are implemented with instances of the same module which has a parameter to set the width of the buffer, WIDTH. The incoming data is saved on an internal buffer on a synchronous manner, as shown in Code snippet.

Listing 4.7: Intermediary Pipeline Register Module

```

1  reg [WIDTH-1:0] buffer;
2
3  assign Out = (!stall) ? buffer : Out;
4
5  always@(posedge clk)
6  begin
7    if(Rst)
8      buffer <= 0;
9    else
10      buffer <= In;
11  end

```

These registers implement the stall function by preventing the output from being updated with the most recent data from the buffer, therefore the data remains in the buffers. Once the stall ends, the data stored in the buffers is fed to the modules again, thus resuming the pipeline.

First Stage. In this stage, as it was mentioned in the design, the first step is where the patch's moments are multiplied by themselves, to get the square, and then are summed. This is achieved with a non-blocking wire only connection as shown in Code snippet.

Listing 4.8: Sum of the squares of both patch's moments

```

assign sqr_m10 = (rst) ? 0 : m10 * m10;
assign sqr_m01 = (rst) ? 0 : m01 * m01;
assign s_m01_m10 = sqr_m10 + sqr_m01;

```

This value has now to be converted into the float representation and for that, this instance of the IP core mentioned in the Hardware with Floating-Point Numbers Approach is configured as such. The precision of the input could be

configured to 28 bits however, the IP core can only accept normalised size inputs and therefore 32 bits is selected. The output is customised for single-precision floating-point. Finally, upon a valid output from the conversion, the module outputs a concatenation of the keypoint coordinates, the float representation of sum of the squares of both patch's moments and the patch's moments, as seen below.

Listing 4.9: Output of the stage 1

```
assign out_stage1 = (ovalid) ? {'1b1, kp_coord, ftf_dout, m01, m10} : out_stage1;
```

Second Stage. In this stage is where both patch's moments are converted to floating-point representation and the square root is performed. For that, once again, two instances of the IP core are customised for fixed-to-float conversion and another for square root. As the values come from the buffer in a concatenation manner, it is necessary to assign the corresponding values to the correct inputs. As it can be seen in the Code snippet, the keypoint coordinates come first, then the sum of the squares of both patch's moments and then the patch's moments. Considering the keypoint coordinates have a total of 17 bits, the sum of the squares of both patch's moments is in the 32-bit single-precision representation and each patch's moment has 16 bits, the input for the square root are the 32 bits from 63:32 and the patch's moments can be attained from the bits 31:16 and 15:0.

Once again, upon valid output values of the three IP cores, the output of the module is the concatenation of the keypoint coordinates, bits 80:64 that come from the previous pipeline register, the result from the square root and the float representation of the patch's moments.

Considering the keypoint coordinates have a total of 17 bits, the sum of the squares of both patch's moments is in the 32-bit single-precision representation and each patch's moment has 16 bits, the input for the square root are the 32 bits from 63:32 and the patch's moments can be attained from the bits 31:16 and 15:0.

Listing 4.10: Output of stage 2

```
assign out_stage2 = (sqrt_ovalid & fm01_ovalid & fm10_ovalid) ?
{'1b1, pl_register[80:64],sqrt_dout, fm01, fm10} : out_stage2;
```

Third Stage. In this stage, the cosine and sine are computed and the keypoint coordinates are converted to the 32-bit single-precision representation. Once again, two instances of the IP core are configured for fixed-to-float conversion but, this time, each with the corresponding input precision since x and y have different sizes. Another two instances of the IP core are configured for the division operation.

Considering the order of the concatenation in the previous stage, x corresponds to bits 111:104 and y to bits 103:96 of the data coming from the previous pipeline register. For the cosine computation, as discussed before, the moment of the patch m_{10} , bits 31:0, is divided by the sum of the squares of both patch's moments, bits 95:64. Similarly, for the sine the m_{01} is used instead. The output requires a valid output from all the modules to pass the results to the next stage. This module passes on the sine, cosine and the coordinates converted to the 32-bit single-precision format.

Listing 4.11: Output of stage 3

```
assign out_stage3 = (div1_ovalid & div2_ovalid & x_ovalid & y_ovalid) ?
{'1'b1, sine, cosine, float_x, float_y} : out_stage3;
```

Fourth Stage. In this final stage, the values of the new keypoint coordinates x' and y' are computed and converted to fixed-point representation. For that, $xcos$, $ysin$, $ycos$ and $xcos$ are calculated recurring to the IP core configured as a multiplier. Then, new keypoint coordinates are obtained using the IP core as adder and subtractor following the equations stipulated earlier.

Finally, the new keypoint coordinates are converted to the fixed-point representation recurring to the same IP core.

Listing 4.12: Output of stage 4

```
assign out_stage4 = (dy_ovalid && dx_ovalid) ? {new_x, new_y} : out_stage4;
```

A utilisation report is made, and the main issue of this implementation is confirmed as a very large amount of board resources are used for this implementation, as it can be seen in the figure below.

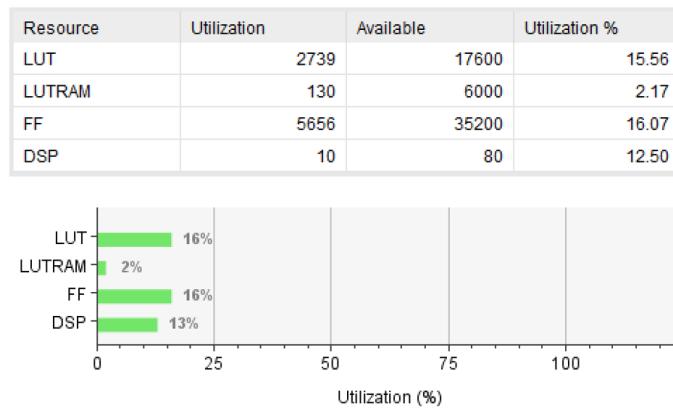


Figure 4.1.4: Utilisation report for Hardware with Floating-Point Numbers Approach

Hardware with Fixed-Point Numbers Approach. Unlike the floating-point approach, this approach is built as only one stage, so it can output a result as fast as possible.

The declaration of this module denotes that the module's interface is exactly how it was designed to be, receiving as inputs the keypoint coordinates and the patch's moments, while having only one output, the oriented keypoint coordinates.

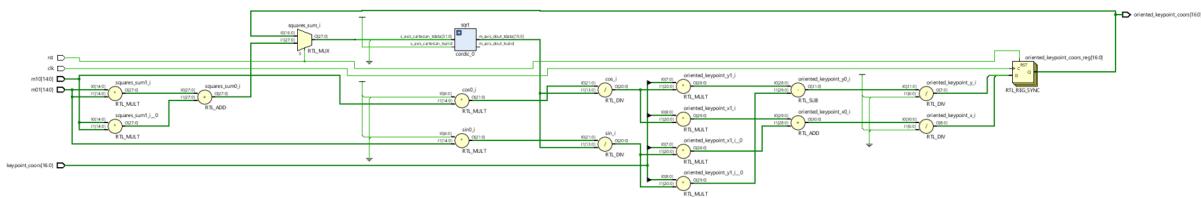


Figure 4.1.5: Hardware with fixed-point numbers approach module's declaration.

The declaration of this module denotes that the module's interface is exactly how it was designed to be, receiving as inputs the keypoint coordinates and the patch's moments, while having only one output, the oriented keypoint coordinates.

Listing 4.13: Hardware with fixed-point numbers approach module's declaration

```

1  module rotation_invariance(
2    input      clk,
3    input      rst,
4    input      [16:0] keypoint_coors,
5    input      [14:0] m01,
6    input      [14:0] m10,
7    output     reg [16:0] oriented_keypoint_coors
8  );

```

Reviewing the implementation and starting by the computation of the sum of the patch's moments' squares, the part common to every hardware approach, the calculation is made by multiplying the moments of the patch by themselves and then sum the results.

Listing 4.14: Sum of the patch's moments' squares computation

```

1  assign squares_sum = (rst) ? oriented_keypoint_coors :
2                                (m01*m01)+(m10*m10)      ;

```

To perform the square root of the squares_sum value, the Xilinx CORDIC IP is instantiated having the squares_sum as input, recurring to the template seen in the design section. The sin and cos are then calculated, having the numerator multiplied by a scalar N, as it was planned in the Analysis Phase. Since a sine or a cosine have a number between 0 and 1, these results sin and cos will have a maximum value N and the variables have a corresponding number of bits. Per example, if N is 128, the maximum value for sin and cos is 128 and thus they are required to be stored in 8-bit variables.

Listing 4.15: Sin and cos of the rotation angle computation

```

1  assign sin = (N*m01)/sqrt_squares_sum;
2  assign cos = (N*m10)/sqrt_squares_sum;

```

Finally, the oriented keypoint coordinates are calculated, and the scalar N is suppressed. The output, to carry the same format as the input, is the concatenation of the oriented keypoint's x with the oriented keypoint's y.

Listing 4.16: Oriented keypoint's coordinates computation

```

1  assign oriented_keypoint_x = (((keypoint_x*cos)+(keypoint_y*sin))/N);
2  assign oriented_keypoint_y = (((keypoint_y*cos)-(keypoint_x*sin))/N);
3
4  always@posedge clk
5  begin
6    if (rst) oriented_keypoint_coors <= 0;
7    else oriented_keypoint_coors <= {oriented_keypoint_x, oriented_keypoint_y};
8  end

```

The utilisation report for this approach reveals that the module implemented following this approach has a significant resources utilisation. However, the consumption is much less than the approach employing floating-point precision and might be more viable for that reason. Note that this report was made using a scalar of N=128 and the resources utilisation may slightly vary according to the stipulated value.

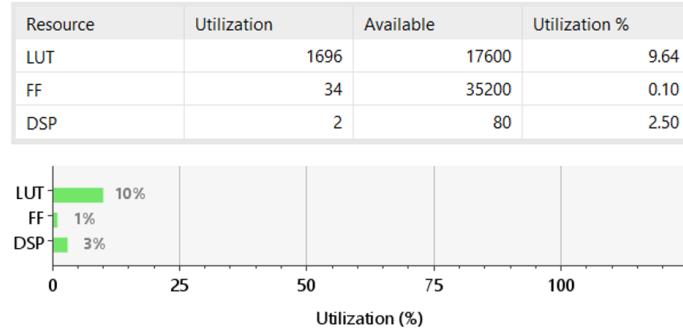


Figure 4.1.6: Utilisation report for Hardware with Fixed-Point Numbers Approach

4.1.6 Software Refactoring

Software Implementation. The functional implementation of the ORB detection was done. For this, the following ORB class was made.

Listing 4.17: ORB class

```

2  class ORB {
3
4      private:
5          int initial_position;
6          FAST_type fastN;
7          const int (*offsets)[2];
8          list<list < nms_data_t>> nms_horizontal;
9          list<list < nms_data_t>> nms_vertical;
10         double threshold;
11
12         void organize(list<nms_data_t> &line);
13         int score(int dark[], int bright[]);
14         void nonMax();
15         int contiguity;
16
17         public:
18             ORB(FAST_type FAST);
19             ~ORB();
20             list<point_t> detection(cv::Mat img , double threshold);
21
22     };

```

In the detection, the first thing to be done is to classify each pixel as dark or bright, according to the bresenham circle of each type of Fast.

Listing 4.18: Bresenham circle for all fasts

```

static const int offsets16[][2] =
{0, 3}, {1, 3}, {2, 2}, {3, 1}, {3, 0}, {3, -1}, {2, -2}, {1, -3}, {0, -3}, {-1, -3}, {-2, -2}, {-3, -1}, {-3, 0}, {-3, 1},
{-2, 2}, {-1, 3}};

static const int offsets12[][2] =
{
    {0, 2}, {1, 2}, {2, 1}, {2, 0}, {2, -1}, {1, -2}, {0, -2}, {-1, -2}, {-2, -1}, {-2, 0}, {-2, 1}, {-1, 2}};

```

```

7   static const int offsets8[][][2] =
{
    {0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}};

```

At the same time, it will be checked if the pixel is a possible keypoint. For this, the number of ones in a row in the dark or bright vector must be equal to, or greater than the number required for each type of Fast. (ex FAST12 needs 12 pixels with the number of ones in a row). That said, it has to be done the contiguity test. If the contiguity is verified the pixel is a possible keypoint and it is inserted in the listPossible_keypoint. All the calculations necessary to obtain the oriented coordinates are also performed. For a better data re-ordering to perform the NMS, whenever we change lines in the image we insert the list of possible keypoints in the list of nms_horizontal lists.

After the image analysis is finished, NMS is performed. In the NMS it will be evaluated which keypoints are possible to suppress, setting the Boolean variable to true for that keypoint. After the analysis is completed it will be suppressed the keypoints that have the Boolean variable at TRUE. After that, the list of keypoints lists will be arranged vertically and the same process will be done as it was made horizontally.

Finally, all keypoints presented at the end of the NMS are placed in a list of keypoints.

Listing 4.19: detect function

```

for (line = initial_position; line <= (rows - initial_position); line++)
{
    for (column = initial_position; column <= (cols - initial_position); column++)
    {
        int centralPix= img.at<uchar>(line, column);
        m10 += (line+1) * centralPix;
        m01 += (column+1) * centralPix;
        /*      goes around the Bresenham circle and classifies the pixel */

        for (position = 0; position < fastN; position++)
        {
            x = (offsets[position][0]);
            y = (offsets[position][1]);
            int pixelToCompare = img.at<uchar>(line + x , column + y);
            if (centralPix - pixelToCompare - threshold > 0)
            {
                is_bright[position] = 1;
                is_dark[position] = 0;
            }
            else if (pixelToCompare - centralPix - threshold > 0)
            {
                is_dark[position] = 1;
                is_bright[position] = 0;
            }
            else
            {
                is_bright[position] = 0;
                is_dark[position] = 0;
            }
        }
        m10 += (line+1) * centralPix;
        m01 += (column+1) * centralPix;
        scr = score(is_dark, is_bright);
        int is_corner = scr >= contiguity ? 1 : 0;
        if (is_corner == 1)
        {
            //insert on the list possible_keypoint
        }
    }
}

```

```

        }
        if (!potentialKeypoints.empty()){
            /*insert on the list of list (nms_horizontal) */
        }
    }
    nonMax();
    list<point_t> keypointsList;

    for (auto it = nms_vertical.begin(); it != nms_vertical.end(); ++it) {
        for (auto it2 = it->begin(); it2 != it->end(); ++it2){
            n_x= (it2->coord.x * cos) + (it2->coord.y * sin);
            n_y= (it2->coord.y * cos) - (it2->coord.x * sin);
            /*calculate the oriented coordinates and place in the list keypointsList*/
        }
    }
    return keypointsList;
}

```

Listing 4.20: score function

```

int ORB::score(int dark[], int bright[])
{
    for (i = 0; i < max; i++)
    {
        index = (i >= (fastN)) ? i - (fastN) : i;
        if (dark[index] == 1)
        {
            d_count++;
            maxA = (d_count > maxA) ? d_count : maxA;
        }
        else
        {
            d_count = 0;
        }
        if (bright[index] == 1)
        {
            b_count++;
            maxB = (b_count > maxB) ? b_count : maxB;
        }
        else
        {
            b_count = 0;
        }
    }
    score = maxA >= maxB ? maxA : maxB;
}
return score;
}

```

4.2 BRISK

4.2.1 Image Slicing and BRAM

The BRAM channel's functionality is divided into:

- An input: for writing by the BRAM controller;
- An output: for reading by the detection stage.

Listing 4.21: Image Slice Module: Output Forwarding and BRAM Generation

```

module image_slice
#(
  parameter RAM_DEPTH = 8,          /* Number of lines */
  parameter ADDRESS_WRITE_BLOCKS = 8,    /* AUTO number of blocks for write channels */
  parameter ADDRESS_READ_BLOCKS = 8,    /* AUTO number of blocks for read channels */
  parameter ADDRESS_WRITE_BLOCK_SIZE = 4, /* AUTO number of bytes on a write for write channels */
  parameter ADDRESS_READ_BLOCK_SIZE = 4, /* AUTO number of bytes on a read for read channels */
  parameter DATA_WRITE_DEPTH = 1,      /* Number of blocks to insert (vertically) */
  parameter DATA_READ_DEPTH = 7       /* Number of blocks to fetch (vertically) */
) (
  input  clk,
  input  write_en,
  13   input [${clog2(RAM_DEPTH)-1:0}]      line_addr_write,
  input [${clog2(ADDRESS_WRITE_BLOCKS)-1:0}] block_addr_write,
  input [${clog2(RAM_DEPTH)-1:0}]      line_addr_read,
  input [${clog2(ADDRESS_READ_BLOCKS)-1:0}] block_addr_read,
  input [ADDRESS_WRITE_BLOCK_SIZE*8*DATA_WRITE_DEPTH-1:0] data_write,
  18   output [ADDRESS_READ_BLOCK_SIZE*8*DATA_READ_DEPTH-1:0] data_read
);

```

This is done because the port dedicated to the detection needs to be available for reading at every clock cycle. In the future, measures could be taken to allow for only one channel or true dual channel with double the throughput, but a better controller and a good arbitration system should be implemented along with it.

Moreover, one has two inputs clocks, since the BRAM is single dual-port, which means that there is a port for writing and another port for reading, as forementioned, needing one clock for each port. Both ports are always enabled, as specified by the ena and enb wires, and the writing in memory is done through the A channel and the reading through the B channel.

BRAM Output. Each line has its output, and all of them are enabled. However, only the selected lines will be forwarded to the output, as one can see in fig. 4.2.1 and listing 4.22.

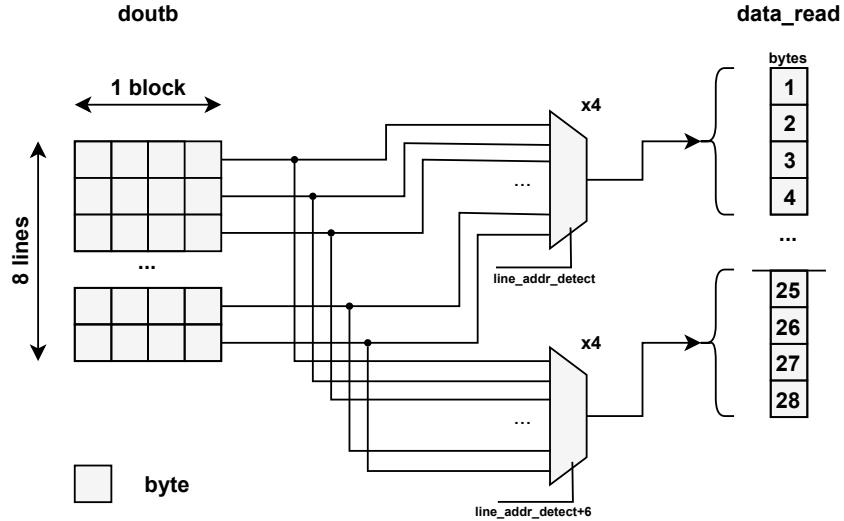


Figure 4.2.1: BRAM Output Forwarding

Listing 4.22: Image Slice Module: Output Forwarding and BRAM Generation

```

1   wire [ADDRESS_READ_BLOCK_SIZE*8-1:0]      doutb[RAM_DEPTH-1:0];
...
/* Generate RAM blocks */
generate
  genvar i_lines;
  for (i_lines=0; i_lines<RAM_DEPTH; i_lines=i_lines+1)
    begin : lines
      /* Only the selected line is write-enabled */
      wire chosen_for_write = line_addr_write == i_lines ? wea : 1'bo;
11
      image_slice_ram line(
        .clk_a(clk),
        .ena(ena),
        .wea(chosen_for_write), /* Only the selected line is write-enabled */
16      .addr_a(block_addr_write), /* Selected block for writing*/
        .dina(dina), /* All lines have the same input */
        .clk_b(clk),
        .enb(enb),
        .addr_b(block_addr_read), /* Selected block for reading*/
        .doutb(doutb[i_lines]) /* Forwarded to the selectors */
      );
    end
  endgenerate

```

Dynamic Mapping. In terms of the dynamic mapping referred to in section 3.3.2, its implementation can be seen in listing 4.23 and its simulation can be observed in fig. 5.2.23.

Listing 4.23: Image Slice Module: Dynamic Mapping

```

...
/* Vertical address mapping (interface to RAM) - detector */
generate
  genvar i_map_read;
  for (i_map_read=3'be; i_map_read<DATA_READ_DEPTH; i_map_read=i_map_read+3'b1)
    begin : ram_to_output
      wire [$clog2(RAM_DEPTH)-1:0] dynamic_address = line_addr_read+i_map_read;
      assign data_read[i_map_read*ADDRESS_READ_BLOCK_SIZE*8 +: ADDRESS_READ_BLOCK_SIZE*8] = doutb[dynamic_address];
    end
  endgenerate
10

```

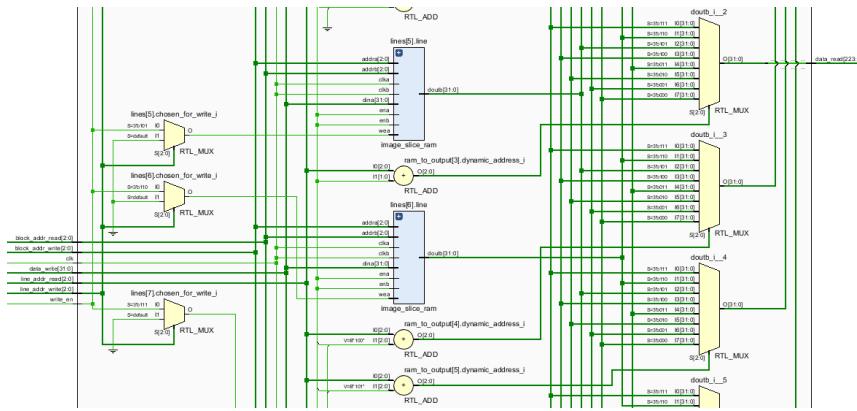


Figure 4.2.2: BRISK Image Slice RTL Schematic

4.2.2 Bright/Dark Classifier

The classifier schematic of fig. 4.2.3 follows what was stipulated in the classifier design of section 3.3.4. Although, note that some intermediate registers where added every two operations to ensure data validity at the output of the classifier.

Additionally, some attention was needed to guarantee that the intermediate results where correct, using 10 bits to represent the operation's results assured that the two subtractions' results where correctly interpreted when the outcome was a negative value.

The change between the different types of FAST (fig. 2.1.14) only influences the number of dark and bright classification blocks and the size of the `is_bright` and `is_dark` arrays, this fact eases code portability between FAST types.

Listing 4.24: Bright/Dark Classifier Interface

```

module classifier
#(
  parameter CIRCLE_SIZE = 16,
  parameter WIDTH = CIRCLE_SIZE
5
)(

  input i_clk;
  input i_rst;
  input i_clk_en;

```

```

10      input [7:0]      i_center_pixel;           // Center pixel of Bresenham Circle
11      input [7:0]      i_threshold;             //
12      input [CIRCLE_SIZE*8-1:0]  i_bresenham_circle; // Bresenham Circle
13
14      output [CIRCLE_SIZE-1:0]   o_is_bright;        //
15      output [CIRCLE_SIZE-1:0]   o_is_dark;          //
16      output [CIRCLE_SIZE*8-1:0] o_score_br;         // score of all pixels concatenated
17      output [CIRCLE_SIZE*8-1:0] o_score_dk;         //
18  );

```

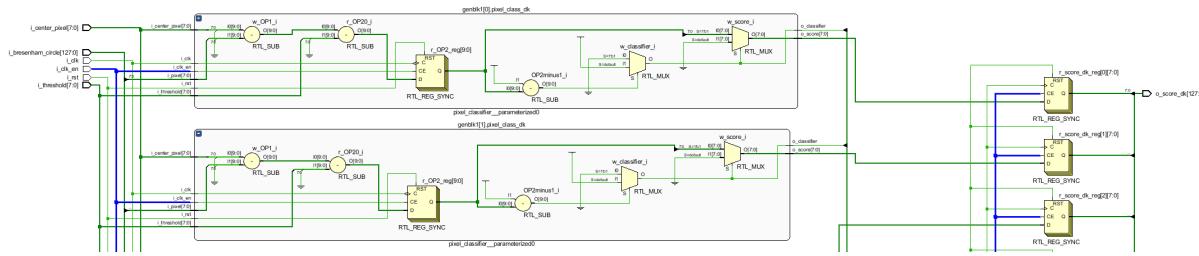


Figure 4.2.3: Classifier Module: Schematic (FAST-12)

Classification Block Operations. One created the classification blocks within the classifier module, making sure each one of them performs the operations stipulated in section 3.3.4, as represented in listing 4.25.

Listing 4.25: Classifier Module: Generated Block

```

...
2   generate
3     if (p_bright == 1'bo)
4       assign w_OP1 = {2'b00, i_center_pixel} - {2'b00, i_pixel};
5     else
6       assign w_OP1 = {2'b00, i_pixel} - {2'b00, i_center_pixel};
7   endgenerate
8
9   ...
10  always @ (posedge i_clk) begin
11
12    if(i_RST)
13      r_OP2 <= 10'b0;
14    else if (i_CLK_EN)
15      r_OP2 <= w_OP1 - {2'b00, i_THRESHOLD};
16
17  end
18 endmodule

```

4.2.3 Detection Controller and Register File

In order to control the various parts of the detection stage independently, a scheme with different control and clock signals was devised. In order to have modules working at the correct pace with the best energy efficiency practicable, some clocks were gated. Listing 4.26

Listing 4.26: Detection Controller Interface

```

module classifier_controller
#(

```

```

4      parameter RAM_DEPTH = 8,           /* Number of lines */
5      parameter ADDRESS_FETCH_BLOCKS = 8, /* AUTO number of blocks for detect channels */
6      parameter ADDRESS_FETCH_BLOCK_SIZE = 4, /* AUTO number of bytes on a detect for read channels */
7      parameter DATA_FETCH_DEPTH = 7,       /* Number of blocks to fetch (vertically) */
8      parameter COUNTER_SIZE_X = 9,         /* Size of x counter */
9      parameter COUNTER_SIZE_Y = 5,         /* Size of y counter */
10     parameter IMAGE_WIDTH = 32,          /* Total image height */
11     parameter IMAGE_HEIGHT = 8,          /* Total image width */
12     parameter NMS_DELAY = 5             /* Delay pixels take to reach NMS */
13 );
14
15     /* RAM interface */
16     output [${clog2(RAM_DEPTH)-1:0}] line_addr_fetch, /* Pixel fetch line address */
17     output [${clog2(ADDRESS_FETCH_BLOCKS)-1:0}] block_addr_fetch, /* Pixel fetch block address */
18     /* Memory Access Controller */
19     output [COUNTER_SIZE_Y-1:0] o_y_cnt, /* Line count */
20     /* Register File interface */
21     output o_rf_en_load, /* Register file load operation control */
22     /* NMS interface */
23     output [COUNTER_SIZE_X-1:0] o_x_cnt_nms, /* Column count with NMS_DELAY */
24     output [COUNTER_SIZE_Y-1:0] o_y_cnt_nms, /* Line count with NMS_DELAY */
25     output o_valid_score, /* Validity of current circle */
26
27     /* Global space interface */
28     input clk_en, /* Global enable signal */
29     input clk, /* Global clock signal */
30     input reset /* Global reset signal */
31 );

```

Counters. The logic behind the manipulation of the counters for the coordinates of the scan is presented in listing 4.27. All the logic for it is established through named wires at their input and they are updated at the negative edge of the clock, since the Image Slice module operates at the positive edge, reducing problems with synchronization.

Listing 4.27: Detection Controller - Counters

```

2      reg [COUNTER_SIZE_X-1:0] x_cnt; /* x_cnt and y_cnt allow us to know the coordinates of the keypoints, ... */
3      reg [COUNTER_SIZE_Y-1:0] y_cnt; /* ... what block to fetch next and what line(s) to load into the slice */

4      wire [COUNTER_SIZE_X-1:0] w_x_cnt = x_cnt == 'X_LIMIT ? 'X_RESET : /* Line wrap */
5                                x_cnt - 4'd1; /* Decrement */
6      wire [COUNTER_SIZE_Y-1:0] w_y_cnt = x_cnt == 'X_LIMIT ?
7                                y_cnt == 'Y_LIMIT ? 'Y_RESET : /* End of image */
8                                y_cnt + 4'd1 : /* Increment on line wrap */
9                                y_cnt; /* Keep value */

```

Register File. Enabling or disabling the loading of new information in the Register File is controlled through `rf_en_load`, following what was defined in the design stage in section 3.3.3. It is, too, updated at the negative edge of the clock, after reading from memory (third edge of the cycle), since the PISO cells in the Register File are updated at the positive edge.

Listing 4.28: Detection Controller - Register File

```

1      /* Register file load operation control */
2      reg rf_en_load;

3      /* Load enable */
4      wire w_rf_en_load = global_reset ? 1'd0 : /* Reset response */
5                                clk_count_cycle == 2'b01; /* Enable load when clk_count_cycle is equal to 1 */

```

NMS Interface and Flow Control. The logic used to calculate the coordinates of the circle corresponding to the results from the scoring stage at the input of the NMS module are presented in listing 4.29.

Listing 4.29: Detection Controller - Non-Max Suppression interface

```

4      /* Start up flag */
5      wire startup = (x_cnt > 'X_RESET - 10 - NMS_DELAY) && y_cnt == 0;    /* Clock gating for the classifier clock */
6      wire line_change = (x_cnt < 'X_RESET - 7) &&
7          (x_cnt > 'X_RESET - 7 - DATA_FETCH_DEPTH);                      /* ... */
8
9
10     /* NMS line and column count */
11     wire x_past_delay =
12         x_cnt <= 'X_RESET - NMS_DELAY - 10;
13     assign o_x_cnt_nms =
14         x_past_delay ? x_cnt + NMS_DELAY + DATA_FETCH_DEPTH[2:1];
15         x_cnt + NMS_DELAY - 'X_RESET + DATA_FETCH_DEPTH[2:1];
16     assign o_y_cnt_nms =
17         x_past_delay ? y_cnt + DATA_FETCH_DEPTH[2:1];
18         y_cnt - 1 + DATA_FETCH_DEPTH[2:1];
19     assign o_valid_score =
20         ~startup & ~line_change;

```

4.2.4 Detection Integration and IP Packaging

Having implemented all modules of the BRISK classification substage all that is left to do is to integrate all the modules into a unified mechanism, as represented in the RTL schematic of the full integration fig. 4.2.4.

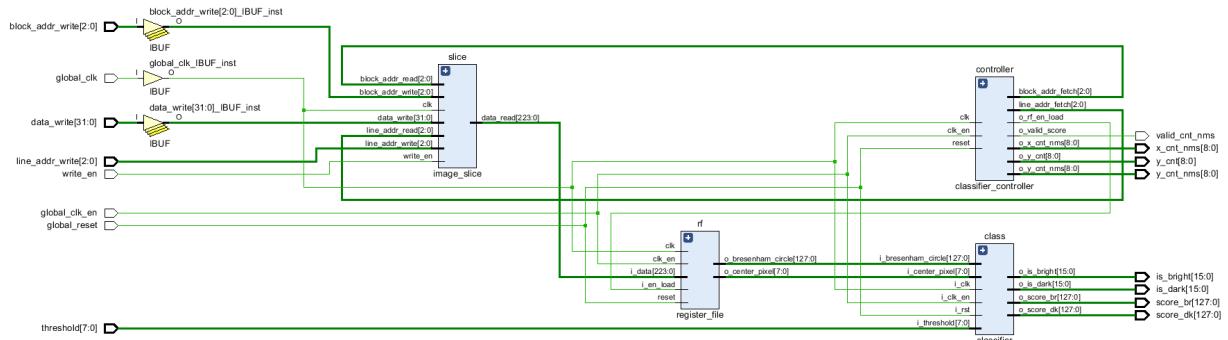


Figure 4.2.4: BRISK Classification RTL Schematic

After the implementation of the accelerator is done, one should create IPs for the latter. The fig. 4.2.5, illustrates the flow in the IP packager and its usage model. With the Vivado IP packager an IP developer can create and package files and associated data in an IP-XACT format, and add the IP to the Vivado catalogue. After the IPs are distributed, an end-user can customise them and use them in their designs [24]. The fig. 4.2.6, portraits the IP integration of the detection project's modules.

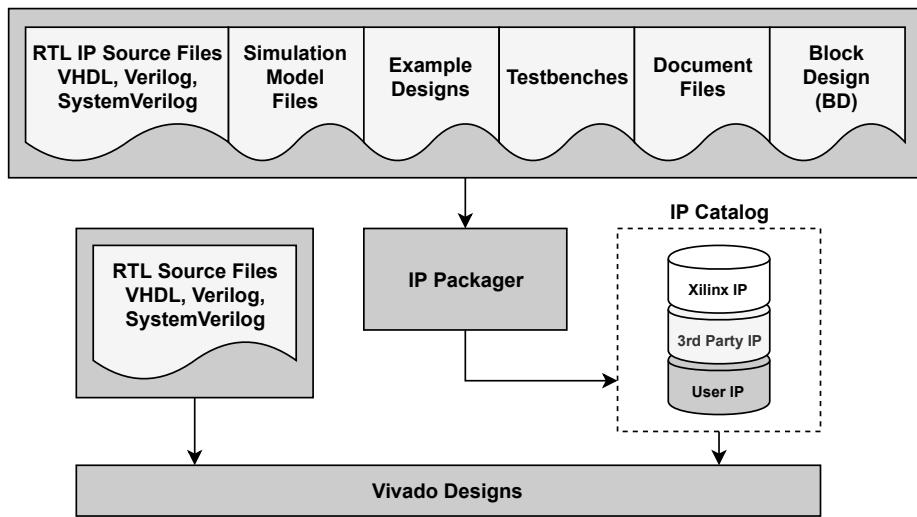


Figure 4.2.5: IP Packaging and Usage Flow

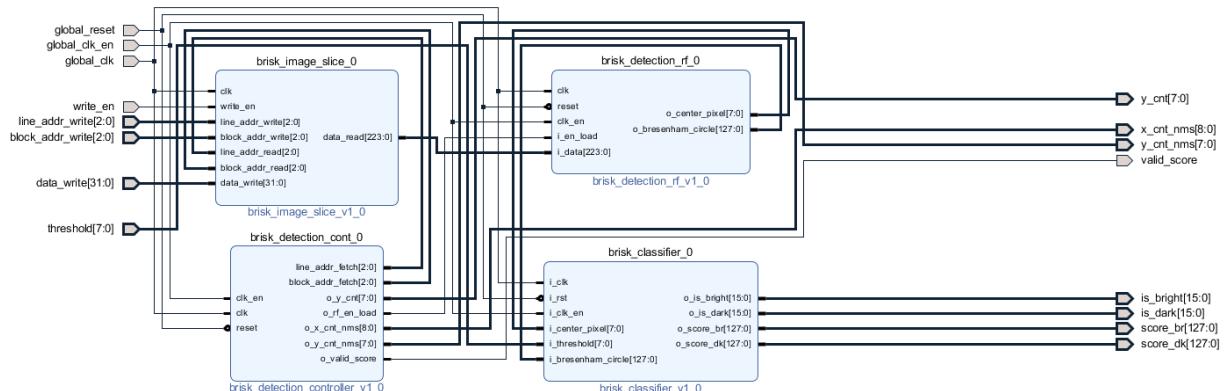


Figure 4.2.6: BRISK Classification Block Design

4.2.5 Score Computation

For different values of N, the modules remain the same, only the sizes of the input and output arrays change. With N equal to 5, the input consists of 64 bits of darker and brighter pixels, in case it is equal to 9, the input arrays have 96 bits and in N equal to 12 they have 128 bits. The output size only differs on fast 5, as can be observed in 4.30, on 9 and on 12 it is equal to 12.

Listing 4.30: FAST-5 Module

```
module score_fast5(
    input i_clk,
    input i_rst,
    input i_clk_en,
    input [63:0] i_score_br,
    input [63:0] i_score_dk,
    output [10:0] o_score
);
```

Listing 4.31: FAST-9 Module

```
module score_fast9(
    input i_clk,
    input i_rst,
    input i_clk_en,
    input [95:0] i_score_br,
    input [95:0] i_score_dk,
    output [11:0] o_score
);
```

Listing 4.32: FAST-12 Module

```
module score_fast12(
    input i_clk,
    input i_rst,
    input i_clk_en,
    input [127:0] i_score_br,
    input [127:0] i_score_dk,
    output [11:0] o_score
);
```

Following the method defined in the design, of performing successive sums of the pixels, it is possible to verify that the design schematic is properly generated for the three different N(4.2.7, 4.2.8, 4.2.9).

This starts by split the sets of incoming arrays so that they are subsequently summed, until the largest value of the sums is obtained, between the dark and bright pixels.

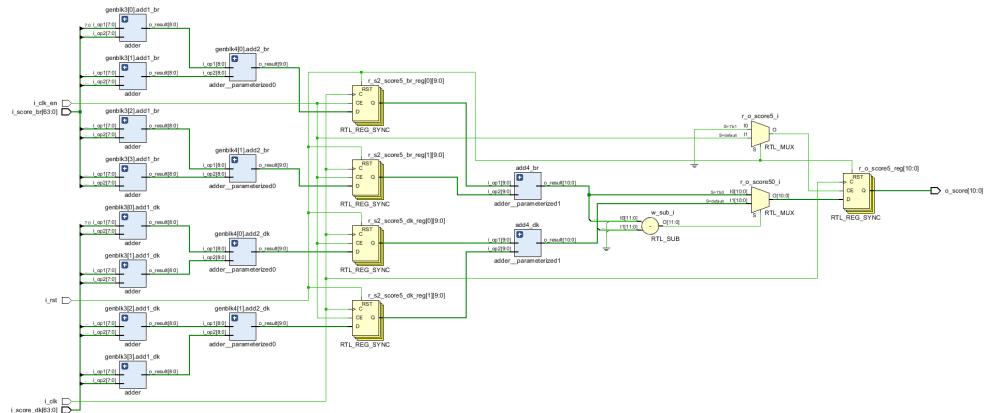


Figure 4.2.7: Score Schematic for FAST-5 Module (Augmented in fig.D.3)

In the case of FAST-5, there are 3 adders chains, the first with four 8bit adders, the second with two 9bit adders and the last with one 10bit adder, for the dark and bright ones.

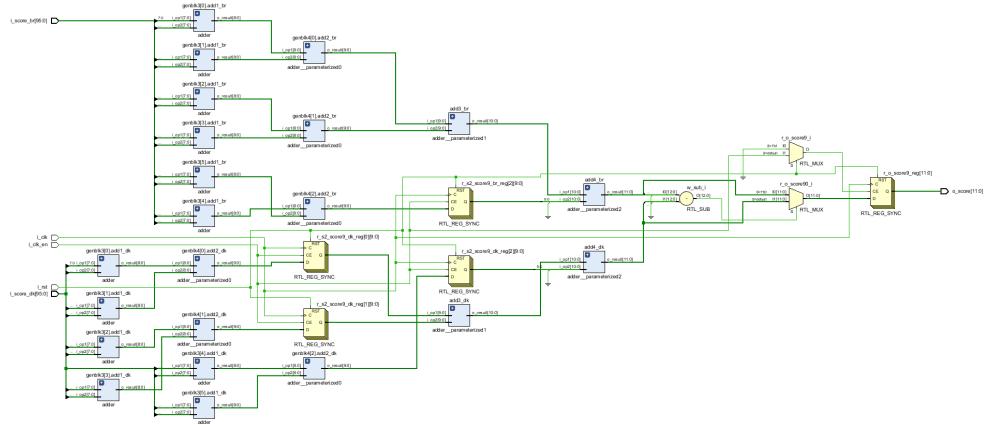


Figure 4.2.8: Score Schematic for FAST-9 Module (Augmented in fig.D.4)

For FAST-9, there are 4 chains of adders, the first with six 8bit adders, the second with three 9bit adders and, as the number of arrays is odd, the third with only one 10bit adder and the last with one 11bit adder, for the dark and bright ones. In this case, it is also necessary to add a bit to the result of the second phase (fig.D.2) so that in the last one it can be added with an 11bit adder.

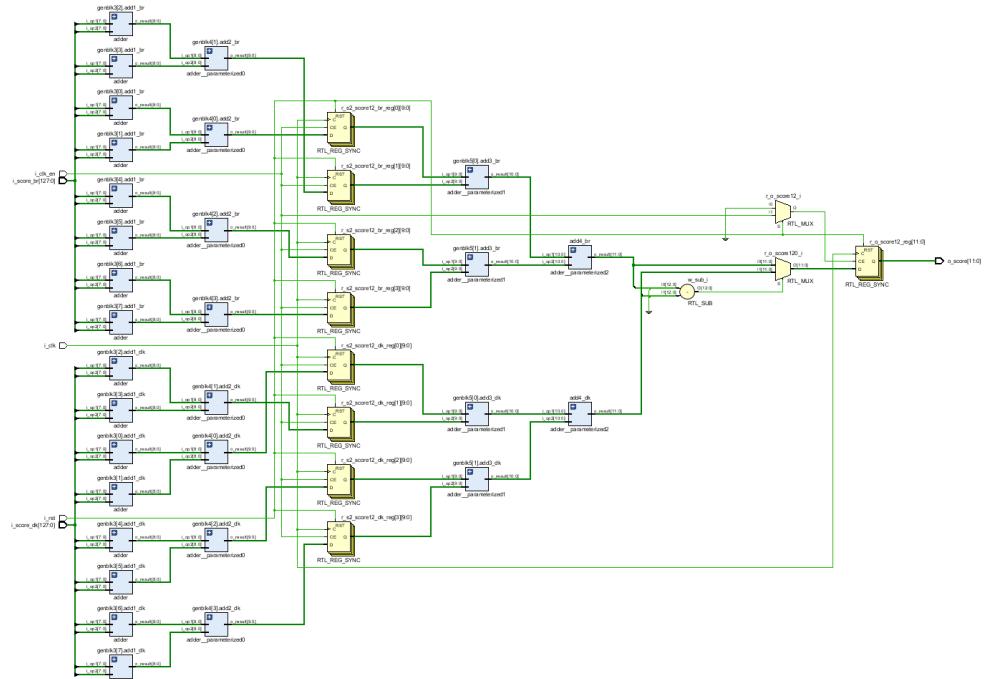


Figure 4.2.9: Score Schematic for FAST-12 Module (Augmented in fig.D.5)

FAST-12 has the same types of adders as FAST-9, but in different quantities, with the first chain having eight, the second four, the third two and finally, only one. Listing 4.24: Transformation arrays for all the fasts In the listing 4.33, it is possible to observe the generation of the first chain of six adders, from a for loop, where the entries have the index i^2

so that they have the proper size for the FAST in question, and this example belongs to FAST-9. The same applies to the index of the output arrays, which will have a maximum size equal to the division by two of the number of adders of the first stage. Each of these cases is replicated for the darkest and the brightest pixels.

Listing 4.33: Score Adders for FAST-9 Module

```

1   generate
2     for (i=0; i < 'd6 ; i=i+1)
3       begin
4         adder  add1_br(
5           .i_op1(w_score_br[i*2]),
6           .i_op2(w_score_br[(i*2)+1]),
7           .o_result(w_s1_score_br[i])
8         );
9         adder  add1_dk(
10           .i_op1(w_score_dk[i*2]),
11           .i_op2(w_score_dk[(i*2)+1]),
12           .o_result(w_s1_score_dk[i])
13         );
14       end
15     endgenerate

```

The variable *w_sub* receives the value of the subtraction between the score of the brightest and darkest pixels, in order to analyze their sign bit. In case the sign bit is null, the most score corresponds to the bright pixels, otherwise it is the darkest pixels.

In all three modules, intermediate registers were implemented, after the second stage of adders and at the end of all sums, to ensure the validity of the data in the course of operations, until the module output. This transition between registers, every clock positive edge, can be observed in listing 4.34.

Listing 4.34: Score Assignment and Transition Between Registers

```

assign w_sub = {1'b0, w_sum_score_br} - {1'b0, w_sum_score_dk};

assign o_score = r_o_score9;

5  always @ (posedge i_clk) begin
6    if(i_rst) begin
7      for (j = 0; j <= 2; j=j+1) begin
8        r_s2_score9_br[j] <= 10'b0;
9        r_s2_score9_dk[j] <= 10'b0;
10      end
11      r_o_score9 <= 12'b0;
12    end
13    else if(i_clk_en) begin
14      for (j = 0; j <= 2; j=j+1) begin
15        r_s2_score9_br[j] <= w_s2_score_br[j];
16        r_s2_score9_dk[j] <= w_s2_score_dk[j];
17      end
18      r_o_score9 = (w_sub[12] == 1'b0) ? w_sum_score_br : w_sum_score_dk;
19    end
20  end

```

This code block works for the other different types of FAST as well, adjusting only the number of adders to be generated.

4.2.6 Contiguity Test

As previously established in the design phase, each type of FAST (FAST-5, FAST-9, and FAST-12) will have its own module. Each module has two vectors as input containing information about the number of pixels brighter or darker than the center one and as output the pixel classification with logical value '1' if it is keypoint or '0' otherwise. This is shown in the snippets of code below.

Listing 4.35: FAST-5 Module

```

1  module contig_fast5(
2    input i_clk,
    input i_rst,
    input i_clk_en,
    input [4:0] i_is_bright,
    input [4:0] i_is_dark,
    output o_is_corner
);

```

Listing 4.36: FAST-9 Module

```

2  module contig_fast9(
3    input i_clk,
    input i_rst,
    input i_clk_en,
    input [8:0] i_is_bright,
    input [8:0] i_is_dark,
    output o_is_corner
);

```

Listing 4.37: FAST-12 Module

```

6  module contig_fast12(
7    input i_clk,
    input i_rst,
    input i_clk_en,
    input [15:0] i_is_bright,
    input [15:0] i_is_dark,
    output o_is_corner
);

```

For the contiguity test itself, in order to be able to compare all possible states, successive shifts of a predefined array are made and compared with the input array containing the classification of the pixels of the circle as darker or brighter than the centre pixel. In this way, it is possible to test all possible states to determine the contiguity to be considered a keypoint or not.

The following is an example of how the contiguity test is done. The example is given with FAST-5 using the vector *is_bright* with pixel classification but this applies to all types of FAST. A mask is applied to the *is_bright* vector in order to keep only the bits with logical value '1' that are contiguous. Then it is compared with the possible contiguous states that can exist. In the example of figure 4.2.10 the first state is not contiguous, so it is not considered a keypoint.



Figure 4.2.10: Example of a Contiguity Test(1)

To compare it with all possible states, a shift is made to the mask that is applied to the vector *is_bright*. Thus it is possible to compare with the possible states in which these will vary in number depending on the type of FAST.

Note that this is also done for the *is_dark* vector and that it is done in parallel to make good use of the resources in order to be as fast as possible.



Figure 4.2.11: Example of a Contiguity Test(2)

Following the FAST-5 example mentioned above, in the listing below 4.38 one can see all the shifts and also the comparisons needed to be able to test the contiguity of the *is_bright* and *is_dark* arrays. If any of the conditions are true, the *r_2r* takes on the logical value '1', thus signalling a keypoint.

Listing 4.38: Contiguity Test for FAST-5 Module

```

2      assign r_2r = ((i_is_bright & fast5)
3          & (i_is_dark & fast5)
4          & (i_is_bright & {fast5[0],fast5[7:1]}) == {fast5[0],fast5[7:1]) ? 1:
5          & (i_is_dark & {fast5[0],fast5[7:1]}) == {fast5[0],fast5[7:1]) ? 1:
6          & (i_is_bright & {fast5[1:0],fast5[7:2]}) == {fast5[1:0],fast5[7:2]) ? 1:
7          & (i_is_dark & {fast5[1:0],fast5[7:2]}) == {fast5[1:0],fast5[7:2]) ? 1:
8          & (i_is_bright & {fast5[2:0],fast5[7:3]}) == {fast5[2:0],fast5[7:3]) ? 1:
9          & (i_is_dark & {fast5[2:0],fast5[7:3]}) == {fast5[2:0],fast5[7:3]) ? 1:
10         & (i_is_bright & {fast5[3:0],fast5[7:4]}) == {fast5[3:0],fast5[7:4]) ? 1:
11         & (i_is_dark & {fast5[3:0],fast5[7:4]}) == {fast5[3:0],fast5[7:4]) ? 1:
12         & (i_is_bright & {fast5[4:0],fast5[7:5]}) == {fast5[4:0],fast5[7:5]) ? 1:
13         & (i_is_dark & {fast5[4:0],fast5[7:5]}) == {fast5[4:0],fast5[7:5]) ? 1:
14         & (i_is_bright & {fast5[5:0],fast5[7:6]}) == {fast5[5:0],fast5[7:6]) ? 1:
15         & (i_is_dark & {fast5[5:0],fast5[7:6]}) == {fast5[5:0],fast5[7:6]) ? 1:
16         & (i_is_bright & {fast5[6:0],fast5[7]}) == {fast5[6:0],fast5[7]) ? 1:
17         & (i_is_dark & {fast5[6:0],fast5[7]}) == {fast5[6:0],fast5[7]) ? 1:
18
19      o;

```

As the contiguity test and the scoring are done in parallel, it is of utmost importance that the contiguity test result is obtained at the same time as the scoring result. For this, it was necessary to delay the test using two registers where the values are passed from one to the other and the result is obtained in the next clock cycle. Below one can check the listing 4.39 that performs the delay.

Again, this is done for the three contiguity test modules where two registers are always used for the delay.

Listing 4.39: Transition Between Registers

```

...
reg r_10_is_corner;
reg r_20_is_corner;
...
assign o_is_corner = r_20_is_corner;

always @ (posedge i_clk) begin
    if(i_rst) begin
        r_10_is_corner <= 0;
        r_20_is_corner <= 0;
    end
    else if(i_clk_en) begin
        r_10_is_corner <= r_2r;
        r_20_is_corner <= r_10_is_corner;
    end
end

```

4.2.7 2D Non-maximum Suppression

Two dimensions' non-maximum-suppression implementation is divided into three units. They are all instantiated in a single module whose operation consists of connecting all the inputs and outputs among the different units in order to guarantee the stage's compression.

From a global perspective, the module inputs are the clock and reset signals (to aid synchronization), read_en flag to indicate whether the inputs are valid or not, full flag to stop the procedure when the FIFO gets full, and most important the concatenated pixel information. The outputs are the keypoint coordinates, a write_en flag which allows us to write it on the FIFO, and a wire to inform the next stage when we finish the image analysis.

Listing 4.46 illustrates an include file where the image parameters, as well as the pixels', are defined and where we can easily adapt to different dimensions. It is quite helpful either in loop definitions to go through all image rows and columns, and to the input pixel decoding. On the other side, it increases the code readability.

Listing 4.40: Defines.

```
//image definition
2   'define X_MAX          320
3   'define Y_MAX          240

//pixel information decode
4   'define X_BEGIN        29
5   'define X_END          21
6   'define Y_BEGIN        20
7   'define Y_END          13
8   'define IS_CORNER      12
9   'define SCR_BEGIN      11
10  'define SCR_END        0
```

Listing 4.41: Module declaration.

```
//nms 2dimensions
3   module nms_2d(
4     input  clock,
5     input  rst,
6     input  read_en,
7     input  [29:0] pixel,
8     input  full,
9     output [16:0] coordinates,
10    output write_en,
11    output complete
12  );
```

Listings 4.42, 4.43, 4.44 show the three modules instantiation where we can visualize each one inputs and outputs:

Listing 4.42: Horizontal inst.

```
2   nms_horizontal horizontal(
3     .clk(clock),
4     .rst(rst),
5     .read_en(read_en),
6     .pixel(pixel),
7     .fifo_full(full),
8     .output_corner(out_pixel),
9     .last_pixel(last)
10    );
```

Listing 4.43: Transition inst.

```
5   horizontal2vertical transition(
6     .clk(clock),
7     .rst(rst),
8     .last_pixel(last),
9     .pixel(out_pixel),
10    .above_corner(c1),
11    .center_corner(c2),
12    .below_corner(c3),
13    .finish(complete)
14  );
```

Listing 4.44: Vertical inst.

```
5   nms_vertical vertical(
6     .clk(clock),
7     .rst(rst),
8     .above_corner(c1),
9     .center_corner(c2),
10    .below_corner(c3),
11    .output_corner(coordinates),
12    .write_en(write_en)
13  );
```

Once compiled and generated the elaborated design, the modules block accquires the following aspect:

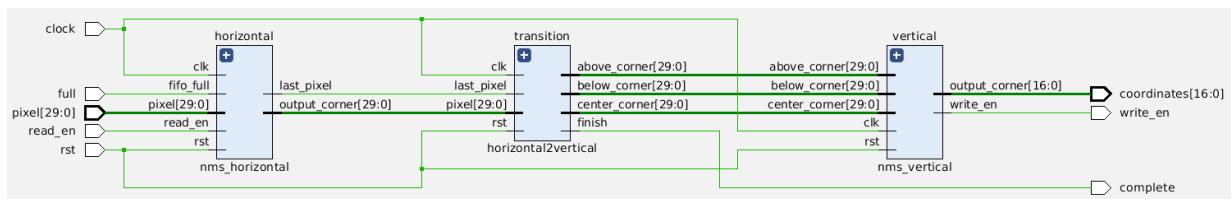


Figure 4.2.12: Generic schematic.

Horizontal The horizontal stage includes the filter logic, but the most critical and challenging part is the implementation of the state machine designed in figure 3.3.18. Every time there is a positive edge clock, the machine's state will be checked and adjusted according to auxiliary parameters whose function is to help to situate the pixel in the respective line. After this switch case, the state's outcome interferes with the filter's logic as a restriction of the neighborhood verification.

Listing 4.45: State machine implementation - part I.

```

case (state)
    begin_l :
        begin
            count <= count + 1;
            if (new_line && (count == 1))
                begin
                    state <= begin_l;
                    count <= 1;
                    forward <= 1;
                end
            else if (new_line && (count == 2))
                begin
                    forward <= 0;
                    state <= end_l;
                end
            else if(count == 2)
                begin
                    forward <= 0;
                ...

```

Listing 4.46: State machine implementation - part II.

```

state <= middle_l;
end
end
middle_l :
begin
    if (new_line)
        state <= end_l;
    else
        state <= middle_l;
end
end_l :
begin
    count <= 1;
    state <= begin_l;
    if(new_line)
        forward <= 1;
end
default :
state <= begin_l;
endcase

```

The module's code results in the figure 4.2.13 design. Through its observation, it is easier to understand the whole logic behind the algorithm. At the first look, we can perceive the pixel as an input through which the state machine works. Its entire logic derives as a state register value, which is an input of the filter logic and where the corners are effectively compared.

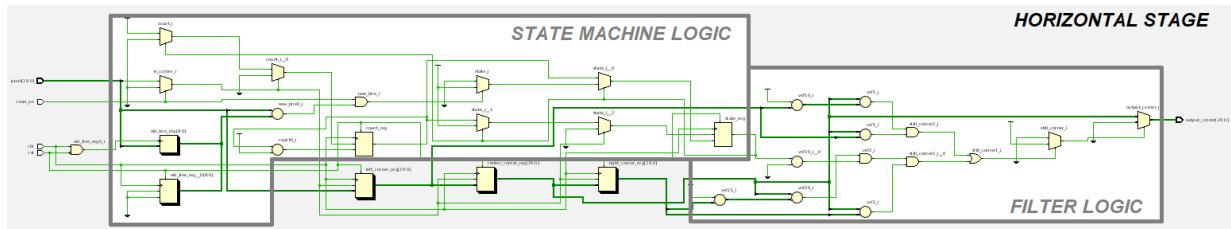


Figure 4.2.13: Horizontal data handling schematic.

Outside of both circuits representation, but inside of the module are the three registers accountable for keeping the three comparison pixels information. Their values are circularly updated every clock cycle, meaning that the left register gets the upcoming value, when the center and right get the left and center's, respectively.

Transition As previously specified, it is necessary to prepare the vertical stage by organising pixels according to its x coordinate, so this module handles it every time a new pixel arrives and without any propagation delay because it

occurs "on the wire". Besides, the critical path can be considered as short, therefore there will be no problem when starting the vertical filtering right a way.

The global linkage resumes itself a three output values, each one keeping a column position, and always in concordance with the entry pixel vertical coordinate. Basically, when a new pixel arrives, the two most recent pixels from the same column are loaded and they are all forwarded to the next stage. The next time the least recent pixel is discarded and it outputs another combination of three (using circularity again in order to save resources).

The only condition to start sending pixels to compare is to have at least two pixels per column. Otherwise, it would be impossible to filter anything. When there are no more pixels to come, the module is informed of such a situation by forwarding the last two pixels. Through this way, it guarantees the last pixel of each column is also considered as a corner candidate.

Due to the updating only when the current pixel is different from the previous one and the clock rising edge, the final verification will induct one clock cycle per column, representing the only delay in the whole accelerator. However, from another perspective, this implementation saves countless clock cycles because it allows the vertical filtering to run at the same time as the horizontal one, contrasting with the basic software implementation.

Listing 4.47: Transition module essential code.

```
// Outputs ****
assign above_corner = count[column] > 1 ? corners[((column * 3) + 0)] : 3'b0;
assign center_corner = count[column] >= 1 ? corners[((column * 3) + 1)] : 3'b0;
assign below_corner = count[column] > 1 ? corners[((column * 3) + 2)] : 3'b0;
//-----
// Control codes ****
assign is_new = (old_pixel != pixel) ? 1'b1 : 1'b0;
//-----
// Auxiliar assignments ****
always(posedge clk && is_new)
begin
    column = pixel['X_BEGIN:X_END];
    if(count[column] < 2)
        count[column] = count[column] + 1;
    corners [(column * 3)] <= pixel;
    corners [((column * 3) + 1)] <= corners [((column * 3) + 0)];
    corners [((column * 3) + 2)] <= corners [((column * 3) + 1)];
end
//-----
```

Vertical The vertical stage is the simplest one as the inputs are already arranged for comparison. There are two flags for neighbourhood verification (one for each side) and two for the scores. We verify them in pairs by executing an AND operation and it takes at least one of the conditions to be true for the corners to be suppressed. The output only gets the values of the non-suppressed ones.

Listing 4.48: Defines

```

1  assign vnf1 = (center_corner['Y_BEGIN':'Y_END] == (above_corner['Y_BEGIN':'Y_END]+1'd1))
2      ? 1'b1
       : 1'b0;
3  assign vnf2 = (center_corner['Y_BEGIN':'Y_END] == (below_corner['Y_BEGIN':'Y_END]-1'd1))
4      ? 1'b1
       : 1'b0;
5  assign vsf1 = (center_corner['SCR_BEGIN':'SCR_END] < above_corner['SCR_BEGIN':'SCR_END])
6      ? 1'b1
       : 1'b0;
7  assign vsf2 = (center_corner['SCR_BEGIN':'SCR_END] < below_corner['SCR_BEGIN':'SCR_END])
8      ? 1'b1
       : 1'b0;
9  assign still_corner = ((vnf1 && vsf1) || (vnf2 && vsf2))
10     ? 1'b0
       : 1'b1;
11  assign output_corner = still_corner ? center_corner
12      : 17'b0;
13
14
15
16
17

```

It was already present in figure 4.2.13 but here, in more detail, and also illustrating the vertical stage, there is the NMS filter schematic.

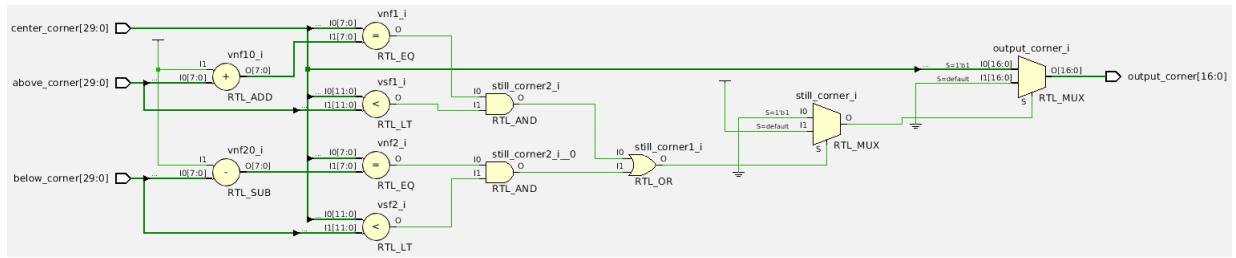


Figure 4.2.14: Filter's schematic.

4.2.8 3D NMS

Three dimensions NMS has firstly to guarantee that everything is ready to perform such suppression filter. For that, there are some flags designated in the following code fragment by control codes. Then, validated that part and completed the variables interpolation, we are plenty of conditions to operate the algorithm.

Listing 4.49: 3D Module - Control, interpolation and suppression

```

/***** Control codes *****/
3  assign ready_a = (full_pos_a > full_pos_int_a) ? 1'b1
       : 1'b0;
4  assign ready_b = (full_pos_b > full_pos_int_b) ? 1'b1
       : 1'b0;
5  assign o_ready2compare = (ready_a && ready_b);
//-----
8  /***** 3D suppression *****/
9  assign o_is_keypoint = (((score > i_score_a) && (score > i_score_b))
10    && o_ready2compare) ? 1'b1 : 1'b0;
11  assign o_keypoint_coord = (o_is_keypoint) ? {x_pos,y_pos}
12    : 17'b0;

```

```
13 //-----  
14 //***** Interpolation *****  
15     assign      o_x_int_a = x_pos * 2 / 3;  
16     assign      o_x_int_b = x_pos * 3 / 2;  
17     assign      o_y_int_a = y_pos * 2 / 3;  
18     assign      o_y_int_b = y_pos * 3 / 2;  
19 //-----
```

After the implementation show in the listing 4.49, it was necessary to create and package IPs. With help of Vivado IP Packager, any developer can create and package files and associated data in an IP-XACT format and add the IP to the Vivado catalog for further instantiations. Once that step is concluded, the IP can be inserted and customised in any project design. That's exactly what 3D refining implementation required, as each layer is in contact with two others, so we've connected the IPs inputs and outputs to transfer data among them and execute the convenient verification.

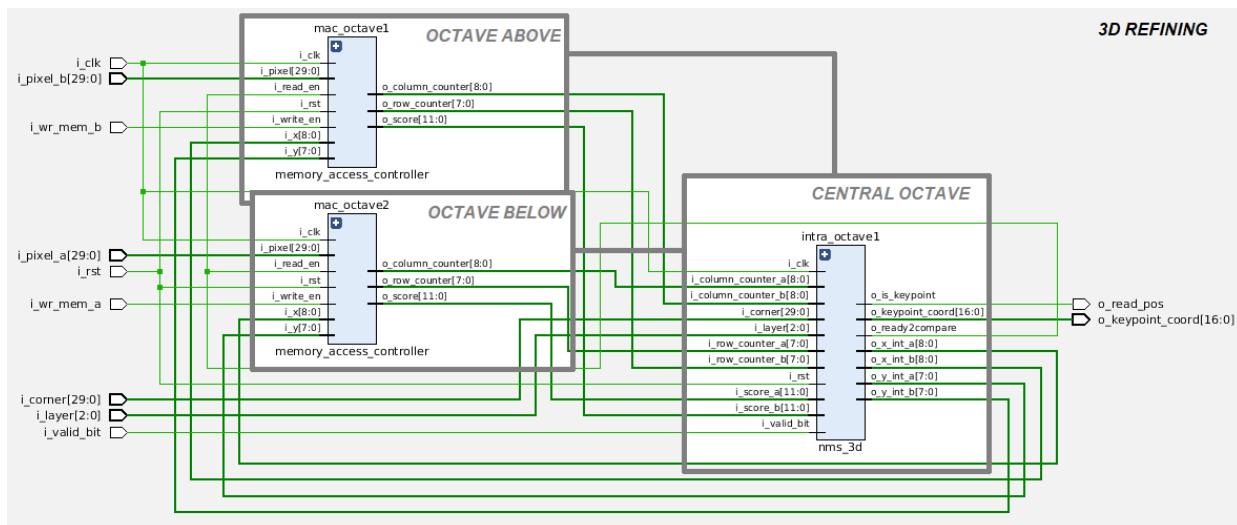


Figure 4.2.15: Octave schematic.

4.2.9 Detection integration

Once the implementation of all stages was completed, it was necessary to join them all. In figure 4.2.16 it is possible to observe the integration among all detection modules (Detection, Scoring and Non-maximum Suppression).

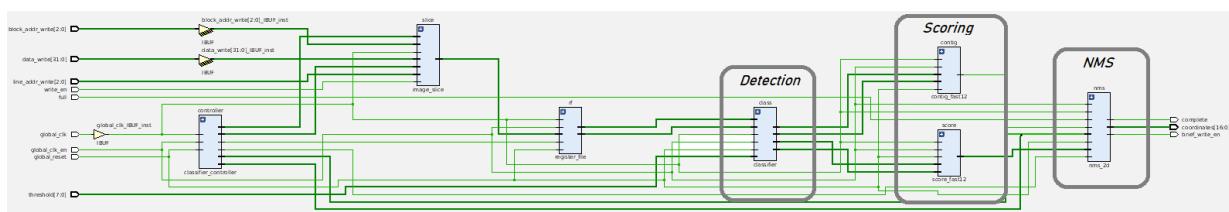


Figure 4.2.16: Schematic: Integration of BRISK detection stages.

4.2.10 Software Refactoring

Initial Code. The first goal was to develop a functional implementation. This implementation follows almost the same algorithm as the Hybrid implementation.

For each pixel P it is find the Bresenham circle, and then it is executed a comparison between the brightness of the pixel P and each of the pixels of the circle. For that it will be used a transformation array that depends of the value of fast since for each fast the circle will be different. This array will contain the transformation needed for both coordinates of the pixel P to access to all the pixels of the circle. This way is only necessary to go through the array and add the value of x and y of each position of the array to the values o x and y of the pixel P.

Listing 4.50: Transformation arrays for all the fasts

```

point_t circleTransf_12[16] = {{0, -3}, {1, -3}, {2, -2}, {3, -1},
                               {3, 0}, {3, 1}, {2, 2}, {1, 3},
                               {0, 3}, {-1, 3}, {-2, 2}, {-3, 1},
                               {-3, 0}, {-3, -1}, {-2, -2}, {-1, -3}};
5
point_t circleTransf_9[12] = {{0, -2}, {1, -2}, {2, -1},
                               {2, 0}, {2, 1}, {1, 2},
                               {0, 2}, {-1, 2}, {-2, 1},
                               {-2, 0}, {-2, -1}, {-1, -2}};
10
point_t circleTransf_5[8] = {{0, -1}, {1, -1},
                             {1, 0}, {1, 1},
                             {0, 1}, {-1, 1},
                             {-1, 0}, {-1, -1}};

```

After that, if the value of pixels, that are brighter or darker than the pixel P, is equal or greater than the value of the fast chosen by the user than the next step is the scoring an contiguity test, otherwise it will be processed another pixel.

For the contiguity test, will be created a function that will test if there are at least the value of fast pixels in a row that are brighter or darker than the pixel P. If the function returns true than P is a potential keypoint and it will be calculated its score. Otherwise, it is not a keypoint and it will be processed another pixel.

For each line will be created a list of potential keypoints that contain the coordinates, the *score* and the *suppressed* state (that will be used in the next stage and is false by default).

It will also be created a list of lists. This list will contain all the lists of potential keypoints of all the lines.

After the detection it is necessary to suppress some of the potential keypoints that are close to other keypoints. For that will be used the non-maximum suppression (NMS) algorithm. Initially, will be executed the horizontal filter for every list of the list of lists. For each *suppressed* potential keypoint the variable *suppressed* will be true and that way is possible to know which are the positions of the list that will be removed.

In the end of the horizontal stage the list of lists is cleaned, this is, all the positions which have the *suppressed* variable as true are deleted. After that the list of lists is organised in order to be easier to compare vertical neighbours.

In the vertical stage of the NMS the suppressed keypoint will also have the *suppressed* variable as true, and those will be eliminated.

Finally, the list of lists will be converted in a simple list of keypoints that only contain the coordinates of the final keypoints in order to send only the necessary information to the description stage.

In the initial code was used a cv::Mat image as input of the class. The system worked as expected but since the type of the obtained image as input belongs to the OpenCV library, was necessary to use OpenCV's functions to get the intensity of each pixel of the image. This way, there was a big dependency in the OpenCV library. To solve this

problem, the type of the image was replaced by a type of matrix that belongs to the boost library allowing the program to be faster and more independent.

Boost Implementation. This library aims to make logging significantly easier for the application developer. It provides a wide range of out-of-the-box tools along with public interfaces for extending the library. The main goals of the library are:

- Simplicity. A small example code snippet should be enough to get the feel of the library and be ready to use its basic features
 - Extensibility. A user should be able to extend functionality of the library for collecting and storing information into logs
 - Performance. The library should have as little performance impact on the user's application as possible

With that in mind, was decided that the ideal would be to change the previously linked lists with the standard library to boost intrusive lists.

Listing 4.51: BRISK structs using boost library

```
1 #include <boost/intrusive/list.hpp>
2
3 enum NFASTN{
4     FAST5 = 5,
5     FAST9 = 9,
6     FAST12 = 12
7 };
8
9
10 typedef struct{
11     int x;
12     int y;
13 } point_t;
14
15 struct Keypoint_t : public boost::intrusive::list_base_hook<> {
16     point_t point;
17     Keypoint_t () { point.x = 0; point.y = 0; }
18     Keypoint_t (int x, int y) : point({x, y}){}
19     explicit Keypoint_t (point_t p) : point{p} {}
20 };
21
22 struct keypoint_new_cloner
23 {
24     Keypoint_t *operator()(const Keypoint_t &clone_this)
25     { return new Keypoint_t(clone_this); }
26 };
27
28 struct keypoint_delete_disposer
29 {
30     void operator()(Keypoint_t *delete_this)
31     { delete delete_this; }
32 };
33
34
35 typedef boost::intrusive::list<Keypoint_t> Keypoint_l;
36
37
38 typedef struct{
39     Keypoint_t point;
40     int score;
41 }
```

```

        bool supressed;
    } potentialKeypoint_t;

41 struct PotentialKeypoint_t : public boost::intrusive::list_base_hook<> {
    potentialKeypoint_t potentialKeypoint;
    PotentialKeypoint_t () {{potentialKeypoint.point.point.x = 0; potentialKeypoint.point.point.y = 0;}
                           potentialKeypoint.score = 0; potentialKeypoint.supressed = false;};

    PotentialKeypoint_t (Keypoint_t p, int score, bool supressed) : potentialKeypoint({p, score, supressed}){};
46 explicit PotentialKeypoint_t (potentialKeypoint_t p) : potentialKeypoint{p} {}

};

struct potentialKeypoint_new_cloner
{
51     PotentialKeypoint_t *operator()(const PotentialKeypoint_t &clone_this)
    { return new PotentialKeypoint_t(clone_this); }
};

struct potentialKeypoint_delete_disposer
{
56     void operator()(PotentialKeypoint_t *delete_this)
    { delete delete_this; }
};

61 typedef boost::intrusive::list<PotentialKeypoint_t> PotentialKeypoint_l;

struct ListPotentialKeypoint_t : public boost::intrusive::list_base_hook<> {
    PotentialKeypoint_l potentialKeypointList;
    ListPotentialKeypoint_t () {}
66    ListPotentialKeypoint_t (PotentialKeypoint_l p) {}
};

struct listPotentialKeypoint_delete_disposer
{
71     void operator()(PotentialKeypoint_l *delete_this)
    { delete delete_this; }
};

typedef boost::intrusive::list<ListPotentialKeypoint_t> ListPotentialKeypoint_l;

```

The linked lists used are intrusive, this means that the list itself doesn't exist, there is only a pointer for the first position. This means that every time that it is necessary to create an object to put in a list, this object needs to be a pointer since it is necessary to allocate space, otherwise in the end of the object's scope, it would be deleted and the list would be corrupted.

Beyond the lists it was also used the boost library for the matrix of the input image. The function "*convert2matrix*" converts the image of the format cv::Mat that depends of the OpenCV library. After the conversion, the code is completely free of OpenCV functions and types.

Listing 4.52: Function to convert image to boost library's matrix

```

matrix<uchar> convert2matrix(cv::Mat image)
{
    matrix<uchar> imageMatrix(image.rows, image.cols);
    for (int i = 0; i < imageMatrix.size1 (); ++ i)
        for (int j = 0; j < imageMatrix.size2 (); ++ j)
            imageMatrix(i, j) = image.at<uchar>(i, j);
    return imageMatrix;
}

```

Table 4.1: Comparison between implementation with and without boost

Average Execution Time	Time (ms)
Initial Implementation	18.665
Boost Implementation	17.504

Multi-Threading Implementation. The results of the implementation with the boost library were not too significant, and because of that was decided to use a multi-threading implementation using the POSIX Threads library. There was a need to re-implement some section such as the detection stage. In the previous implementation there was only a function responsible for the detection of all the keypoints of the image. In this implementation, was created threads, where each of them computes a single row of the image, and returns a list with the keypoints of that row, and for that was necessary to create a function that receives the number of the row wanted and computes only that row. This implementation requests more attention since it is necessary to organise the lists of lists of keypoints, since it is necessary to be in order to be possible to execute the NMS stage correctly.

Listing 4.53: BRISK compute_threads function

```

1  Keypoint_l* BRISK_threads::compute_threads() {
2      createThreads((void*)(tComputeThreadFunction));
3      joinThreads();
4      nonMax();
5      return convertVerticalListsToList();
6  }

```

Listing 4.54: Create and Joint Threads functions

```

1  void BRISK_threads::createThreads(void *function) {
2      for(int i = 0; i < number_of_threads; i++)
3          pthread_create(&threads_array[i], nullptr, (void * (*) (void *)) function, (void*)(this));
4
5  void BRISK_threads::joinThreads() {
6      for(int i = 0; i < this->number_of_threads; i++)
7          pthread_join(threads_array[i], nullptr);
8

```

The NMS stage only begins when all the threads are over. The NMS stage is not implemented using the multi-threading approach since there is not a significant difference between the implementation without multi-threads.

4.3 BRIEF

To introduce BRIEF's implementation section, are presented the RTL diagrams of the two architectures implemented. The first, presented in the Figure 4.3.1, is an architecture where the modules execute in a sequential manner among them and therefore there is no kind of parallelism. The appearance of this diagram is similar to the design diagrams that were made.

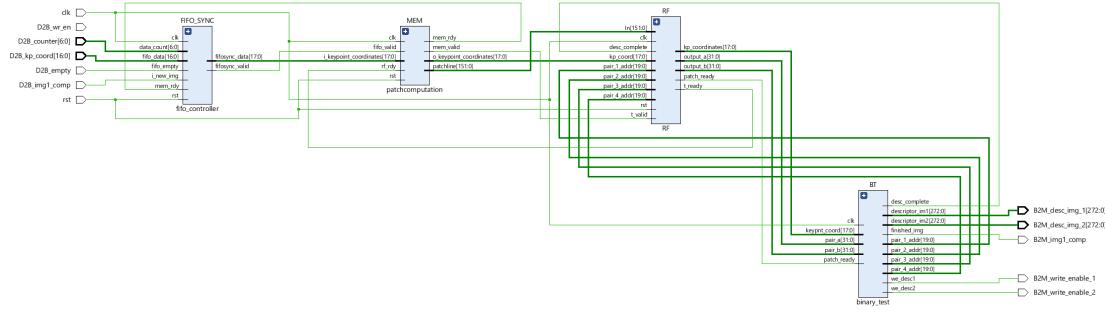


Figure 4.3.1: BRIEF RTL Diagram

The second architecture, seen in the Figure 4.3.2, is a systolic one, and the goal is to increase the performance and throughput of this stage, in exchange of a greater resources usage. To achieve this, the register file module is duplicated, as well as the binary test one. In order to synchronise the resulting scheme, additional control is required and for that are created two another modules with that purpose.

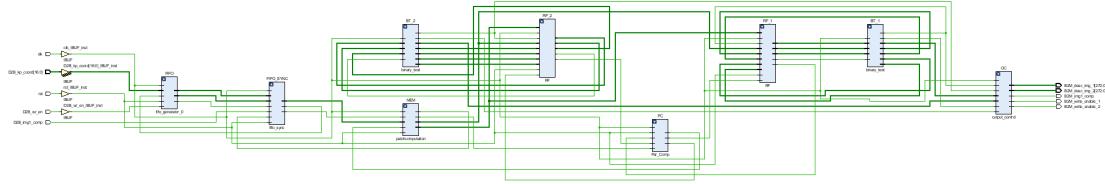


Figure 4.3.2: BRIEF with systolic architecture RTL Diagram

4.3.1 FIFO Controller

The detection stage and the description stage share a FIFO data struct between them, where the keypoints are stored by the detection stage and grabbed by the description stage whenever the descriptor is ready to grab them. The module that sets up the access interface to this FIFO from the description side is declared as presented in the Listing 4.55. As functional inputs, this module receives keypoints coordinates incoming from the FIFO (fifo_data) and a bit informing if the detection in the first or the second image is over (i_new_img), incoming from the detection stage. The output is a concatenation of this bit with the keypoint coordinates present at the input.

Listing 4.55: Fifo Controller module declaration

```

module fifo_controller(
    input wire clk,
    input wire rst,
    4   input wire fifo_empty,
    input wire [6:0] data_count,
    input wire [16:0] fifo_data,
    9   input wire mem_rdy,
    input wire i_new_img,
```

```

    output wire fifosync_rdy,
    output wire fifosync_valid,

14   output wire [17:0] fifosync_data
);

```

As inputs incoming from the FIFO for synchronization purposes, fifo_empty is set when the FIFO is out of data and data_count indicates how many keypoints coordinates are present in the FIFO ready to be grabbed.

To interface with the next module, the Patch Computation module, the input mem_rdy reveals if that module is ready to receive new keypoint coordinates, the output fifosync_rdy hints if the Fifo Controller module is ready to send data and the the output fifosync_valid suggest if the data shown at the output is valid.

Listing 4.56: Fifo Controller module's outputs assignment

```

assign fifosync_rdy = mem_rdy && ~wait1CC;
assign fifosync_valid = ~fifo_empty;
assign fifosync_data = {o_new_img, fifo_data};

```

Taking in account the value of the FIFO internal counter and the flag signaling that the detection is finished in the first image, that is set when the last keypoint coordinates of the first image are stored in the FIFO and stays set until the last keypoint coordinates of the second image are stored in the FIFO, the module needs to be able to set the outputs according to the current state it is in.

Put this, an internal counter is assigned with the value of the current FIFO internal counter value when the input image flag is set and the internal counter is decremented every clock positive edge until it reaches 1, being the output image flag set at that point.

Listing 4.57: Fifo Controller module control

```

//image 1 ended
2 if(i_new_img && !wait_new_img)
begin
    counter <= data_count;
    wait_new_img <= 1;
end
7
//image 2 ended
else if(!i_new_img) begin
    wait_new_img <= 0;
    o_new_img <= 0;
end
12

//image 1 ended but there are some keypoints of the image 1 in the fifo
else if(wait_new_img && fifosync_valid && mem_rdy && counter > 1) begin
    counter <= counter - 1;
end
17

//there are no keypoints left of the image 1 in the fifo
else if(wait_new_img && fifosync_valid && mem_rdy && counter == 1) begin
    o_new_img <= 1;
    wait_new_img <= 0;
end
22

```

4.3.2 Patch Computation

The Patch Computation module is in charge of requesting the processing subsystem the image pixels around a keypoint that form a patch and, to achieve that, this module calculates the memory addresses of all the pixels that build a patch and output these lines, line by line, to the Register File module, until the full patch is built. Put this, the module's declaration can be observed in the Listing 4.58. As can be seen, this module has as functional inputs keypoint coordinates (16 bits) and the information which is the image where the inputted keypoint belongs, which is the most significant bit of the 17-bit input. The coordinates of the keypoint under processing and the information about which is the image being processed are outputted until a new keypoint is grabbed using the same concatenation strategy.

Listing 4.58: Patch Computation module declaration

```

1  module patchcomputation #(parameter N=15)(
2    input          clk,
    input          rst,
    input          fifo_valid,
    input          rf_rdy,
    input [17:0]   i_keypoint_coordinates,
    output [17:0]  o_keypoint_coordinates,
    output [((N*8)-1):0] patchline,
    output         mem_rdy,
    output         mem_valid
  );

```

The other inputs and outputs are present due to synchronisation reasons and are essential to assure the integration of this module in the BRIEF's module. As it was already seen, fifo_valid indicates that the fifo shared between the detection and the description stage is populated, while rf_rdy imply that the register file is ready to receive a new patch. The output mem_rdy indicates the module responsible for the fifo synchronisation that the Patch Computation module is ready to read a new keypoint. For this flag to be set, the iterator regarding the line of the patch being outputted needs to be 0, what means the board was reset or a keypoint ended up being processed, and the flag that indicates if there is currently a keypoint read that was not processed yet, mem_has_data, needs to be 0. Then, if the flags mem_rdy and fifo_rdy are set, a new keypoint is read and stored in a local variable, leading to a rising edge in the mem_has_data flag and a falling edge in the mem_rdy.

Listing 4.59: Patch Computation mem_rdy flag

```

assign mem_rdy = (line_iterator == 0 && !mem_has_data) ? 1 :
               0;

```

Regarding the flag denoting if the data being outputted is valid, mem_valid, it is set whenever the line iterator is greater than 1. At the point the line iterator is incremented to 1, the patch line linked to the value 0 of the line iterator is outputted.

Listing 4.60: Patch Computation mem_valid flag

```

assign mem_valid = (line_iterator > 0) ? 1 :
                  0;

```

The line iterator is incremented when rf_rdy and mem_has_data are set to 1 and is incremented until its value is equal to the patch size, returning to 0 afterwards.

Listing 4.61: Patch Computation line_iterator and mem_has_data flag

```

11    always@(posedge clk)
12        begin
13            if(rst)
14                begin
15                    line_iterator <= 0;
16                    mem_has_data <= 0;
17                end
18            else
19                begin
20                    if(rf_rdy && mem_has_data)
21                        line_iterator <= (line_iterator < N) ? line_iterator+1 : 0;
22                    if(mem_rdy && fifo_valid)
23                        mem_has_data <= 1;
24                    if(line_iterator == N)
25                        mem_has_data <= 0;
26                end
27        end

```

Previously, in the design stage, was deducted an equation able to calculate the addresses of the first pixels of all the lines in the description patch, the equation 3.12. Extending this equation, so it does not have any substitution variables, the equation 4.1 is obtained. It is worth recalling that it is being taken in account that the images are stored in the memory in a way that each memory position possesses 32 bits, and thus 4 pixels per position

$$\text{patch line add} = (x - 1) \left(\frac{W}{4} \right) + \frac{y - 1}{4} - \frac{N - 1}{2} \left(\frac{1}{4} \right) - \left(\frac{W}{4} \right) \left(\frac{N - 1}{2} - i \right) \quad (4.1)$$

In order to implement this equation, it was reduced to require less logic and, therefore, less resources. The equation 4.2 exhibits a middle step of the reduction and the equation 4.3 the reduced equation.

$$\text{patch line add} = \frac{1}{4} \left[(x - 1) W + (y - 1) - \left(\frac{N - 1}{2} \right) - W \left(\frac{N - 1}{2} - i \right) \right] \quad (4.2)$$

$$\text{patch line add} = \frac{1}{4} \left[(x - 1) W + (y - 1) - \left(\frac{N - 1}{2} \right) (1 + N) + Ni \right] \quad (4.3)$$

Since the image size is fixed as 320x240 pixels, W is 320, N is the patch size and i is the iterator that goes from 0 till N-1 to output a new address each clock cycle. The result of this equation is multiplied by 100 in the implementation to suppress the necessity of floating-point precision, being the two less significant decimal digits the pixel position's representation in the content of the memory address obtained. This result is called extended address due to the fact that it represents not only the address of the first pixel of the patch line i-1, but the position of that pixel in the address. To exemplify, an extended address with the value 15025 indicates the pixel is the 2nd pixel of the 150th memory position.

Put this, the implementation of the extended address calculation is presented in the Listing 4.62. This calculation is naturally influenced by which image carries the keypoint being processed, since the two images are placed in different regions of the memory and the offset needs to be set accordingly.

Listing 4.62: Patch Computation extended_address

```

1 assign extended_address = (!i_img1_comp) ? (((keypoint_x-1)*320)+keypoint_y-1-(((N-1)/2)*(321))+(320*line_iterator))*25 :
2     (i_img1_comp) ? (((keypoint_x-1)*320)+keypoint_y-1-(((N-1)/2)*(321))+(320*line_iterator))*25)+(N*80)-1 :
3         0;

```

As it was stated, the two less significant decimal digits of the extended address hint what is the position of the keypoint in the memory position where it is contained. Considering that a decimal number ended in 00 has 00 as the two less significant bits in binary, a number ended in 25 has 01 as the two less significant bits in binary, a number ended in 50 has 10 as the two less significant bits in binary, a number ended in 75 has 11 as the two less significant bits in binary, and that these cover all the possible outcomes for the extended address, it is possible to deduct a conditional output that extracts the necessary pixels to form a patch line. This output is influenced by the patch size and the mentioned two less significant binary bits of the extended address, besides being evidently influenced by the address. The listing 4.63 presents how this conditional output is implemented for a patch size of 11x11.

Listing 4.63: Patch Computation patchline conditional output

```

1 assign new_data = (rst)
2     0 :
3         (rf_rdy && extended_address[1:0] == 2'b00) ?
4             {memory[address], memory[address+1], memory[address+2], memory[address+3][7:0]} :
5             (rf_rdy && extended_address[1:0] == 2'b01) ?
6                 {memory[address][31:8], memory[address+1], memory[address+2], memory[address+3][15:0]} :
7                 (rf_rdy && extended_address[1:0] == 2'b10) ?
8                     {memory[address][31:16], memory[address+1], memory[address+2], memory[address+3][23:0]} :
9                     (rf_rdy && extended_address[1:0] == 2'b11) ?
10                         {memory[address][31:24], memory[address+1], memory[address+2], memory[address+3]} :
11                             data;

```

The RTL diagram of this module is presented in the Figure 4.3.3. Analysing, it is possible to identify the first distinguishable block as the address calculation, while the other block regards the extraction of the patch lines from the memory resorting in the address obtained.

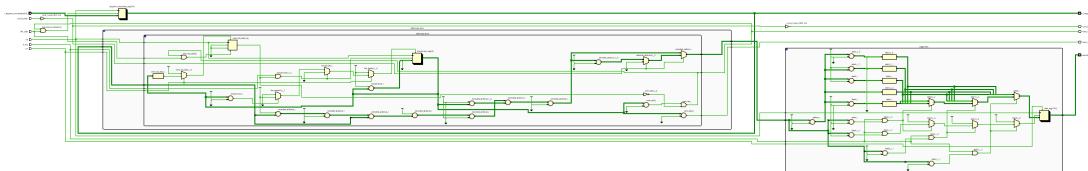


Figure 4.3.3: Patch Computation RTL diagram

4.3.3 Register File

The register file module is responsible for storing two patches along with its keypoint coordinates and image number. As shown in 4.64, the module receives three inputs from the previous module, Patch Computation module, the patch line, which size depends on the patch size N, the corresponding keypoint coordinates and the valid flag which signals valid inputs. The ready flag is sent back to the Patch Computation module to let the module know when a patch can be received in the register file.

From the Binary Tests module come 5 inputs, the flag which signals the end of the description of the patch in use and four pairs' addresses which are a concatenation of the x and y in the patch.

All the remaining outputs of the module are connected to the Binary Test module, these outputs send all A points, one of each pair, in a 32 bit concatenation and in the same manner, all the B points, the corresponding keypoint coordinates, with the image number as a concatenation, and a signal to let the Binary Test know when a complete patch is ready for description.

Listing 4.64: Register File module declaration

```

module RF#(parameter N = 15)(
    //general inputs
    input          clk,
    input          rst,
4

    //MEM -> RF
    input          t_valid,
    input [N*8-1:0] In,
    input [17:0]   kp_coord,
9

    //BT -> RF
    input          desc_complete,
    input [19:0]   pair_1_addr,
    input [19:0]   pair_2_addr,
    input [19:0]   pair_3_addr,
    input [19:0]   pair_4_addr,
14

    //RF -> BT
    output [31:0]  output_a,
    output [31:0]  output_b,
    output          patch_ready,
    output [17:0]   kp_coordinates,
19

    //RF -> MEM
    output          t_ready
24
);

```

As shown in 4.65, upon valid inputs signalled by the previous module and a free space for the next patch, the register file uses a pointer, wrPtr, to index the writing of the incoming patch lines to the available register file, represented by the variable writing_patch. As seen in the figure 4.3.4 and, the patch in register 0 is completed and under description, signalled by the variable patch_num, while the patch in the register 1 is being filled with the incoming data.

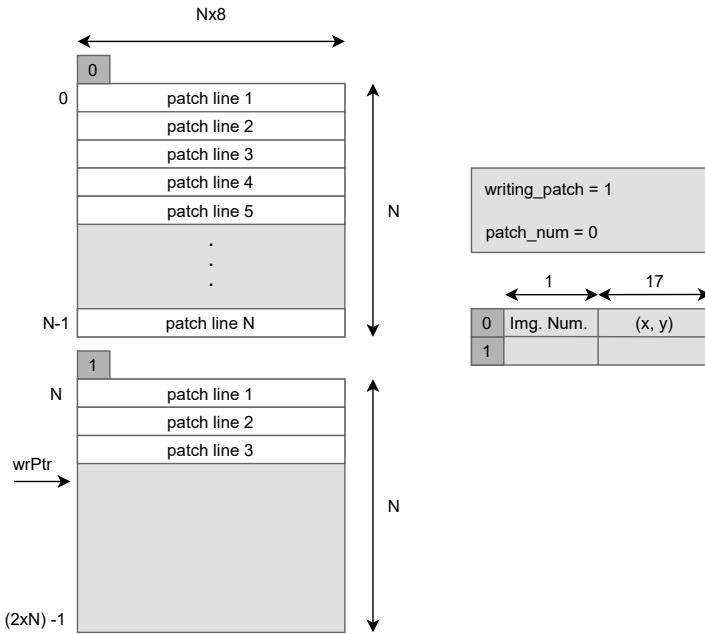


Figure 4.3.4: Register File workflow

These two variables, patch_num and writing_patch, are used to control the inputs and outputs as most of the times one register is being filled and the other has a patch under description.

The patch_num variable only switches patches upon receiving the signal that the description is finished. On the other hand, the writing_patch depends on the location of the wrPtr, since if it is in the register 0 the module is writing on the register 0 and vice-versa.

Listing 4.65: Register File's control variables

```

always@(posedge clk) begin
    if(rst)
        begin
            wrPtr <= 0;
            patch_num <= 0;
            writing_patch <= 0;
            start <= 0;
            img_num[0] <= 0;
            img_num[1] <= 0;
        end
    else begin
        if(t_valid && t_ready)
            begin
                start <= 1;
                register[wrPtr] <= In;
                wrPtr <= (wrPtr < N*2-1) ? wrPtr+1 : 0;
                img_num[writing_patch] <= kp_coord;
            end
        end
        patch_num <= (desc_complete) ? patch_num + 1 : patch_num;
        writing_patch <= (wrPtr < N-1) ? 0 : 1;
    end
end

```

The selected pairs are simply concatenated for the outputs as explained earlier. The patch_num is used to select the correct register to obtain the pairs from. As the registers 1 is right after the 0, the line is simply summed with patch_num x N, this way, N acts as an offset for the register 1. This can be seen in 4.66.

Listing 4.66: Register File's output pairs

```

1 assign output_a[31:24] = register[pair_1_addr[19:15]+(N*patch_num)][(pair_1_addr[14:10]*8 + 7) :- 8];
2 assign output_b[31:24] = register[pair_1_addr[9:5]+(N*patch_num)][(pair_1_addr[4:0]*8 + 7) :- 8];
3 assign output_a[23:16] = register[pair_2_addr[19:15]+(N*patch_num)][(pair_2_addr[14:10]*8 + 7) :- 8];
4 assign output_b[23:16] = register[pair_2_addr[9:5]+(N*patch_num)][(pair_2_addr[4:0]*8 + 7) :- 8];
5 assign output_a[15:8] = register[pair_3_addr[19:15]+(N*patch_num)][(pair_3_addr[14:10]*8 + 7) :- 8];
6 assign output_b[15:8] = register[pair_3_addr[9:5]+(N*patch_num)][(pair_3_addr[4:0]*8 + 7) :- 8];
7 assign output_a[7:0] = register[pair_4_addr[19:15]+(N*patch_num)][(pair_4_addr[14:10]*8 + 7) :- 8];
8 assign output_b[7:0] = register[pair_4_addr[9:5]+(N*patch_num)][(pair_4_addr[4:0]*8 + 7) :- 8];

```

The flags t_ready and patch_ready are very important to synchronise the modules as the first indicates that there is a free register, or with old data that can be overwritten, and the latter indicates that a patch is complete and ready to go under description.

This flags, as it can be seen in 4.67, are dependent on the patch under description and on the location of the pointer. Knowing the position of the pointer and the patch under description it is possible to know when a patch is ready, for example, when the pointer is N and the patch_num is 0 the register 0 has a patch ready.

Listing 4.67: Register File's control variables

```

1 assign t_ready = (start == 0) ? 1 : ((wrPtr == 0 & patch_num == 0) || (wrPtr == N & patch_num == 1)) ? 0 : 1;
2 assign patch_ready = ((patch_num == 0 & wrPtr == N) || (patch_num == 1 & wrPtr == 0)) ? 1 : 0;

```

The RTL design of the Register file module is presented bellow where it is possible to see the 3 registers for the 3 variables, wrPtr, patch_num and writing_patch, the register itself and all the logic involved in granting this module its correct functionalities.

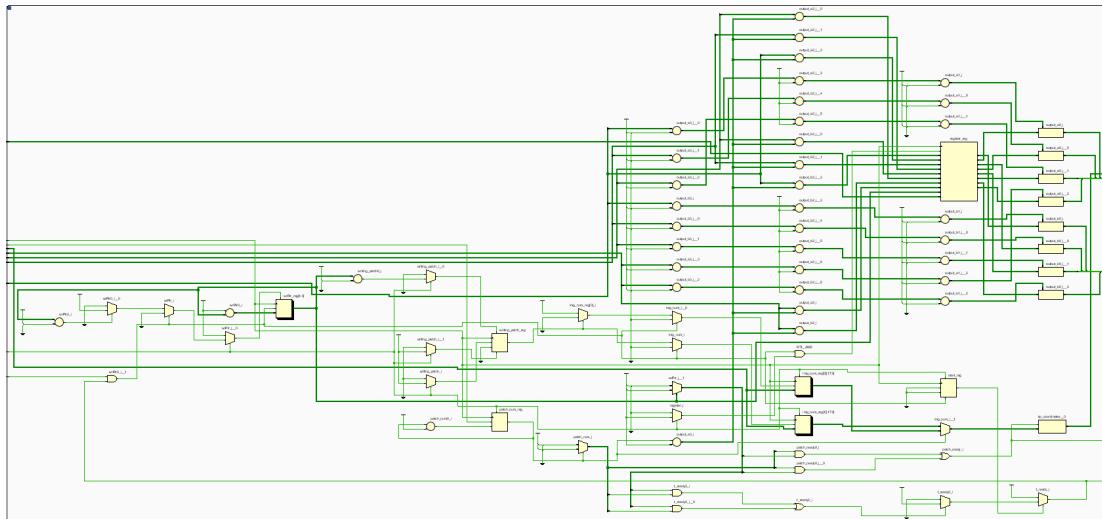


Figure 4.3.5: Register File RTL diagram

4.3.4 Binary Test

The Binary Test module is where the descriptor for each keypoint is created, this module can compare 4 pairs in a single clock cycle, for instance for a descriptor size of 256 bits it will take 64 clock cycles to complete it. There is an ongoing interaction between the binary test module and the register file module, the binary test module will at the same clock cycle compare all the current pairs and generate the addresses of the next 4 pairs to compare, the register file, will receive on the first half of the clock cycle the addresses and since reading only takes the second half of the clock cycle the moment the binary test compares all the pairs, will output the next 4 pairs to it.

Listing 4.68: Binary Test module declaration

```

1  module binary_test #
2  (
3      parameter integer MAX_DESCRIPTOR = 256,
4      parameter integer COORD_SIZE = 17,
5      parameter integer PAIR_NUM = 4
6  )
7  (
8      input clk,
9      input [31:0] pair_a,
10     input [31:0] pair_b,
11     input patch_ready,
12     input img_sel,
13     input [COORD_SIZE-1:0] keypt_coord,
14
15     output reg [19:0] pair_1_addr,
16     output reg [19:0] pair_2_addr,
17     output reg [19:0] pair_3_addr,
18     output reg [19:0] pair_4_addr,
19     output desc_complete,
20     output finished_img,
21     output [MAX_DESCRIPTOR-1 + COORD_SIZE:0] o_descriptor
22 );

```

In the Listing 4.68 one can see all the inputs and outputs of this module, the pair_a and pair_b inputs contain each two pairs to be compared. patch_ready is a flag that indicates if the input pairs are valid data. The img_sel input holds the information of which image the keypoints that are being processed belong to. The keypt_coord input has the keypoint coordinates of the current keypoint description.

In terms of outputs, one can see 4 pair addresses that are explained on the Register File module, 2 output descriptors, descriptor_im1 and descriptor_im2 for the corresponding image, and 4 flags, one for signaling that a given keypoint description is complete, desc_complete, one to tell that all keypoints of an image are described, finished_img, and 2 to tell if the descriptors outputs are valid, we_desc1 and we_desc2.

The patch pair addresses are acquired on every positive edge of the clock if there is a patch available, as shown in the Listing 4.69. In case this is true, the addresses will be acquired by concatenating the Cartesian coordinates for each point of each pair from a set of saved coordinates which are defined as parameters in the binary test module. If the description is complete for a given keypoint, then the addresses index variable is reset, otherwise it increments a decimal value of 20, which is the equivalent of incrementing 4 points, given each point has a size of 5 bits and we address the array by bit.

Listing 4.69: Computation of patch pair addresses

```

1      always @(posedge clk)
2          begin
3              if(patch_ready) delay_keypnt_coord <= keypnt_coord;
4              if(patch_available)
5                  begin
6                      pair_1_addr <= { point_a_y[addresses_index +: 5], point_a_x[addresses_index +: 5], point_b_y[addresses_index +: 5],
7                          point_b_x[addresses_index +: 5]};
8                      pair_2_addr <= { point_a_y[addresses_index+5 +: 5], point_a_x[addresses_index+5 +: 5], point_b_y[addresses_index+5 +: 5]
9                          , point_b_x[addresses_index+5 +: 5]};
10                     pair_3_addr <= { point_a_y[addresses_index+10 +: 5], point_a_x[addresses_index+10 +: 5], point_b_y[addresses_index+10 +
11                         : 5], point_b_x[addresses_index+10 +: 5]};
12                     pair_4_addr <= { point_a_y[addresses_index+15 +: 5], point_a_x[addresses_index+15 +: 5], point_b_y[addresses_index+15 +
13                         : 5], point_b_x[addresses_index+15 +: 5]};
14                     if(!desc_complete) addresses_index <= addresses_index + 20;
15                 else addresses_index <= 0;

```

When a new patch is available to be described, new pair addresses start to be formed, when this happens on the next clock cycle the begin_desc flag comes at a high value which starts the comparisons of the inputted pairs as shown in the Listing 4.70 :

Listing 4.70: Computation of descriptor

```

1      if(begin_desc)
2          begin
3              descriptor[descriptor_index] <= (pair_a[31:24] < pair_b[31:24]) ? 1 : 0;
4              descriptor[descriptor_index+1] <= (pair_a[23:16] < pair_b[23:16]) ? 1 : 0;
5              descriptor[descriptor_index+2] <= (pair_a[15:8] < pair_b[15:8]) ? 1 : 0;
6              descriptor[descriptor_index+3] <= (pair_a[7:0] < pair_b[7:0]) ? 1 : 0;
7              if(!desc_complete) descriptor_index <= descriptor_index + PAIR_NUM;
8              else descriptor_index <= 0;
9          end

```

When a positive edge of the clock happens, the descriptor assigns 4 new bits that correspond to the 4 comparisons performed above. If the descriptor is not complete the index increases its value by 4, otherwise its value is reset.

4.3.5 Systolic Architecture

The implementation of BRIEF has a heavy impact in the computation performance, as shown by the previous modules. The execution time of BRIEF can be observed in figure 4.3.6, as well as the parallelism.

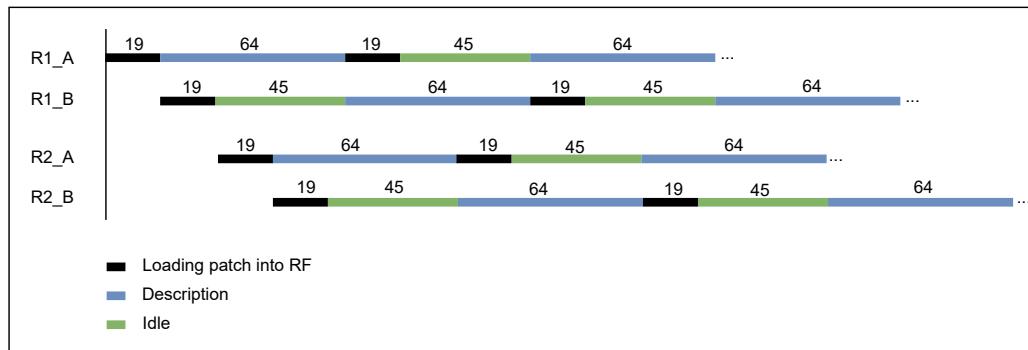


Figure 4.3.6: Execution time for BRIEF without and with parallelism

By analysing the figure above, the need for parallelism becomes evident, since each time a description is being done on a keypoint, there is another patch ready for description who will need to wait for 45 clock cycles before it is used. This occurs due to the fact that the process of loading a patch into a register file is much faster than the description itself. To minimise the impact of this, both the register file and the binary test modules were duplicated, allowing a new description to take place in parallel with another, thus decreasing the idle time. The alternative to this solution would be to increase the register file and binary test throughput, however this option would be less practical in terms of resource use, so it was discarded at the beginning.

With the strategy for the parallelism being the duplication of both the register file and the binary test, came the necessity to control the order that each register is filled and the output of each binary test (as represented in figure 4.3.7), making sure that both register files are almost always full and the transition from the first to the second memory occurred perfectly with the number of the keypoints for each image.

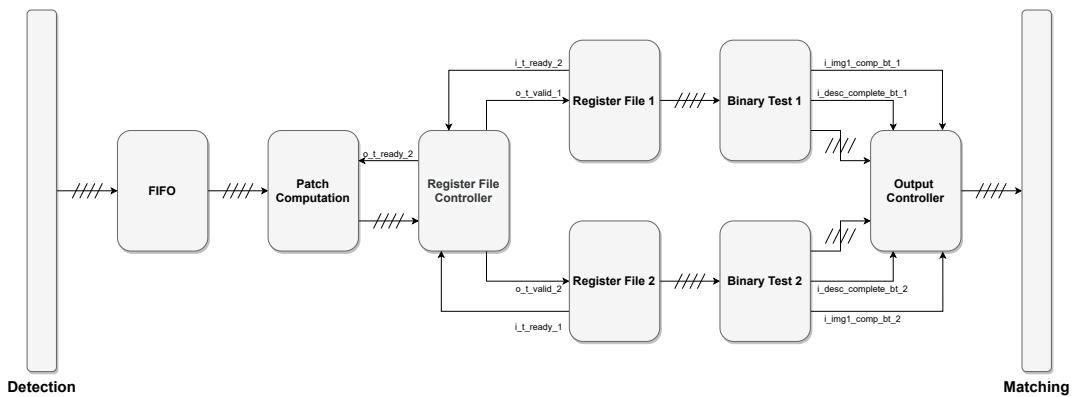


Figure 4.3.7: Block diagram for the implementation of the BRIEF parallelism

Register File Controller. The working concept for the register file controller chosen was very simple, where the main idea was only to control the synchronisation of the modules, where the controller will send the valid flag to the respective controller and ask the patch computation data every time each register ask him for data, being in a more simple term not a controller but actually a "mediator".

Listing 4.71: Register File Controller module declaration

```

1  module Par_Comp(
2    input i_clock,
3    input i_rst,
4    input i_t_valid,
5    input i_t_ready_1,
6    input i_t_ready_2,
7    output o_t_valid_1,
8    output o_t_valid_2,
9    output reg o_t_ready
10 );

```

The listing 4.71 presents all the inputs and outputs for the register file controller module, both `i_t_ready_1` and `i_t_ready_2` inputs are two flags where each one represents the register file associated with it can receive more data, while the output `o_t_ready` represents the controller asking for more data to the patch computation module. The input `i_t_valid` is once again a flag which informs the controller that the patch computation is ready to send data to one of

the register files, while the outputs $o_t_valid_1$ and $o_t_valid_2$ inform each register file is getting the data from the patch.

Listing 4.72: Computation of the register file controller

```

reg descriptor = 0;

assign o_t_valid_1 = (i_t_valid && !descriptor) ? 1 : 0;
assign o_t_valid_2 = (i_t_valid && descriptor) ? 1 : 0;

5
always@(posedge i_clock)
begin
    if(i_rst)
    begin
        o_t_ready <= 0;
        descriptor <= 0;
    end
    if(i_t_ready_1)
    begin
        o_t_ready <= 1;
        descriptor <= 0;
    end
    else if(i_t_ready_2)
    begin
        o_t_ready <= 1;
        descriptor <= 1;
    end
    else o_t_ready <= 0;
end

```

In the listing above the logic of the controller can be observed, where if the patch has data ready to be sent ($i_t_valid = 1$), then the controller will alert one of the descriptor based on the register value (lines 3 and 4 in the listing 4.72).

Each time a positive edge of the clock occurs the controller checks each flag, as shown in listing 4.72. The controller needed to know which register would have to fill depending on the flag ($i_t_ready_1/2$) that was received, so a single bit register named descriptor as implemented to guarantee that would happen (represented in line 1 of the listing 4.72).

To know each register it has to file with the data from the patch, each register sends the respective flag, meaning that the register file is available to receive more data. The controller then at each clock cycle checks the flag if it is high (1), asks more data from the patch(wasting as few cycles as needed) and changes the descriptor register to the correct value, allowing the register file to receive the data requested.

Is important to notice that first the register file 1 will be filled and then the register file 2, as shown in figure 4.3.6.

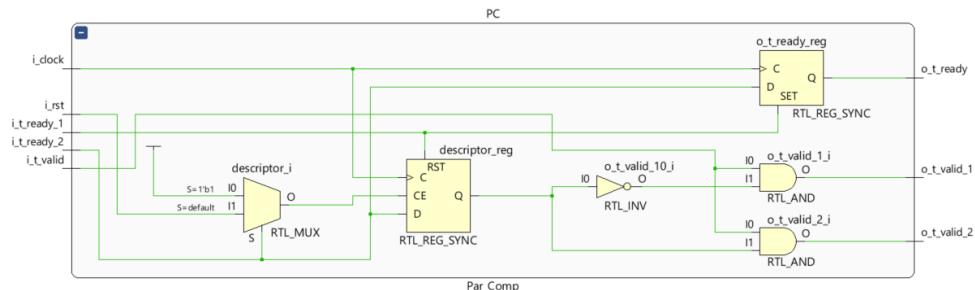


Figure 4.3.8: RTL design of the register file controller

Output Controller. The output controller is responsible for selecting the output to the matching stage based on which binary test has finished describing a keypoint. This is done to prevent collision between the binary test outputs, so only one output is used at a time.

Listing 4.73: Output Controller module declaration

```

1  module output_control #
2  (
3      parameter integer MAX_DESCRIPTOR = 256,
4      parameter integer COORD_SIZE = 17
5  )
6  (
7      input i_clock,
8      input [MAX_DESCRIPTOR-1 + COORD_SIZE:0] i_descriptor_bt_1,
9      input [MAX_DESCRIPTOR-1 + COORD_SIZE:0] i_descriptor_bt_2,
10     input i_desc_complete_bt_1,
11     input i_desc_complete_bt_2,
12     input i_img1_comp_bt_1,
13     input i_img1_comp_bt_2,
14     output [MAX_DESCRIPTOR-1 + COORD_SIZE:0] o_descriptor_im1,
15     output [MAX_DESCRIPTOR-1 + COORD_SIZE:0] o_descriptor_im2,
16     output o_img1_comp,
17     output o_we_im1,
18     output o_we_im2
19 );

```

The listing 4.73 displays all the inputs and outputs of the output controller module, the inputs `i_descriptor_bt_1` and `i_descriptor_bt_2` contain the concatenated data of the keypoint coordinates and descriptor itself for each of the binary test instances. The inputs `i_desc_complete_bt_1` and `i_desc_complete_bt_2` are flags that indicate when one of the binary tests has finished a description, and the flags `i_img1_comp_bt_1` and `i_img1_comp_bt_2` represent whether the keypoint belongs to the first or second image for each binary test. The outputs of this module include a descriptor output for each image, `o_descriptor_im1` and `o_descriptor_im2`, a write enable for each image, `o_we_im1` and `o_we_im2`, and a flag to indicate which image is being processed, `o_img1_comp`.

Listing 4.74: Computation of the output controller

```

1   reg im1_comp;
2   reg we_im1;
3   reg we_im2;
4   reg enable_desc1;
5   reg enable_desc2;
6
7   assign o_descriptor_im1 = (enable_desc1 && we_im1) ? i_descriptor_bt_1 : (enable_desc2 && we_im1) ? i_descriptor_bt_2 : 0;
8   assign o_descriptor_im2 = (enable_desc1 && we_im2) ? i_descriptor_bt_1 : (enable_desc2 && we_im2) ? i_descriptor_bt_2 : 0;
9   assign o_img1_comp = im1_comp;
10  assign o_we_im1 = we_im1;
11  assign o_we_im2 = we_im2;
12
13  always @ (posedge i_clock)
14  begin
15      if(i_desc_complete_bt_1)
16          begin
17              im1_comp <= i_img1_comp_bt_1;
18              enable_desc1 <= 1;
19              if(!i_img1_comp_bt_1) we_im1 <= 1;
20              else we_im2 <= 1;
21          end
22      else if(i_desc_complete_bt_2)
23

```

```

begin
    im1_comp <= i_img1_comp_bt_2;
    enable_desc2 <= 1;
26    if(!i_img1_comp_bt_2) we_im1 <= 1;
        else we_im2 <= 1;
    end
else
begin
31    enable_desc1 <= 0;
    enable_desc2 <= 0;
    we_im1 <= 0;
    we_im2 <= 0;
end
end

```

In the listing 4.74 the logic for the output controller is presented, on each positive edge of the clock the value of the flags `i_desc_complete_bt_1` and `i_desc_complete_bt_2` are verified to identify if one of the binary tests has finished the description of a keypoint. If for example `i_desc_complete_bt_1` is active, it means that the binary test 1 module has finished a description, so the corresponding image number flag is taken as `im1_comp`, the flag `enable_desc1` is enabled, and the write enable flag (either `we_im1` or `we_im2`) is enabled for the image being processed, depending on the input `i_img1_comp_bt_1`. The assignment of the output descriptors will be done depending on whether `enable_desc1` or `enable_desc2` is active, and `we_im1` or `we_im2`, with only one of `o_descriptor_im1` and `o_descriptor_im2` being valid at a time.

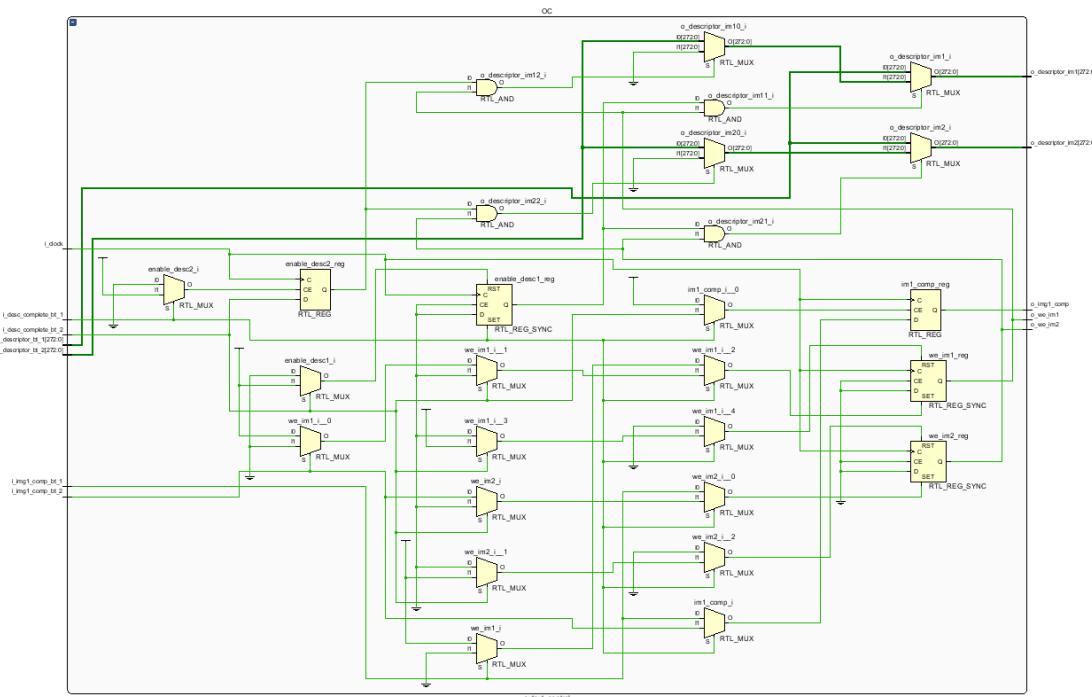


Figure 4.3.9: RTL design of the output controller

4.3.6 Software Refactoring

Sampling Pattern Generation. As explained in the chapter 3.6.3, from all the geometries and the research done, in most cases the second algorithm shows the best results, enjoying a small advantage over the rest in terms of recognition rate. So this will be the chosen method for the sampling geometry and was implemented as shown in the code below.

Listing 4.75: Matlab sampling pattern generation code

```

function pattern = gaussian_sampling_generator(window_size,BRIEF_n, addr)
    x1 = zeros(BRIEF_n,1);
    x2 = zeros(BRIEF_n,1);
    4     y1 = zeros(BRIEF_n,1);
    y2 = zeros(BRIEF_n,1);

    for i = 1:BRIEF_n
        x1(i) = floor(normrnd(0,0.04*window_size^2));
    9     while x1(i)>floor(window_size/2) || x1(i)<ceil(-window_size/2)
            x1(i) = floor(normrnd(0,0.04*window_size^2));
        end
        x2(i) = floor(normrnd(0,0.04*window_size^2));
        while x2(i)>floor(window_size/2) || x2(i)<ceil(-window_size/2)
    14       x2(i) = floor(normrnd(0,0.04*window_size^2));
        end
        y1(i) = floor(normrnd(0,0.04*window_size^2));
        while y1(i)>floor(window_size/2) || y1(i)<ceil(-window_size/2)
            y1(i) = floor(normrnd(0,0.04*window_size^2));
        end
        y2(i) = floor(normrnd(0,0.04*window_size^2));
        while y2(i)>floor(window_size/2) || y2(i)<ceil(-window_size/2)
            y2(i) = floor(normrnd(0,0.04*window_size^2));
        end
    19     end
        X = [x1 y1 x2 y2];
        disp(X);
        pattern = [x1 y1 x2 y2];
    end

```

Given this code, as demonstrated in line 26 that the pattern values were displayed, this values where then copied and put in a txt file for later being used by the code we implemented. This process was then repeated 9 times with the necessary values for the patch and descriptor size.

Similarly to the generation of the sampling pattern, Matlab was used to get the keypoints location into a txt file, allowing the execution of the code developed in a easy and accessible way wihtout depending on another modules. The respective values of the descriptors where also obtained allowing to later test and validate the code implemented.

Code Development. The development of the software implementation was divided into three different phases, the first being the implementation of the BRIEF descriptor in a simple way, the second and third phase where meant to improve the code performance via the boost library and multi-threading, respectably.

So the first phase is presented in listing 4.76, where the a simple and functional implementation was accomplished. The listing presented below follows the line of thought presented in the design phase in the figure 3.4.14.

Listing 4.76: Initial Software Implementation of BRIEF

```

1 void brief::generateDescriptor(sKeypoints keypoint) {
2     /* [...] */
3     std::string file = "brief_" + std::to_string((int)_selection) + ".txt";
4     /* [...] */
5     firstPoint.x = v[0];
6     firstPoint.y = v[1];
7     secondPoint.x = v[2];
8     secondPoint.y = v[3];
9
10    cv::Scalar firstIntensity = _image.at<uchar>(keypoint.point.x + firstPoint.x, keypoint.point.y + firstPoint.y);
11    cv::Scalar secondIntensity = _image.at<uchar>(keypoint.point.x + secondPoint.x, keypoint.point.y + secondPoint.y);
12
13    //Binary test
14    if(firstIntensity.val[0] >= secondIntensity.val[0])
15        des.str.append("0");
16    else
17        des.str.append("1");
18    /* [...] */
19}

```

Similarly to the state chart presented in chapter 3.6.5, the function presented in the listing 4.76, receives a single keypoint and generates the descriptor for that said keypoint, being later used to run across a linked list and generate the descriptor for each keypoint. The pixel intensity for each pixel of the sampling pair is obtained with the help of the OpenCV library (is presented in listing 4.76 in lines 10 and 11), which was undesirable.

Boost Implementation. Boost libraries are widely useful, and usable across a broad spectrum of applications, especially when there are large amounts of data like in computer vision technology. With that in mind, and relying on both the "boost" manuals and the website, it was decided that the ideal would be to change the previously linked lists with the standard library to boost intrusive lists, as seen in listing 4.77.

Listing 4.77: BRIEF software implementation using boost library

```

1 //***** Boost Functions *****/
2 #include <boost/numeric/ublas/matrix.hpp>
3 #include <boost/numeric/ublas/io.hpp>
4 #include <boost/intrusive/list.hpp>
5
6 typedef struct Point_t {
7     int x=0;
8     int y=0;
9 }sPoint;
10 struct Keypoint_t : public boost::intrusive::list_base_hook<> {
11     /* [...] */
12 };
13 //Cloner object function
14 struct keypoint_new_cloner
15 {
16     /* [...] */
17 };
18 //The disposer object function
19 struct keypoint_delete_disposer
20 {
21     /* [...] */
22 };
23 struct BRIEF_t : public boost::intrusive::list_base_hook<> {
24     Keypoint_t keypoint{};
25 }

```

```

26     std::string str;
27
28     BRIEF_t () { keypoint.point.x = 0; keypoint.point.y = 0; }
29     BRIEF_t (Keypoint_t p, std::string s) : keypoint{p}, str{std::move(s)} {}
30 };
31 //Cloner object function
32 struct brief_new_cloner
33 {
34     BRIEF_t *operator()(const BRIEF_t &clone_this)
35     { return new BRIEF_t(clone_this); }
36 };
37 //The disposer object function
38 struct brief_delete_disposer
39 {
40     void operator()(BRIEF_t *delete_this)
41     { delete delete_this; }
42 };
43 //Define an list that will store using the public base hook
44 typedef boost::intrusive::list<Keypoint_t> Keypoint_l;
45 typedef boost::intrusive::list<BRIEF_t> BRIEF_l;
46 }
```

Since the implementation for one singular keypoint was done (shown in listing 4.76), all that was left to do was making possible the generation of a sequence of keypoints. To meet this end, a function named compute was implemented in accordance with the state chart presented in figure 3.4.11, which receives a linked list and simply runs trough it as demonstrated in listing 4.78.

In the same listing we can also see that between line 1 to 6, a verification of the keypoint location is also needed since the sampling pattern can be bigger than the space given between the keypoint coordinates and the end of the image.

Listing 4.78: Software implementation of the general BRIEF compute function

```

const BRIEF_l &brief::compute(Keypoint_l &keypointList) {
    for(auto &it : keypointList) {
        if ( ( it.point.x > _patchSize/2 &&
              it.point.y > _patchSize/2 ) &&
            ( it.point.x < (_image.size1() - _patchSize/2) &&
              it.point.y < (_image.size2() - _patchSize/2) )
        )
    }
    /* [...] */
    generateDescriptor(*tempKeypoint);
}
else
    std::cerr << "ERROR: KEYPOINT[" << it.point.x << "," << it.point.y << "] out of limits" << std::endl;
}
return _descriptors;
}
```

Taking advantage of the boost library resources, the image was transferred to a matrix (demonstrated in listing 4.79), this change was made with the intention of removing the need of OpenCV in the BRIEF module (exemplified in listing 4.80).

Listing 4.79: Software implementation of main function for BRIEF with boost optimizations

```

int main() {
    boost::numeric::ublas::matrix<uchar> imageMatrix1(image1.rows, image1.cols);
    boost::numeric::ublas::matrix<uchar> imageMatrix2(image2.rows, image2.cols);
```

```

4   for (int i = 0; i < imageMatrix1.size1(); ++ i)
    for (int j = 0; j < imageMatrix1.size2(); ++ j)
        imageMatrix1 (i, j) = image1.at<uchar>(i, j);
5   for (int i = 0; i < imageMatrix2.size1(); ++ i)
    for (int j = 0; j < imageMatrix2.size2(); ++ j)
        imageMatrix2 (i, j) = image2.at<uchar>(i, j);
6   /* [...] */
7   auto *descriptor1 = new brief(PATCH11, BRIEF8, imageMatrix1);
8   auto *descriptor2 = new brief(PATCH11, BRIEF8, imageMatrix2);
9   descriptorsList1.clone_from(descriptor1->compute(keypointsList1), brief_new_cloner(), brief_delete_disposer());
10  descriptorsList2.clone_from(descriptor2->compute(keypointsList2), brief_new_cloner(), brief_delete_disposer());
11  delete descriptor1, descriptor2;
12  /* [...] */
13  descriptorsList1.clear_and_dispose(brief_delete_disposer());
14  descriptorsList2.clear_and_dispose(brief_delete_disposer());
15  return 0;
16 }

```

Listing 4.80: Software implementation of BRIEF binary test without OpenCV dependencies

```

void brief::generateDescriptor(const Keypoint_t& keypoint) {
    /* [...] */
    u_char firstIntensity = _image (keypoint.point.x + firstPoint.x, keypoint.point.y + firstPoint.y);
    u_char secondIntensity = _image (keypoint.point.x + secondPoint.x, keypoint.point.y + secondPoint.y);
8   //Binary test
9   if(firstIntensity >= secondIntensity)
    des->str.append("0");
10  else
    des->str.append("1");
11  /* [...] */
12 }

```

Multi-Threading Implementation. Since the software implementation will "compete" with the hardware, a multi-threading implementation was carried out using the POSIX Threads library completing and optimising the code already implemented.

Listing 4.81: Software implementation of BRIEF constructor for Multi-threading

```

briefThreads::briefThreads(ePatchSize patchSize, ebriefType briefSize,
                           const boost::numeric::ublas::matrix<u_char> &image,
                           uint8_t numberCores,
4                           Keypoint_l &keypointList)
                           : brief(patchSize, briefSize, image),
                           _numberCores(numberCores) {
    /* [...] */
    _it = _keypointList.begin();
9
    _thread_group = static_cast<pthread_t *>(malloc(sizeof(pthread_t) * _numberCores * 2));
    _lock = PTHREAD_MUTEX_INITIALIZER;
}

```

The `briefThreads` constructor is responsible for initialisation of all variables, including the `brief` class variables, and allocate space required to each threads.

Listing 4.82: Software implementation of BRIEF Create and Joint Threads functions

```

void briefThreads::createThreads() {

3   for (int i = 0; i < (_numberCores * 2); ++i) {
    pthread_create(&_thread_group[i], nullptr, tComputeThreadFunction, (void *)(this));
  }

8   void briefThreads::jointThreads() {
    for (int i = 0; i < (_numberCores * 2); ++i) {
      pthread_join(_thread_group[i], nullptr);
    }
}

```

Once threads are allocated, the creation of these threads are necessary to execute the createThreads function. In order for the program to wait for the threads to terminate, it was necessary to execute the joint function.

Listing 4.83: Software implementation of BRIEF compute function for Multi-threading

```

void *briefThreads::tComputeThreadFunction(void *arg) {
  /* [...] */
  for (;;) {
    pthread_mutex_lock(&_brief->_lock);
    if (_brief->_it != _brief->_keypointList.end() )
    {
8      it = _brief->_it;
      _brief->_it++;
      pthread_mutex_unlock(&_brief->_lock);
      /* [...] */
    }
13    else
    {
      pthread_mutex_unlock(&_brief->_lock);
      return nullptr;
    }
18  }
  return nullptr;
}

```

Since threads can execute the same parts of the code at the same time, but with different execution states they were used to accelerate the description process (demonstrated in listing 4.83), making it possible to describe more than one keypoint at a time.

A mutex lock mechanism is not only used for synchronization but also to protect data sharing between threads, so it was strictly necessary to implement it, so the threads access to the lists is done one at a time safely.

Listing 4.84: Software implementation of main function for BRIEF with Multi-threading

```

#include "briefThreads.h"
#include "opencv2/opencv.hpp"
/* [...] */
int main() {
5   std::cout << "Software Implementation - BRIEF Threads" << std::endl;
  /* [...] */
  const std::string imagePath1 = "/simba-the-lion-king-2019-movie.jpg";
  const std::string imagePath2 = "/lioness-predator-relaxed.jpg";
  std::string path1 = path + imagePath1;
}

```

```

10     std::string path2 = path + imagePath2;
11     cv::Mat image1 = cv::imread(path1, cv::IMREAD_GRAYSCALE );
12     cv::Mat image2 = cv::imread(path2, cv::IMREAD_GRAYSCALE );
13     cv::GaussianBlur(image1, image1, cv::Size(3, 3), 0);
14     cv::GaussianBlur(image2, image2, cv::Size(3, 3), 0);
15     /* [...] */
16     if ( f_Image1.is_open() && f_Image2.is_open() ) {
17         std::string line;
18         while (!f_Image1.eof()) {
19             /* [...] */
20         }
21         while (!f_Image2.eof())
22         {
23             /* [...] */
24         }
25     }
26     else
27     {
28         std::cerr << "Unable to open Keypoints file" << std::endl;
29         /* [...] */
30         auto *descriptor1 = new briefThreads(PATCH11, BRIEF8, imageMatrix1, 1, keypointsList1);
31         auto *descriptor2 = new briefThreads(PATCH11, BRIEF8, imageMatrix2, 1, keypointsList2);
32         descriptor1->createThreads();
33         descriptor2->createThreads();
34         descriptor1->jointThreads();
35         descriptor2->jointThreads();
36         /* [...] */
37         std::cout << std::endl << "----- Description List 1
38         -----" << std::endl;
39         for(auto & it : descriptorsList1)
40             std::cout << std::endl << "Keypoint["<< it.keypoint.point.x << "," << it.keypoint.point.y << "]"
41             << tDescriptor: \"<< it.str << "\"" << std::endl;
42         std::cout << std::endl << "----- Description List 2
43         -----" << std::endl;
44         for(auto & it : descriptorsList2)
45             std::cout << std::endl << "Keypoint["<< it.keypoint.point.x << "," << it.keypoint.point.y << "]"
46             << tDescriptor: \"<< it.str << "\"" << std::endl;
47         /* [...] */
48         return(0);
49     }
50 }
```

4.4 Matching

4.4.1 Hardware

Matching Core Hardware

In the implementation of the matching core in hardware, two modules were developed. One to match the descriptors and the other to select the descriptor with the best index and the smallest hamming distance.

The match module takes as inputs the two descriptors to be compared and outputs the hamming distance between them. This module also takes as input the index of one of the descriptors, in order to select the best descriptor at the end of all comparisons.

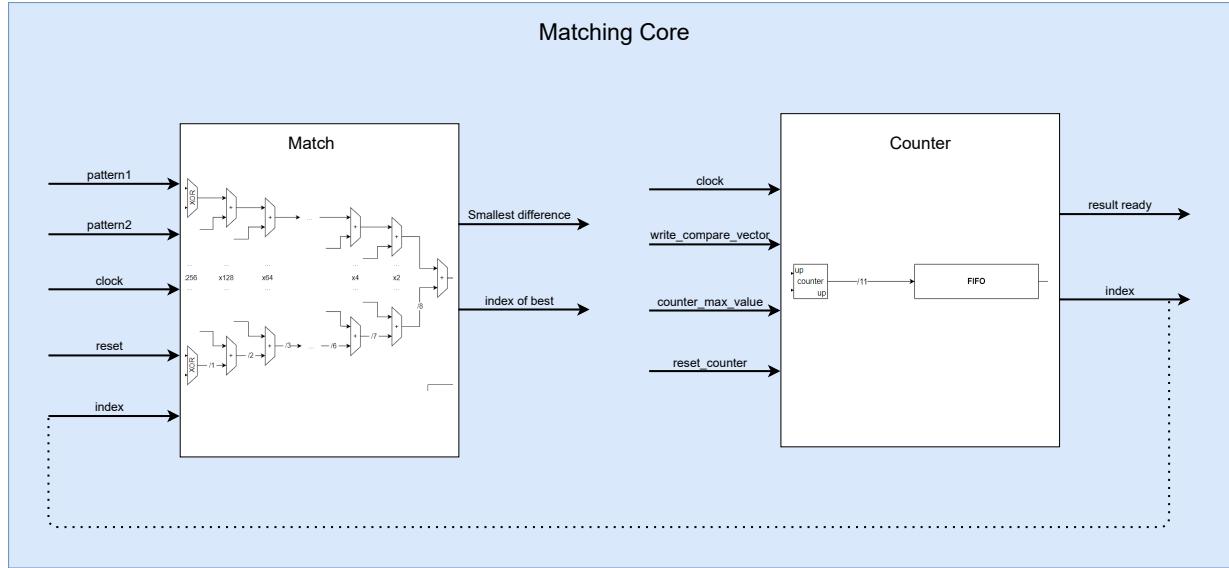


Figure 4.4.1: Matching Core Diagram Modules

As previously said, a pipeline technique is used in order to compute the hamming distance. Knowing that, it is relevant to define the stages in the pipeline and what the different stages will do.

There will be nine different pipeline stages to solve the problem. The first stage will be responsible to make the xor operation between the two descriptors, one being the pattern and the other descriptor being a descriptor from the common part. The following stages will be responsible for transforming the result of the xor operation into the value that represents the hamming distance, thus proceeding to successive summations. The diagram 4.4.2 represents the different stages of pipeline and the pipeline registers, responsible for carrying the results from one stage to the next.

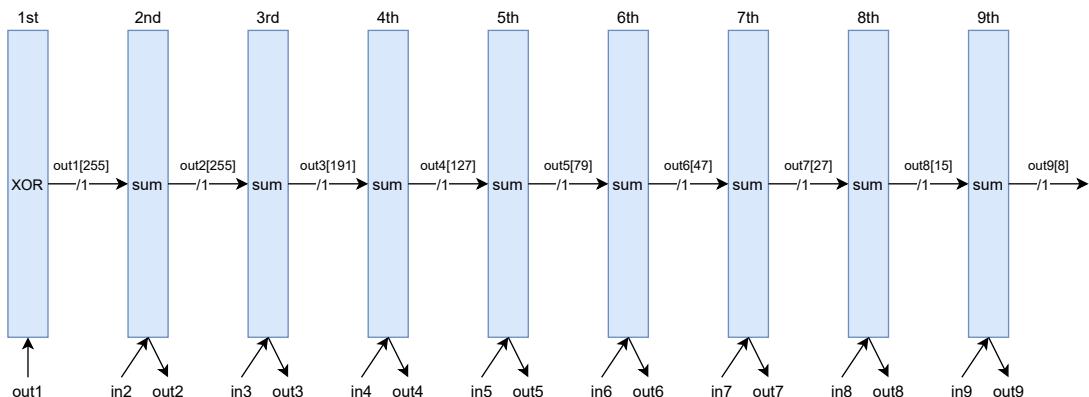


Figure 4.4.2: Matching Core Pipeline Registers and Stages

The size in bits of the register of each stage is calculated taking into account the different sums.

Knowing that the first stage results in a 256-bit vector and that it will be necessary to sum the number of 1's in the subsequent stages, it will be necessary to come up with a method to perform the sum as developed in the project

design. This method will affect the number of bits between registers in each stage of the pipeline. Therefore, the strategy used is presented below.

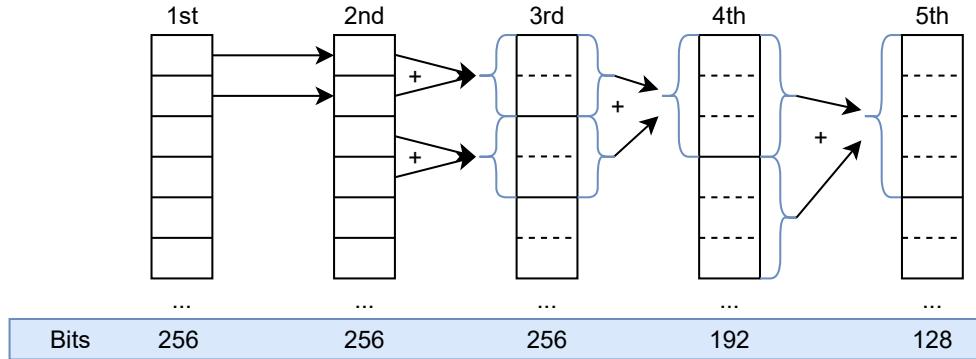


Figure 4.4.3: Matching Core Sums

The figure 4.4.11 explains the fact that the registers have different sizes. The bits represent the size between the stages. In the first stage there are 256 bits, as well as the second stage. In the second stage to the third, the sums begin. So, considering the vector positions starting in zero, the value in position zero with the value in position one will be added. Consequently, the second position will be added with the third position, repeating this process until the end of the vector. The sums of the position zero with the first position results in a number with two bits. So, the third stage will still result into a vector with 256 positions.

The third step is responsible for adding the two bits of the last sum with the next two bits. So, from position zero to the second position will be added the value of the third position to the fourth. The result is a number with three bits that is stored into an vector that only needs 192 positions.

This reasoning is repeated up to the ninth stage resulting in a vector of only 9 positions and representing the hamming distance between the two descriptors being matched.

In order to compute the index of the best descriptor, a FIFO is used. The FIFO length is equal to nine, which is the value responsible for representing the latency associated to the pipeline.

The inputs that this module has are the pattern to start the counter, the reset responsible for resetting the counter, and the counter maximum value, responsible for defining the maximum value that the counter should count, putting the result ready to one at the end.

The schematic RTL of the two modules is represented on figure 4.4.4. It is possible to know the number of input/output ports used as well as the number of Nets used.

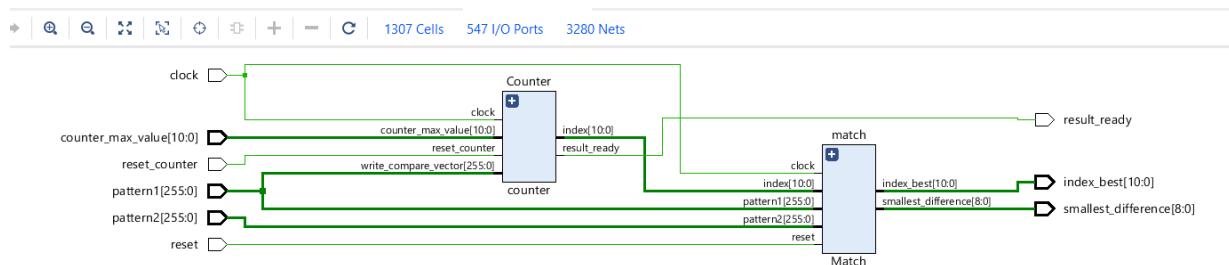


Figure 4.4.4: Matching Core RTL schematic

With these schematic design is possible to know the number of LUTs and Flip flops that the system spent. These values are presented below, on figure 4.4.5. This of course is a number representative for only one matching core.

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	LUT	FF	BRAM	URAM	DSP	Start	Elapsed	Run Strategy
✓ synth_1	constrs_1	synth_design Complete!							775	1026	0.0	0	0	6/5/21, 9:17 PM	00:00:28	Vivado Synthesis Defaults (Vivado Synthesis)
▷ impl_1	constrs_1	Not started														Vivado Implementation Defaults (Vivado Implementation)

Figure 4.4.5: Matching Core LUT and FF

Going into a more in-depth analysis, it will be important to look at the design of each module, and see what similarities they have compared to the design analysis.

Therefore, looking into the matching module, it was expected to see the nine stages of pipeline and the successive sums of each stage.

In the figure 4.4.6 is possible to see the nine stages, being the first, the xor operation, and then the successive sums.

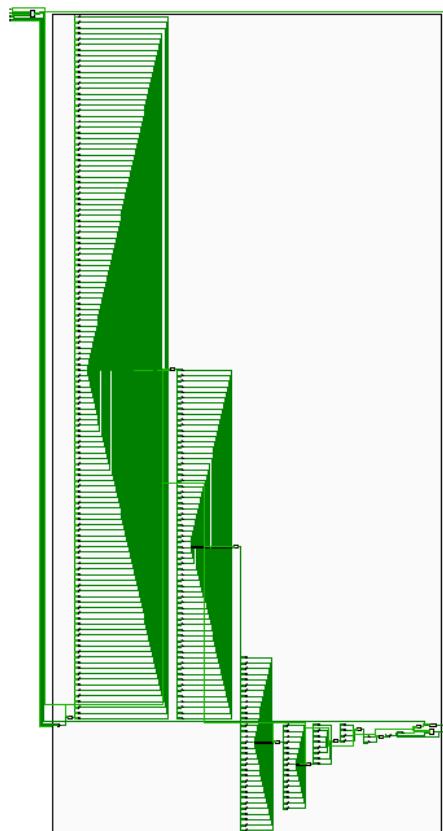


Figure 4.4.6: Matching Core Nine stages Schematic

Looking in more detail, the sums can be seen better in the last stages. With that, it is also possible to see the registers responsible for saving the index of the best descriptor and the smallest difference of the respective descriptor.

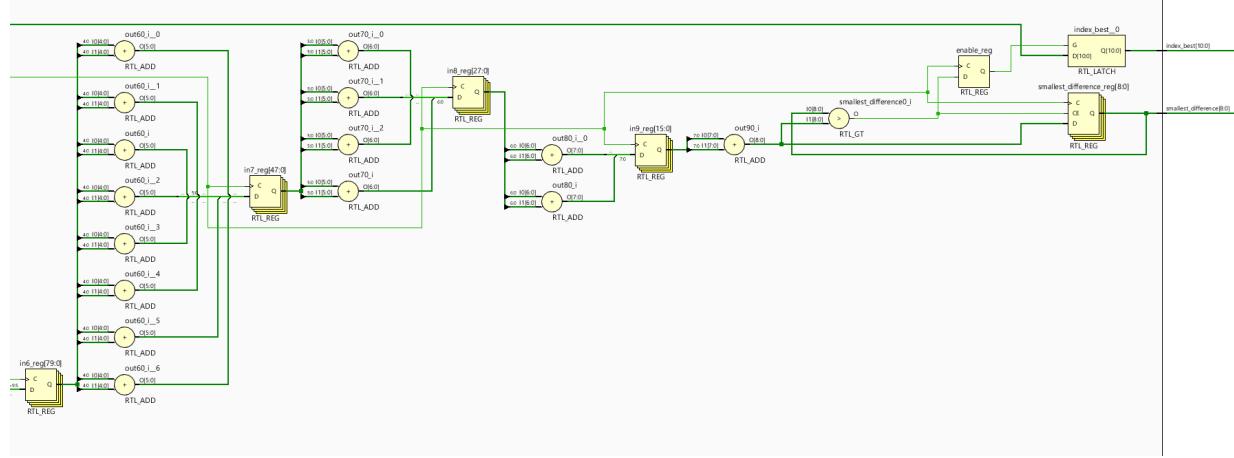


Figure 4.4.7: Matching Core Last Stages

The schematic design that was developed for the counter module, responsible for computing the FIFO with the nine positions, is represented in figure 4.4.8. It is possible to see the nine registers, responsible for saving the values of each descriptor.

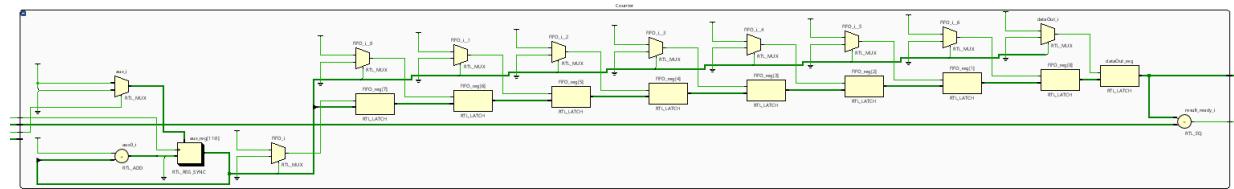


Figure 4.4.8: Matching Core Counter RTL

To implement the matching core a module called match was developed, which is responsible for performing the xor operation and the sums that will result in the hamming distance between two given descriptors. This module can be consulted bellow.

Listing 4.85: Matching Core Code

```

1  module Match
2
3    input clock,
4    input reset,
5    input [255:0]pattern1,
6    input [255:0]pattern2,
7    input [10:0] index,
8    output reg [8:0] smallest_difference,
9    output [10:0] index_best
10   );
11

```

```

16   //inputs and outputs of stages
17   wire [255:0] out1;
18   wire [255:0] out2;
19   wire [191:0] out3;
20   wire [127:0] out4;
21   wire [79:0] out5;
22   wire [47:0] out6;
23   wire [27:0] out7;
24   wire [15:0] out8;
25   wire [8:0] out9;

26   // registers between stages
27   reg [255:0] in2;
28   reg [255:0] in3;
29   reg [191:0] in4;
30   reg [127:0] in5;
31   reg [79:0] in6;
   reg [47:0] in7;
   reg [27:0] in8;
   reg [15:0] in9;

parameter N = 256;
reg enable;

```

The match module has as inputs the two descriptors and the index of the descriptor that will be constantly changed, being the common part. As outputs it has a register responsible for storing the smallest hamming distance and the index of the corresponding descriptor that was just compared with the pattern.

All the registers in 2,3,4,5,6,7,8 and 9 are the ones that will store the result of the sum of the different pipeline stages implemented, allowing each stage to access the results of the previous stages.

Listing 4.86: Matching Core Code

```

//pipeline
//2nd stage  input2  output2
genvar ii;
4 generate
  for (ii=0;ii<N;ii=ii+2) begin
    assign out2[ii+1:ii] = in2[ii] + in2[ii+1];
  end
endgenerate

```

To implement each stage of the pipeline it was necessary to use a for loop to generate the number of sums required for each stage.

The code above demonstrates the implementation of stage 2 of the pipeline. To store the index of the descriptor with the smallest hamming distance, the module (counter module), referenced before was developed to determine this index.

The module's main input is counter_max_value, which will indicate the maximum number of descriptors that will be compared, in other words, the number of descriptors that the common part has. This module also has as output the descriptor index and a flag that will be activated whenever the comparisons are all executed.

Listing 4.87: Matching Core Code

```

// **** COUNTER UP ****
4 reg [11:0] aux = 4095; //12 bits number 11 bits value 120 bit signal
  reg [10:0] FIFO [7:0];
  reg [10:0] dataOut;

```

```

1  always @( posedge clock )begin
2    if(reset_counter)
3      aux <= 0 ;
4    else if (write_compare_vector >= 0)
5      begin
6        aux[11] <= 0;
7        aux[10:0] <= aux[10:0] + 1;
8      end
9
10 end
11
12
13
14

```

The strategy adopted to calculate the index of the descriptor being compared is through a counter, that is, whenever there is a positive clock transition the index will increase, corresponding to the entry of the next descriptor in the matching core.

Listing 4.88: Matching Core Code

```

1 // **** result_ready ****
2 assign result_ready = (index == counter_max_value) ? 1 : 0;

```

When the index reaches the maximum value (index = counter_max_value), it means that all comparisons in the list of descriptors to be compared have already been performed, and therefore the descriptor with the smallest hamming distance has already been found.

Listing 4.89: Matching Core Code

```

//match with pipeline
Match match(
  .clock(clock),
  .reset(reset),
  .pattern1(pattern1),
  .pattern2(pattern2),
  .index(index),
  .smallest_difference(smallest_difference),
  .index_best(index_best)
);

//common part with FIFO
counter Counter(
  .clock(clock),
  .reset_counter(reset_counter),
  .write_compare_vector(pattern1),
  .counter_max_value(counter_max_value),
  .result_ready(result_ready),
  .index(index)
);

```

Finally, the Matching_Core module will call the two previously mentioned modules and together they will constitute a matching core.

4.4.2 BRAM

As previously stated, a BRAM was chosen to store all descriptors. Therefore, two BRAM's were implemented and consequently the tests for their validation were developed.

The inputs and outputs required for the BRAM will be the data to be written, being on this case all the descriptors, a read address to make the read operation, an address to write the descriptors on the memory, a write enable to allow

or disallow the write operation, the read and write clock signals for the read and write synchronous operation. BRAM's output will be the data that wants to be read. It is possible to see the inputs and outputs of BRAM on the following code.

Listing 4.90: BRAM Code

```

1  module Dual_port_ram #(
2    parameter DATA_WIDTH=256,           //width of data bus
3    parameter ADDR_WIDTH=8            //width of addresses buses
4  )(
5    input      [DATA_WIDTH-1:0] data,      //data to be written
6    input      [ADDR_WIDTH-1:0] read_addr,   //address for read operation
7    input      [ADDR_WIDTH-1:0] write_addr,  //address for write operation
8    input      we,                      //write enable signal
9    input      read_clk,                //clock signal for read operation
10   input      write_clk,               //clock signal for write operation
11   output reg [DATA_WIDTH-1:0] q        //read data
12 );
13
14
15   reg [DATA_WIDTH-1:0] ram [2**ADDR_WIDTH-1:0]; // ** is exponentiation
16
17   always @(posedge write_clk) begin //WRITE
18     if (we) begin
19       ram[write_addr] <= data;
20     end
21   end
22
23   always @(posedge read_clk) begin //READ
24     q <= ram[read_addr];
25   end
26
27 endmodule

```

4.4.3 Controller

The controller is the entity responsible for storing the patterns received by BRIEF in the memory explained above, for implementing the state machine to perform the brute force algorithm and also computed the cross checking. The controller module inputs and outputs are related to its functionality, being some of those responsible for interacting with the matching core and signals to control the BRAM access, between others presented next.

Listing 4.91: Controller Core Inputs and Outputs

```

1 controller_1 #( .DATA_WIDTH(256), .ADDR_WIDTH(7))
2 controller
3 (
4   .reset(reset),
5   .clk(clk),
6   .data_1(data_1),
7   .data_2(data_2),
8   .result_image1(result_image1),
9   //All descriptors of image are stored in memory
10  .result_ready(w_result_ready),
11  .write_enable_image1(write_enable_image1),
12  .write_enable_image2(write_enable_image2),
13  .best_match(w_index_best),
14  .best_match2(w_index_best2),
15

```

```

16     .best_match3(w_index_best3),
17     .r_common_part(w_common_part),
18     .r_first_pattern(w_first_pattern),
19     .r_second_pattern(w_second_pattern),
20     .r_third_pattern(w_third_pattern),
21     .w_counter_max_value(w_counter_max_value),
22     .r_out_reset(w_out_reset)
23 );

```

The first responsibility of the controller is to receive the patterns from BRIEF, and store those patterns on the BRAM.

The number of descriptors on image one is incremented every time that the enable image1 is one and the result ready is not one. The result ready input is a flag that comes to one if the image 1 is completely passed to the BRAM. When the image 2 is also complete this signal becomes zero. The number of descriptors on image two is incremented every time that the write enable becomes one and if the result image1 is one. The w_write_addr is an input on the BRAM that represents the address of both BRAM's. This is important to know where is the address to store the incoming descriptors on the BRAM.

Listing 4.92: Controller module Store Patterns

```

//Increase counters and address, if BRIEF enable the write_enable lines
assign w_new_num_desc_image1=(write_enable_image1 && !result_image1) ? r_wr_num_desc_image1 + 1 : r_wr_num_desc_image1;
assign w_new_num_desc_image2=(write_enable_image2 && result_image1) ? r_wr_num_desc_image2 + 1 : r_wr_num_desc_image2;
assign w_write_addr1 = (write_enable_image1 && w_new_num_desc_image1 > 1) ? r_write_addr_bram1 + 1 : r_write_addr_bram1;
5 assign w_write_addr2 = (write_enable_image2 && w_new_num_desc_image2 > 1) ? r_write_addr_bram2 + 1 : r_write_addr_bram2;

```

First is important to initialized all the registers when the reset comes to one. At each clock cycle, the descriptors data is passed to the matching cores module and the write and read address registers of the descriptors are updated.

Listing 4.93: Controller module Store Patterns - continuation

```

1 //Reset and update registers at posedge of clock
2 always @(posedge clk) begin
3   if(reset)begin
4     r_status <= 'WAIT;
5     r_wr_num_desc_image1 <= 0;
6     r_wr_num_desc_image2 <= 0;
7     r_write_addr_bram1 <= 0;
8     r_write_addr_bram2 <= 0;
9     r_rd_num_desc_image2 <= 0;
10    end
11   else begin
12     r_status <= w_new_status;
13     r_write_addr_bram1 <= w_write_addr1;
14     r_write_addr_bram2 <= w_write_addr2;
15     r_wr_num_desc_image1 <= w_new_num_desc_image1;
16     r_wr_num_desc_image2 <= w_new_num_desc_image2;
17   end
18   if(r_status>3) begin
19     r_common_part <= w_data_image2;
20     r_first_pattern <= w_data_image1;
21     r_second_pattern <= w_data2_image1;
22     r_third_pattern <= w_data3_image1;
23   end
24   else begin
25     r_common_part <= w_data_image1;
26     r_first_pattern <= w_data_image2;
27     r_second_pattern <= w_data2_image2;
28     r_third_pattern <= w_data3_image2;
29   end
30 
```

```
    end
end
```

To do the brute force algorithm a state machine is implemented on the controller. Therefore, the controller has seven states. The first 4 states are responsible for implementing the brute force algorithm and the rest to perform the cross checking - discussed in the next subsection.

Listing 4.94: Controller States

```
//Define Status States
2  'define WAIT    0
  'define LOAD    1
  'define RUN     2
  'define FINISH  3
  'define CC_LOAD 4
7  'define CC_RUN  5
  'define CC_FINISH 6
```

The WAIT state is responsible for waiting that BRIEF starts sending the second image descriptors in order to do the algorithm.

The LOAD state is responsible to charge the descriptors into the matching core and The RUN is responsible for starting the matching core operation and then waiting until it receives the result ready signal. The result ready signal means that the index of the best and the smallest difference were found and so it is time to charge a new descriptor to the common part of the matching core, if there are still descriptors left to be compared. If not, it is time finish the algorithm.

Listing 4.95: Controller State Machine Status

```
//New_status of the system
3  assign w_new_status = ((r_status == 'WAIT) && r_num_desc_left_img2 == 0) ? 'WAIT: //Wait for BRIEF
    to put the second image descriptors
        ((r_status == 'WAIT) && r_num_desc_left_img2 > 2) ? 'LOAD: //Charge the
            patterns if descriptors available
        ((r_status == 'LOAD) && (r_count==2)) ? 'RUN: //Starting
            running the descriptors to compare
        //((r_status == 'RUN) && !result_ready)
            //Matching core is not finished
        ((r_status == 'RUN) && w_finish && result_ready) ? 'FINISH: //Go to Finish if
            the match is over and no more descriptors left
        ((r_status == 'RUN) && result_ready && r_num_desc_left_img2 == 0) ? 'WAIT: //Go to Wait if
            2nd image descriptors are not available
        ((r_status == 'RUN) && result_ready && r_num_desc_left_img2 > 0) ? 'LOAD: //Charge patterns
            if 2nd image descriptors are available
        ((r_status == 'FINISH) && r_finish) ? 'CC_LOAD: //Reset the
            system
        ((r_status == 'CC_LOAD) && (r_count==2)) ? 'CC_RUN: //Reset the
            system
        //((r_status == 'CC_RUN) && !result_ready)
        ((r_status == 'CC_RUN) && counter >= r_wr_num_desc_image2 && result_ready) ? 'WAIT:
        ((r_status == 'CC_RUN) && result_ready && counter < r_wr_num_desc_image2) ? 'CC_LOAD: //Charge patterns if 2nd image descriptors are available
13
```

The addresses of the descriptors to be evaluated in the matching core must be updated (incremented) during the Run state and if there are still descriptors of image 1 and 2.

Listing 4.96: Controller descriptors address

```

//Charge pattern and common_part with the desired values ready to compare and make the pipeline
assign w_new_address_1 = r_status == 'CC_LOAD ? best_matches[counter]:
5      (r_address_1 < r_wr_num_desc_image1-1) && (r_status == 'RUN) ? r_address_1 + 1 : r_address_1;
assign w_new_address2_1 = r_status == 'CC_LOAD ? best_matches[counter+1]:
(r_address_1 < r_wr_num_desc_image1-1) && (r_status == 'RUN) ? r_address_1 + 1 : r_address2_1;
assign w_new_address3_1 = r_status == 'CC_LOAD ? best_matches[counter+2]:
(r_address_1 < r_wr_num_desc_image1-1) && (r_status == 'RUN) ? r_address_1 + 1 : r_address3_1;

10 assign w_new_address_2 = (r_address_2 < r_wr_num_desc_image2-1) && (r_status == 'CC_RUN) ? r_address_2 + 1:
(r_count==1 && r_rd_num_desc_image2 > 0) ? r_address_2 + 3 : r_address_2 ;

```

The state machine is performed at each clock cycle.

Listing 4.97: Controller State Machine code

```

//Status machine actions
3 always @(posedge clk) begin
    if(reset)begin
        r_address_1 <= 0;
        r_address2_1 <= 0;
        r_address3_1 <= 0;
8        r_address_2 <= 0;
        r_count <= 0;
        r_out_reset <= 1;//Reset the matching core
    end
    else if(r_status=='WAIT)begin
13        r_out_reset <= 1;//Reset the matching core
        r_finish <= 0;
        r_address_1 <= 0;
        r_address2_1 <= 0;
        r_address3_1 <= 0;
    end
    else if(r_status=='LOAD)begin
        r_out_reset <= 1; //Reset the matching core
        r_address_1 <= 0; // Reset address 1 to start a new matching cycle
        r_address2_1 <= 0; // Reset address 1 to start a new matching cycle
23        r_address3_1 <= 0; // Reset address 1 to start a new matching cycle
        r_rd_num_desc_image2 <= w_new_rd_num_desc_image2;
        r_count <= r_count + 1; //Increment to change pattern
        r_address_2 <= w_new_address_2;
    end
    else if(r_status=='RUN) begin
28        r_out_reset <= 0; //Release the reset to start
        r_count <= 0; //Reset counter
        r_address_1 <= w_new_address_1; //New descriptor to the matching core
        r_address2_1 <= w_new_address2_1; //New descriptor to the matching core
33        r_address3_1 <= w_new_address3_1; //New descriptor to the matching core
    end
    else if(r_status=='CC_LOAD)begin
        r_out_reset <= 1; //Reset the matching core
        r_address_2 <= 0; // Reset address 1 to start a new matching cycle
38        r_count <= r_count + 1; //Increment to change pattern
        r_address_1 <= w_new_address_1;
        r_address2_1 <= w_new_address2_1; //New descriptor to the matching core
        r_address3_1 <= w_new_address3_1; //New descriptor to the matching core
    end
    else if(r_status=='CC_RUN) begin
        r_out_reset <= 0; //Release the reset to start
        r_count <= 0; //Reset counter

```

```

        r_address_2 <= w_new_address_2; //New descriptor to the matching core
    end
48  else if (r_status=='FINISH)begin
      r_out_reset <= 1;//Reset the matching core
      r_address_2 <= 0;
      r_address_1 <= 0;
      r_finish <= 1;
      counter<=0;
    end
53

```

It can be seen that there are three addressed for addressing image 1, each one being one pattern for each matching core, thus the system containing three matching cores. There is only need for one address for image 2 because the common part will be the same for all patterns being compared.

4.4.4 Cross Checking

To implement the Cross Checking three new states were added to the controller state machine.

When the matching process ends the next state is the CC_LOAD. It is responsible to reset the Matching Core models and send to it the new descriptors from image 1 and the first of image 2. The new descriptors from the image 1 comes from an array that saves all the indexes of the best matches from the matching process, implementing the process as it was design during the design phase.

The array best_matches stores the index of image 1 that presents the best match. To do the cross-checking the contents of each index from the BRAM address is loaded, in order to get the descriptor of image 1, to do the matching with all descriptors from image 2 (cross-checking).

In the CC_RUN the common part descriptor address are successively loaded. In the CC_RUN state every time the result ready comes to one and the crossing check is not yet finished the status goes from CC_RUN to CC_LOAD, loading new patterns to the matching cores.

During the cross checking process, each time result ready is active, it is checked if the best descriptor at the matching core output is equal to the array index stored, in order to validate the cross checking. If this condition is met, the most significant bit (Valid Bit) gets the value 1. If not, the bit remains with the value zero.

Listing 4.98: Cross Checking Array

```

1   always @(posedge result_ready) begin
2     counter <= counter + 3;
3     if(r_status == 'LOAD || r_status =='RUN) begin
4       best_matches[counter] <= best_match;
5       best_matches[counter+1] <= best_match2;
6       best_matches[counter+2] <= best_match3;
7     end
8     else if(r_status == 'CC_LOAD || r_status =='CC_RUN)begin
9       if(best_match==counter)begin
10         best_matches[counter]['validCC]<=1;
11       end
12       if(best_match2==counter+1)begin
13         best_matches[counter+1]['validCC]<=1;
14       end
15       if(best_match3==counter+2)begin
16         best_matches[counter+2]['validCC]<=1;
17       end
18     end
19   end

```

4.4.5 Wrapper Module

It was necessary to develop a module responsible for being the bridge between the matching core and the controller, in order to do the algorithm. Since the controller was developed to have more than one matching core, the wrapper calls the matching core module three times.

The schematic of the developed module is shown below in figure 4.4.9, where it is possible to see all the modules and the three cores. Furthermore, for this wrapper module with three matching cores, the number of LUT's and 's can be consulted in figure 4.4.10.

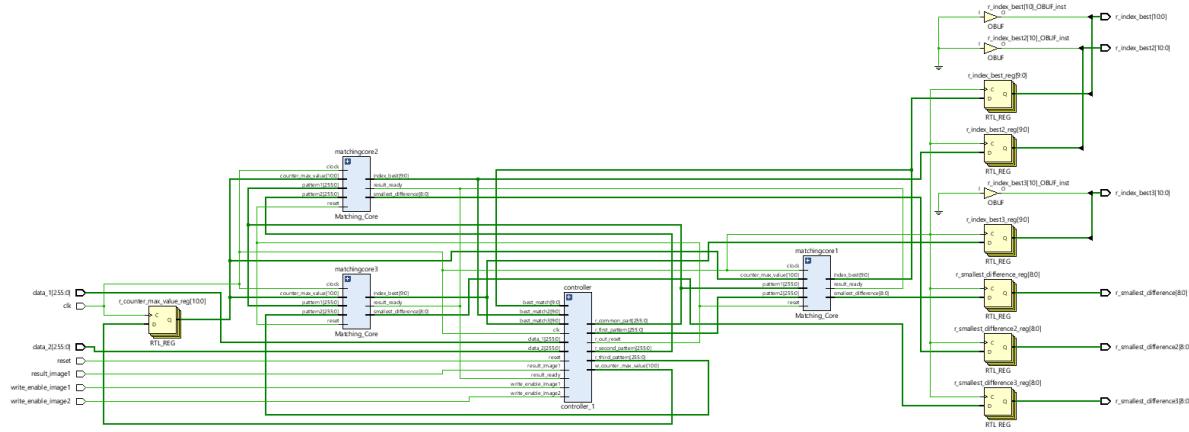


Figure 4.4.9: Wrapper Module

Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	LUT	FF	BRAM	URAM
synth_1	constrs_1	synth_design Complete!							3030	4203	24.0	0
impl_1	constrs_1	Not started										

Figure 4.4.10: Wrapper Module Resources

4.4.6 Software

As mentioned in previous sections, as far as software is concerned there will be two major implementations to it:

- The *openCV* implementation;
- The refactored matching implementation.

The *openCV* implementation will serve as a basis through which the time constraints of the refactored software will be compared with in regards to time constraints. This implementation uses several complex algorithms with complex objects therefore it should, theoretically, be slower than the refactored version.

For validation of the refactored software, the keypoints, descriptors and matches will be converted into *openCV* objects, to be given to the "drawMatches" function, it was opted to use an *openCV* function rather than creating one

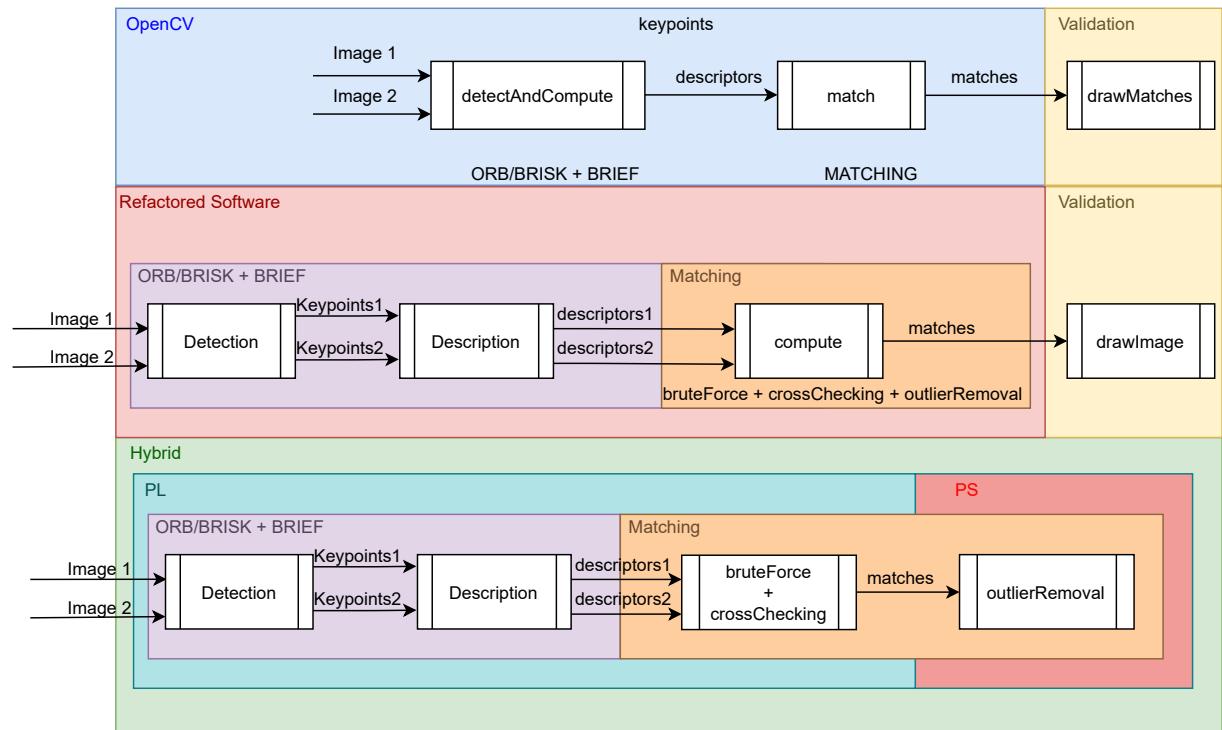


Figure 4.4.11: Overall Matching Implementation Diagram

from scratch because the validation is not part of the matching *per se* and therefore, the time constraints of the rest of the project do not apply.

In regards to the *openCV* implementation itself, all that was specified was the matching algorithm used (brute-force), everything else, including detection, description and outlier removal was left to the library itself.

4.4.7 Software Refactored: Matching

Once the previous phases have the keypoints, their coordinates and their descriptors available it is possible to start the matching.

The first step is the brute force algorithm, it consists of, as mentioned in previous sections, comparing each descriptors from one image to all the descriptors of the other image, calculating the hamming distance and whichever has the lowest distance is considered the best match.

Afterwards begins the cross checking, where all descriptors from the other image are compared to all descriptors of the first image, the hamming distance is calculated and whichever has the least distance is considered the best match. Next it is necessary to check for inconsistencies a process that was dubbed "clearing false matches", for instance if descriptor *A* from the first image has descriptor *B* from the second image as its best match, according to the brute force algorithm but descriptor *B* does not have *A* as the best match according to the cross checking there is an incoherence and both keypoints are discarded.

The implementation was done entirely using the *BOOST* library for dealing with image processing, specifically used for the linked lists of the keypoints, and in the matching itself the boost dynamic bitset was used for dealing with the descriptors.

All the basics and methodology surrounding this implementation can be found in section 3.5.

Matching Refactored: Main Implementation

The brute force method traverses the linked list of image 1, performs the XOR operation and calculates the hamming distance, as mentioned previously it makes full use of the boost dynamic bitset for the manipulation of the descriptors. At the end of this method the best matches of image 1 are stored.

Listing 4.99: Brute Force Method

```

void matcher::bruteForceAlgorithm(BRIEF_l *list1, BRIEF_l *list2) {

3   /* [...] */
  for (BRIEF_l::iterator itList1 = list1->begin(); itList1 != list1->end(); ++itList1) {
    /* [...] */
    for (itList2 = list2->begin(); itList2 != list2->end(); ++itList2) {
      /* [...] */
8     while (it1 != itList1->str.end() && it2 != itList2->str.end()) {
      xDesc[i] = *it1 ^ *it2;
      /* [...] */
    }
    uint16_t HDistance = calcHammingDistance(xDesc);
13   if (HDistance < lowestHDistance) {
      lowestHDistance = HDistance;
      /* [...] */
    }
  }
  /* [...] */
  _bestDescFromImage2.push_back(*bestDescImage2);
}

```

The calculation of the Hamming Distance consists of counting the number of ones in the result of the XOR operation between descriptors:

Listing 4.100: Hamming Distance Method

```

uint16_t matcher::calcHammingDistance(boost::dynamic_bitset<> xTemp) {
  /* [...] */
  for(int i = 0; i < xTemp.size() ; i++) {
    counter += xTemp[i];
  }
  return counter;
}

```

Next step is the Cross Checking, basically the brute force method but for finding the matches for image 2:

Listing 4.101: Cross Checking Method

```

void matcher::crossCheckingAlgorithm(BRIEF_l *list1, BRIEF_l *list2) {
  /* [...] */
  for (BRIEF_l::iterator itList2 = list2->begin(); itList2 != list2->end(); ++itList2) {
    /* [...] */
    for (itList1 = list1->begin(); itList1 != list1->end(); ++itList1) {
      /* [...] */
      while (it1 != itList1->str.end() && it2 != itList2->str.end()) {
        xDesc[i] = *it1 ^ *it2;
        /* [...] */
      }
    }
  }
}

```

```

13     uint16_t HDistance = calcHammingDistance(xDesc);
14     if (HDistance < lowestHDistance) {
15         lowestHDistance = HDistance;
16         /* [...] */
17     }
18     /* [...] */
19     _bestDescFromImage1.push_back(*bestDescImage1);
20 }
```

Finally "removing false matches", or checking for incoherencies and removing the pair of keypoints in case of such incoherencies:

Listing 4.102: Clear False Matches Method

```

1 void matcher::clearFalseMatches(BRIEF_L *list1, BRIEF_L *list2) {
2     /* [...] */
3
4     if(_bestDescFromImage1.size() >= _bestDescFromImage2.size()) {
5         /* [...] */
6     }
7     else {
8         /* [...] */
9     }
10    while ( startIt != endIt ) {
11        /* [...] */
12        while ( bestListIt != bestListEndIt ) {
13            if ((startIt->keypoint.point.x == startListCompareIt->keypoint.point.x) &&
14                (startIt->keypoint.point.y == startListCompareIt->keypoint.point.y) &&
15                (bestListIt->keypoint.point.x == bestListCompareIt->keypoint.point.x) &&
16                (bestListIt->keypoint.point.y == bestListCompareIt->keypoint.point.y)) {
17
18                /* [...] */
19                _matchingList.push_back(*matcher);
20            }
21            /* [...] */
22        }
23        bestListIt = temp1;
24        startListCompareIt = temp2;
25        /* [...] */
26    }
}
```

4.5 DSL

4.5.1 Grammar Rules

The grammatical rules that shape the DSL were easier to implement since the syntax was already defined in the design phase and can be seen in listing 4.103. These grammar rules are strict and impose greater limitations to the user, which is not a disadvantage, since it indirectly makes the user make fewer mistakes and makes the validation rules following this phase easier.

Listing 4.103: Grammar rules

```

Model:
    VisionSystem = System
3 ;
GENERATE:
    superType = [System] ';';
8 System:
    'System' name = ID '{'
        selectMode = SelectMode
        detector = Detector
        descriptor = Descriptor
13     '}' ';';
SelectMode:
    'OperationMode' ':' OperationMode = ('Sw' | 'Sw-Refactored' | 'Hybrid') ';';
18 Detector:
    BRISK | ORB
;
BRISK:
23     detectorSuperType = ('BRISK') '{'
        nFast = NFastN
        threshold = Threshold
    '}' ';'
;
28 ORB:
    detectorSuperType = ('ORB') '{'
        nFast = NFastN
        threshold = Threshold
33     '}' ';'
;
NFastN:
38     'N-FASTN' ':' nValue = INT ';'
;
Threshold:
43     'Threshold' ':' thresholdValue = INT ';'
;
Descriptor:
    'BRIEF' '{'
        stringSize = StringSize
48        patchSize = PatchSize
    '}' ';'
;
StringSize:
53     'StringSize' ':' stringValue = INT ';'
;
PatchSize:
58     'PatchSize' ':' patchValue = INT ';'
;
```

4.5.2 Validation Rules

As mentioned before the grammar definition is not enough, so it is necessary to create some validation rules to guarantee that the values of the input variables are inside the predefined range, otherwise, the system would not function as expected. If there is a validation rule that was not respected then a error message is generated, and displayed to the user. In the tests section will be possible to see some of these messages.

Listing 4.104: Validation Rules

```

1  public class MyDslValidator extends AbstractMyDslValidator {
2      @Check
3      public void checkFastNValue(NFastN nValue) {
4          int aux = nValue.getNValue();
5          if(aux != 5 && aux != 9 && aux != 12)
6              error("ERROR: The value of N-FASTN can't be " + aux + ". \nN-FASTN must be 5, 9 or 12!", MyDslPackage.
7                  Literals.NFAST_N__NVALUE);
8      }
9      @Check
10     public void checkThreshold(Threshold threshold) {
11         int aux = threshold.getThresholdValue();
12         if(aux < 0 || aux > 255)
13             error("ERROR: The value of the threshold can't be " + aux + ". \nThreshold must be an integer between 0 and
14                 255!", MyDslPackage.Literals.THRESHOLD__THRESHOLD_VALUE);
15     }
16     @Check
17     public void checkStringSize(StringSize stringSize) {
18         int aux = stringSize.getStringValue();
19         if(aux != 64 && aux != 128 && aux != 256)
20             error("ERROR: The value of the String size can't be " + aux + ". \nThe String size must be a 64, 128 or
21                 256!", MyDslPackage.Literals.STRING_SIZE__STRING_VALUE);
22     }
23     @Check
24     public void checkPatchSize(PatchSize patchSize) {
25         int aux = patchSize.getPatchValue();
26         if(aux != 11 && aux != 15 && aux != 19)
27             error("ERROR: The value of the Patch size can't be " + aux + ". \nThe Patch size must be a 11, 15 or 19!",
28                 MyDslPackage.Literals.PATCH_SIZE__PATCH_VALUE);
29     }
30 }
```

4.5.3 Code generation

In what concerns to this project, there are three possible implementations:

- OpenCV
- Software Refactored
- Hybrid

These implementation need different files and different approaches were used for each one.

SW - ORB

Firstly, the OpenCV application for ORB does two simple annotations with the @ special character to replace the threshold and FASTN values. For this it is necessary to use the *compileAnnotatedCode* function in order to be

possible to process the file, replace the annotations with the values chosen by the user and generate the final code without annotations.

Listing 4.105: Generation code of OpenCV implementation for the ORB algorithm

```

1 if(e.selectMode.operationMode == "Sw"){
2     System.out.print("Starting software generation...\r\n")
3     if(e.detector.detectorSuperType == "ORB")
4     {
5         System.out.print("      -Generating ORB Detector...\r\n")
6         var file = compileAnnotatedCode(e, "Repository/ORB/SW/OpenCV_ORB.cpp", '@')
7         fsa.generateFile("src/VisionSystem/OpenCV/OpenCV_ORB.cpp", file)
8         var file1 = generateNotAnnotatedFile("Repository/ORB/SW/CMakeLists.txt")
9         fsa.generateFile("src/VisionSystem/OpenCV/CMakeLists.txt", file1)
10    }
11}

```

Listing 4.106: Annotations used in Sw-Refactored script for orb

```

1 #define FAST5      FastFeatureDetector::TYPE_5_8
2 #define FAST9      FastFeatureDetector::TYPE_9_16
3 #define FAST12     FastFeatureDetector::TYPE_7_12
4
5 int main(int argc, char** argv )
6 {
7     Mat image1; //Create two images
8     int Fast_Threshold = Threshold;
9 .
10 .
11 .
12 .
13 .
14     Ptr<FastFeatureDetector> detector = FastFeatureDetector::create(); //Create the ORB detector
15     detector->setThreshold(Fast_Threshold);
16     detector->setType(FASTN-FASTN);

```

SW - BRISK

The OpenCV application for BRISK only contains one script without annotations nor selections of blocks of code. This way to generate the OpenCV file it was generated a file with the exact content of the original one.

Listing 4.107: Generation code of OpenCV implementation

```

1 System.out.print("      -Generating BRISK Detector...\r\n")
2 var file = generateNotAnnotatedFile("Repository/BRISK/OpenCV/OpenCV_Brisk.cpp")
3 fsa.generateFile("src/VisionSystem/OpenCV/OpenCV_Brisk.cpp", file)

```

SW - BRIEF

As seen in the case of the Software segment of BRISK, the OpenCV application for BRIEF only contains one script without annotations nor selections of blocks of code. This way to generate the OpenCV file it was generated a file with the exact content of the original one.

Listing 4.108: Generation code of OpenCV implementation

```

1 System.out.print(" -Generating BRIEF Detector...\r\n")
2 varfile=generateNotAnnotatedFile("Repository/BRIEF/OpenCV/OpenCV_Brief.cpp")
fsa.generateFile("src/VisionSystem/OpenCV/OpenCV_Brief.cpp",file)

```

SW-Refactored - ORB

The code generation for the software refactored mode in the ORB algorithm also uses simple annotations, just like the software mode, but it also uses selections of blocks of code, in order to select the correct value for doing the bresenham circle, which depends on the FASTN value. The code generation and the annotated can be consulted below.

Listing 4.109: ORB SW Refactored code generation

```

1 else if(e.selectMode.operationMode == "Sw-Refactored")
2 {
3     System.out.print("Starting software-refactored generation...\r\n")
4     if(e.detector.detectorSuperType == "ORB")
5     {
6         System.out.print(" -Generating ORB Detector...\r\n")
7         var strign2 = compileAnnotatedCode(e, "Repository/ORB/Refactored/main.cpp", '@')
8         fsa.generateFile("src/VisionSystem/Refactored/main.cpp", strign2)
9         var file = generateNotAnnotatedFile("Repository/ORB/Refactored/orb.cpp")
10        fsa.generateFile("src/VisionSystem/Refactored/orb.cpp", file)
11        var file1 = generateNotAnnotatedFile("Repository/ORB/Refactored/orb.hpp")
12        fsa.generateFile("src/VisionSystem/Refactored/orb.hpp", file1)
13        var file2 = generateNotAnnotatedFile("Repository/ORB/Refactored/CMakeLists.txt")
14        fsa.generateFile("src/VisionSystem/Refactored/CMakeLists.txt", file2)
15    }
16}

```

Listing 4.110: ORB SW Refactored annotated code

```

1 orb.define_offsets(mtrx,@Threshold@);
2 .
3 .
4 .
5 @IF N-FASTN == 5@
6     circle=8;
7 @ENDIF@
8 .
9 @IF N-FASTN == 9@
10    circle=12;
11 @ENDIF@
12 .
13 @IF N-FASTN == 12@
14    circle=16;
15 @ENDIF@

```

SW-Refactored - BRISK

The second implementation, the Software Refactored, contains more files and some of them use annotations. This way was necessary to call the function *compileAnnotatedCode* to be possible to process the file, replace the annotations with the values chosen by the user and generate the final code without annotations. In this implementation are used simple annotations as well as selections of blocks of code with annotations.

Listing 4.111: Generation code of Software Refactored header file

```

System.out.print(" -Generating BRISK Detector...\r\n")
var file = compileAnnotatedCode(e, "Repository/BRISK/Sw-Refactored/BRISK.h", '@')
fsa.generateFile("src/VisionSystem/Sw-Refactored/BRISK.h", file)
file = compileAnnotatedCode(e, "Repository/BRISK/Sw-Refactored/BRISK.cpp", '@')
5 fsa.generateFile("src/VisionSystem/Sw-Refactored/BRISK.cpp", file)
file = compileAnnotatedCode(e, "Repository/BRISK/Sw-Refactored/BRISK_threads.h", '@')
fsa.generateFile("src/VisionSystem/Sw-Refactored/BRISK_threads.h", file)
file = compileAnnotatedCode(e, "Repository/BRISK/Sw-Refactored/BRISK_threads.cpp", '@')
fsa.generateFile("src/VisionSystem/Sw-Refactored/BRISK_threads.cpp", file)

```

For example, in the header file it is used a simple annotations to give the default fast value in the constructor, as the value that the user wants.

Listing 4.112: Annotations used in Sw-Refactored script

```

1 BRISK(cv::Mat img, double threshold = Threshold, NFASTN fast = FASTN-FASTN);
```

In the algorithm is necessary to have an array of transformations to be possible to access to all the positions of the Bresenham's circle. Since this circle changes with the value of the Fast (for a smaller value the circle is also smaller), then it is necessary to change the values of the transformations. For that were created three static arrays, one for each fast (5, 9 and 12), and the array is changed regarding the value of the fast.

Listing 4.113: Annotations used in Sw-Refactored script

```

IF N-FASTN == 12
point_t circleTransf_12[16] = {{0, -3}, {1, -3}, {2, -2}, {3, -1},
4                                {3, 0}, {3, 1}, {2, 2}, {1, 3},
                                {0, 3}, {-1, 3}, {-2, 2}, {-3, 1},
                                {-3, 0}, {-3, -1}, {-2, -2}, {-1, -3}};
ENDIF
IF N-FASTN == 9
point_t circleTransf_9[12] = {{0, -2}, {1, -2}, {2, -1},
9                                {2, 0}, {2, 1}, {1, 2},
                                {0, 2}, {-1, 2}, {-2, 1},
                                {-2, 0}, {-2, -1}, {-1, -2}};
ENDIF
IF N-FASTN == 5
14 point_t circleTransf_5[8] = {{0, -1}, {1, -1},
                                {1, 0}, {1, 1},
                                {0, 1}, {-1, 1},
                                {-1, 0}, {-1, -1}};
ENDIF
19
BRISK::BRISK(cv::Mat img, double threshold, NFASTN fast)
{
    ...
24
    IF N-FASTN == 12
        this->circleTransf = circleTransf_12;
    ENDIF
29
    IF N-FASTN == 9
        this->circleTransf = circleTransf_9;
    ENDIF
}
IF N-FASTN == 5

```

```
34     this->circleTransf = circleTransf_5;
      ENDIF
}
```

In the final code, these annotations disappear and will look like the following code.

Listing 4.114: Annotations used in Sw-Refactored script

```
point_t circleTransf_9[12] = {{0, -2}, {1, -2}, {2, -1},
                               {2, 0}, {2, 1}, {1, 2},
                               {0, 2}, {-1, 2}, {-2, 1},
                               {-2, 0}, {-2, -1}, {-1, -2}};

BRISK::BRISK(cv::Mat img, double threshold, NFASTN fast)
{
    ...
9   this->circleTransf = circleTransf_9;
}
```

For this example the fast used was 9.

SW-Refactored - BRIEF

As seen in the segments above some software refactored code sections need to suffer changes in order to generate the code in accordance with the user inputs. In description Sw-Refactored two functions are used: *compileAnnotatedCode* to analyse the file and replace the annotations with the values chosen by the user, and *generateNotAnnotatedFile* to compile the not annotated code present in the repository.

Listing 4.115: Generation code of Software Refactored header file

```
System.out.print("  -Generating BRIEF Detector...\r\n")
var anot_code = compileAnnotatedCode(e, "Repository/BRIEF/SW-Refactored/brief.cpp", '@')
fsa.generateFile("src/VisionSystem/Sw-Refactored/Brief" + ".c++", anot_code)
5
var code = generateNotAnnotatedFile("Repository/BRIEF/SW-Refactured/brief.h")
fsa.generateFile("src/VisionSystem/Sw-Refactored/Brief" + ".h", code)

code = generateNotAnnotatedFile("Repository/BRIEF/SW-Refactured/main.cpp")
fsa.generateFile("src/VisionSystem/Sw-Refactored/BriefMain" + ".c++", code)
10
```

One example of the annotations made in the code is the following:

Listing 4.116: Example of annotations used in SW-Refactored Code

```
_briefSize = @StringSize@
_patchSize = @PatchSize@

std::string file = "BriefSize_" + std::to_string(_briefSize) + "_PatchSize_" + std::to_string(_patchSize) ".txt";
```

The user selectd the string size of the binary tests and the patch size of the system in order to tune the algorithm in accordance with the requirements of the project, or even the resources available. This will allow the system to select the correct .txt file with the right pairs of pixels that represent the description of a keypoint, according to the sampling geometry.

SW-Refactored - Matching

Since the refactored software performed by matching does not need to be annotated, only the normal simple code generation functions were used.

Listing 4.117: Matching sw-Refactored code generation

```

1 var file = generateNotAnnotatedFile("Repository/MATCHING/Sw-Refactored/BruteForce.cpp")
fsa.generateFile("src/VisionSystem/Sw-Refactored/BruteForce.cpp", file)

6 file = generateNotAnnotatedFile("Repository/MATCHING/Sw-Refactored/BruteForce.h")
fsa.generateFile("src/VisionSystem/Sw-Refactored/BruteForce.h", file)

file = generateNotAnnotatedFile("Repository/MATCHING/Sw-Refactored/Display.cpp")
fsa.generateFile("src/VisionSystem/Sw-Refactored/Display.cpp", file)

11 file = generateNotAnnotatedFile("Repository/MATCHING/Sw-Refactored/Display.h")
fsa.generateFile("src/VisionSystem/Sw-Refactored/Display.h", file)

```

Hybrid - ORB

The code corresponding to the hybrid mode for the ORB detector was annotated according to the two previously mentioned parameters. Furthermore, the code follows a modular approach, so by changing the parameters in the wrapper, all the rest of the code will be synthesised based on those parameters, thus only needing two simple annotations.

Listing 4.118: ORB Hybrid code generation

```

else if(e.selectMode.operationMode == "Hybrid")
{
    System.out.print("Starting Hybrid generation...\r\n")
    if(e.detector.detectorSuperType == "ORB")
    {
        System.out.print("      -Generating ORB Detector...\r\n")

8     System.out.print("\t-Generating Centroid Calculation...\r\n")
var file = generateNotAnnotatedFile("Repository/ORB/Hybrid/centroid_calculation.sv")
fsa.generateFile("src/VisionSystem/Hybrid/centroid_calculation.sv", file)

        System.out.print("\t-Generating Contigity Count...\r\n")
13    var file1 = generateNotAnnotatedFile("Repository/ORB/Hybrid/contg_count.v")
fsa.generateFile("src/VisionSystem/Hybrid/contg_count.v", file1)

        System.out.print("\t-Generating db classifier...\r\n")
18    var file2 = generateNotAnnotatedFile("Repository/ORB/Hybrid/db_classifier.sv")
fsa.generateFile("src/VisionSystem/Hybrid/db_classifier.sv", file2)

        System.out.print("\t-Generating design wrapper...\r\n")
23    var file3 = generateNotAnnotatedFile("Repository/ORB/Hybrid/design_2_wrapper.v")
fsa.generateFile("src/VisionSystem/Hybrid/design_2_wrapper.v", file3)

        System.out.print("\t-Generating detection scoring...\r\n")
28    var file4 = generateNotAnnotatedFile("Repository/ORB/Hybrid/detection_scoring.sv")
fsa.generateFile("src/VisionSystem/Hybrid/detection_scoring.sv", file4)

        System.out.print("\t-Generating Patcher...\r\n")
var file5 = generateNotAnnotatedFile("Repository/ORB/Hybrid/Patcher.sv")

```

```

        fsa.generateFile("src/VisionSystem/Hybrid/Patcher.sv", file5)

        System.out.print("\t-Generating Sliding memory...\r\n")
33     var file6 = generateNotAnnotatedFile("Repository/ORB/Hybrid/sliding_memory.v")
        fsa.generateFile("src/VisionSystem/Hybrid/sliding_memory.v", file6)

        System.out.print("\t-Generating wrapper patcher...\r\n")
        var file7 = compileAnnotatedCode(e, "Repository/ORB/Hybrid/wrapper_patcher.sv", '$')
38     fsa.generateFile("src/VisionSystem/Hybrid/wrapper_patcher.sv", file7)
    }
}

```

Hybrid - BRISK

The Hybrid implementation of BRISK algorithm uses more files and some of them have variability points and others don't. For the ones without variability points it is only necessary to chose the source file.

Listing 4.119: Generation code of Hybrid implementation

```

System.out.print("      -Generating BRISK Detector...\r\n")
//detection
//classifier
System.out.print("\t-Generating Detection...\r\n")
5  var file = compileAnnotatedCode(e, "Repository/BRISK/Hybrid/classifier.v", '@')
    fsa.generateFile("src/VisionSystem/Hybrid/classifier.v", file)

    file = compileAnnotatedCode(e, "Repository/BRISK/Hybrid/classifier_controller.v", '@')
    fsa.generateFile("src/VisionSystem/Hybrid/classifier_controller.v", file)

10   file = compileAnnotatedCode(e, "Repository/BRISK/Hybrid/image_slice.v", '@')
    fsa.generateFile("src/VisionSystem/Hybrid/image_slice.v", file)

//scoring
15   System.out.print("\t-Generating Scoring...\r\n")
    file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/adder.v")
    fsa.generateFile("src/VisionSystem/Hybrid/adder.v", file)
    file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/score_fast" + e.detector.NFast.NValue + ".v")
    fsa.generateFile("src/VisionSystem/Hybrid/score_fast.v", file)
20

//contiguity test
System.out.print("\t-Generating Contiguity Test...\r\n")
file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/contig_fast"+e.detector.NFast.NValue + ".v")
fsa.generateFile("src/VisionSystem/Hybrid/contig_fast.v", file)

25

//nms
System.out.print("\t-Generating Non Maximum Suppression...\r\n")
file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/defines.v")
    fsa.generateFile("src/VisionSystem/Hybrid/nmsDefines.v", file)
30   file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/h_to_v.v")
    fsa.generateFile("src/VisionSystem/Hybrid/nms_h_to_v.v", file)
    file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/nms_horizontal.v")
    fsa.generateFile("src/VisionSystem/Hybrid/nms_horizontal.v", file)
    file = generateNotAnnotatedFile("Repository/BRISK/Hybrid/nms_vertical.v")
    fsa.generateFile("src/VisionSystem/Hybrid/nms_vertical.v", file)
35

```

For example, in the scoring phase, if the value of the fast chosen by the user is 9, then the source file that will be generated is the "score_fast9.v", but if the values of fast changes to 12 then the source script that would be used is "score_fast12.v". The files with variability points have annotations, so it is necessary to use the function *compileAnnotatedCode* in order to process them. For the BRISK implementation were necessary more variables

besides the N-FASTN and threshold. The variables were the *Depth* and the *CircleSize* that vary according to the fast value. The *Depth* is the number of blocks m different lines of the image that will be read. For FAST12 it is 7, for FAST9 is 5 and for FAST5 is 3.

Listing 4.120: How the depth is obtained

```
var int depth = (system.detector.NFast.NValue / 2) + 1
```

Listing 4.121: Variable Depth annotation usage

```
parameter DATA_FETCH_DEPTH = Depth
```

The *CircleSize* is the size of the Bersenham Circle. For FAST12 it is 16, for FAST9 is 12 and for FAST5 is 8.

Listing 4.122: How the circle size is obtained

```
var int circleSize = system.detector.NFast.NValue + 3
if(circleSize % 2 != 0)
    circleSize++
```

Listing 4.123: Variable CircleSize annotation usage

```
2 module classifier
  #(
    parameter CIRCLE_SIZE = CircleSize,
    parameter WIDTH = CIRCLE_SIZE
  )
```

Hybrid - BRIEF

The code generation for Hybrid mode in the BRIEF algorithm uses the selection of blocks of code and annotations, giving the user the ability to customise the following sistem parameters: String Size and Patch Size. Below can be analysed the generation of the files in the repository, synthesised based on those parameter values.

Listing 4.124: BRIEF Hybrid code generation

```
System.out.print("      -Generating BRIEF Descriptor ...\\r\\n")

var code = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/BRIEF.v")
fsa.generateFile("src/VisionSystem/Hybrid/BRIEF.v", code)
5 code = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/fifo_sync.v")
fsa.generateFile("src/VisionSystem/Hybrid/fifo_sync.v", code)

System.out.print("      -Generating Patch...\\r\\n")

10 var anot_code = compileAnnotatedCode(e, "Repository/BRIEF/Hybrid/imageportionmem.v", '@')
fsa.generateFile("src/VisionSystem/Hybrid/imageportionmem.v", anot_code)
anot_code = compileAnnotatedCode(e, "Repository/BRIEF/Hybrid/imageportionaccess.v", '@')
fsa.generateFile("src/VisionSystem/Hybrid/imageportionaccess.v", anot_code)
code = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/patchlinecomputation.v")
15 fsa.generateFile("src/VisionSystem/Hybrid/patchlinecomputation.v", code)
code = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/RF.v")
```

```

fsa.generateFile("src/VisionSystem/Hybrid/RF.v", code)

System.out.print("      -Generating Binary Tests...\r\n")
20
anot_code = compileAnnotatedCode(e, "Repository/BRIEF/Hybrid/binary_test.v", '@')
fsa.generateFile("src/VisionSystem/Hybrid/binary_test.", annot_code)
anot_code = compileAnnotatedCode(e, "Repository/BRIEF/Hybrid/brief.v", '@')
fsa.generateFile("src/VisionSystem/Hybrid/brief.v", annot_code)
25
code = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/verilogwrapper.v")
fsa.generateFile("src/VisionSystem/Hybrid/verilogwrapper.v", code)
var file = generateNotAnnotatedFile("Repository/BRIEF/Hybrid/pair_coord_" + e.descriptor.stringSize.stringValue + "_" + e.descriptor.
    patchSize.patchValue + ".vh")
fsa.generateFile("src/VisionSystem/Hybrid/pair_coord_" + e.descriptor.stringSize.stringValue + "_" + e.descriptor.patchSize.
    patchValue + ".vh", file)

```

As we can see in the pair generation module, the choice between all the different combinations for the values string size and patch size has to be made in order to select the not annotated pair generation file that is intended to be generated. This optimizes the generation of secondary files that are used in the main algorithm since it is possible to generate only the needed file for the case, which means that only one of the 9 pair coordinates files is generated in order to be used in the final code, reducing the amount of generated files.

For example, if the value of the "StringSize" is 64, and the value of "PatchSize" is 11, then the source file that will be generated is the "pair_coord_64_11.vh", using the function *generateNotAnnotatedFile*. The other files with variability points have annotations, so it is necessary to use the function *compileAnnotatedCode* to process them.

Listing 4.125: BRIEF Hybrid conditional code segment

```

1      @IF PatchSize == 11@
//11x11
assign new_data = (rst) ? 0 :
(t_rdy && extended_address[1:0] == 2'b00) ? {memory[address], memory[address+1], memory[address+2], memory[address
+3][7:0]} :
(t_rdy && extended_address[1:0] == 2'b01)
(...)
@ENDIF@

11     @IF PatchSize == 15@
//15x15
assign new_data = (rst) ? 0 :
(t_rdy && extended_address[1:0] == 2'b00) ? {memory[address], memory[address+1], memory[address+2], memory[address
+3][23:0]} :
(t_rdy && extended_address[1:0] == 2'b01)
(...)
@ENDIF@

16     @IF PatchSize == 19@
//19x19
assign new_data = (rst) ? 0 :
(t_rdy && extended_address[1:0] == 2'b00) ? {memory[address], memory[address+1], memory[address+2], memory[address
+3],
(...)}
@ENDIF@

```

In the listing above is represented a code segment where the annotations contain conditions, that according to the variability points values will result in different blocks of generated code in the final source code. This type of code annotation allows for a higher flexibility of the code modularity of the system.

Hybrid - Matching

In what concerns to the matching hybrid implementation the files don't contain any annotation nor selection of blocks of code, this way, it is only necessary to generate the files that are in the repository. For that is used the function `generateNotAnnotatedFile` for all the files.

Listing 4.126: Generation code of Matching Hybrid implementation

```

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/Controller_1.v")
2 fsa.generateFile("src/VisionSystem/Hybrid/Controller_1.v", file)

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/counter.v")
fsa.generateFile("src/VisionSystem/Hybrid/counter.v", file)

7 file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/distance.v")
fsa.generateFile("src/VisionSystem/Hybrid/distance.v", file)

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/distance_stage.v")
fsa.generateFile("src/VisionSystem/Hybrid/distance_stage.v", file)
12

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/Dual_port.v")
fsa.generateFile("src/VisionSystem/Hybrid/Dual_port.v", file)

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/Match.v")
17 fsa.generateFile("src/VisionSystem/Hybrid/Match.v", file)

file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/Matching_Core.v")
fsa.generateFile("src/VisionSystem/Hybrid/Matching_Core.v", file)

22 file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/matching_wrapper.v")
fsa.generateFile("src/VisionSystem/Hybrid/matching_wrapper.v", file)

    file = generateNotAnnotatedFile("Repository/MATCHING/Hybrid/stage1.v")
fsa.generateFile("src/VisionSystem/Hybrid/stage1.v", file)

```

To make sure that the repository is organised, was created a folder for each algorithm. In each of this folders are three folders with the possible implementations (OpenCV, SW-Refactored and Hybrid). Besides that when it is time to generate the files, it is created a folder with the name of the implementation that is being used, this way will be possible to generate the three implementation one at a time and then compare them.

Even though the names of the files in the hybrid implementation, are different regarding the value of the FAST chosen by the user, the generated scripts will always have the same name for all the values of the FAST. In such a way, the final code it is easier to compile, since the names don't change.

Finally, after all the implementation is generated a Runnable JAR file named `generator.jar`. This file will receive the user script and generate all the files.

4.6 Profiling

This is the code that will be under testing. An implementation of the ORB Algorithm in C++.

4.6.1 ORB Detector

Listing 4.127: ORB OpenCV Detector

```

1 int main(int argc, char** argv)
2 {
3     Mat resized;
4     Mat img_1 = imread("lion.png", IMREAD_GRAYSCALE);
5
6     resize(img_1, resized, Size(320, 240), (0, 0), (0, 0), 1);
7     std::vector<KeyPoint> keypoints_1;
8
9     Ptr<FeatureDetector> detector = ORB::create(10000, 1.2f, 8, 31, 0, 2, ORB::FAST_SCORE, 31, 20);
10    detector->detect(resized, keypoints_1);
11
12    return 0;
13 }
14 }
```

It is a simple code that reads an image, resizes it to the standard format, that is used in the global project (320x240), and uses the function of the openCV ORB detect, using the FAST score, to detect the keypoints.

4.6.2 ORB Descriptor

Listing 4.128: ORB OpenCV detection

```

1 #include <iostream>
2 #include "opencv2/opencv_modules.hpp"
3 #include <opencv2/core/core.hpp>
4 #include <opencv2/features2d/features2d.hpp>
5 #include <opencv2/xfeatures2d/xfeatures2d.hpp>
6 #include <opencv2/highgui/highgui.hpp>
7
8 using namespace std;
9 using namespace cv;
10
11 int main ( int argc, char** argv )
12 {
13     Mat img_1 = imread ("1.png", IMREAD_COLOR );
14     Mat img_2 = imread ("2.png", IMREAD_COLOR );
15
16     std::vector<KeyPoint> keypoints_1, keypoints_2;
17     Mat descriptors_1, descriptors_2;
18     Ptr<FeatureDetector> detector = ORB::create();
19     Ptr<DescriptorExtractor> descriptor = ORB::create();
20
21     Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create ( "BruteForce-Hamming" );
22
23     detector->detect ( img_1,keypoints_1 );
24     detector->detect ( img_2,keypoints_2 );
25
26     descriptor->compute ( img_1, keypoints_1, descriptors_1 );
27     descriptor->compute ( img_2, keypoints_2, descriptors_2 );
28
29     Mat outimg1;
30     drawKeypoints( img_1, keypoints_1, outimg1, Scalar::all(-1), DrawMatchesFlags::DEFAULT );
31     imshow("ORB",outimg1);
```

```

    vector<DMatch> matches;
    //BFMatcher matcher ( NORM_HAMMING );
36    matcher->match ( descriptors_1, descriptors_2, matches );

    Mat img_match;
    drawMatches ( img_1, keypoints_1, img_2, keypoints_2, matches, img_match );
    imshow ( "Matching", img_match );
41    waitKey(0);
    return 0;
}

```

This code is divided in three parts, detection, description and matching. First step consists in creating the three objects needed, the orb detector, the orb descriptor and the brute force matcher. After this, the detect, compute and match functions, from OpenCV are responsible to detect, describe and match the keypoints. The focus will be in the Orb descriptor and the compute function.

4.6.3 Open-CV BRISK Detector

Open-CV Library Access Short Example Code

Listing 4.129: BRISK OpenCV Detection

```

int main(int argc, char** argv) {
2
    Mat img1 = imread("C:/Users/joaop/Desktop/box.png", IMREAD_GRAYSCALE);
    Mat img2 = imread("C:/Users/joaop/Desktop/box_in_scene.png", IMREAD_GRAYSCALE);

    if (!img1.data || !img2.data) {
        cout << "Image not found!!!" << endl;
        return -1;
    }

    imshow("img1_box", img1);
12   imshow("img2_scene", img2);

    // Use the Brisk algorithm to detect feature points
    vector<KeyPoint> keypoint_obj;
    vector<KeyPoint> keypoint_scene;
17
    Ptr<Feature2D> detect = BRISK::create();
    Mat descriptor_obj, descriptor_scene;
    // Detect and calculate the descriptor
    for (;;) {
22        if (n == 10000) {
            detectionexecutiontime.close();
            return -1;
        }
        n++;
        printf("iteration %d\r\n", n);

        double t1 = getTickCount(); //calculate run time
        detect->detectAndCompute(img1, Mat(), keypoint_obj, descriptor_obj);
        detect->detectAndCompute(img2, Mat(), keypoint_scene, descriptor_scene);
32        double t2 = getTickCount();
        double t = (t2 - t1) * 1000 / getTickFrequency();
    }
}

```

```

    detectionexecutiontime << t << endl;
    printf("BRISK execution time is (ms):%f\r\n", t);
37 }
//Match the descriptor and show the perspective matrix code goes here.
}

```

About the feature detection, there are mainly five steps to be done:

1. Buil the Scale Space → Gaussian Pyramid construction;
2. Feature Point detection;
3. FAST9-16 looking for feature points → 9 consecutive points greater than or less than the current value are considered feature points;
4. feature point positioning;
5. keypoint descriptor.

The feature point finding must maintain rotation invariance, scale invariance and illumination intensity invariance. With that thought in mind, the following code was created. This section of the code loads two test images and reads them in grey scale. To control the load of the image, an if statement is used. The images are similar. The difference is the scene in which they are set. The image can also be loaded already in greyscale. Since the algorithm that is wanted to execute the project is the BRISK algorithm, it was implemented the creation of BRISK to detect feature points. Now, regarding the detection feature of BRISK, two equal functions were used to detect the feature points and compute the descriptors. These lines are the ones that will be emphasised on the Results topic. The rest of the code is represented in the Appendices section since it is not what is going to be analysed. However this part of the code is really important for the proper development of the algorithm and all tests were made with this code integrated.

Batch File to run Previous Code

To profile the code and obtain reliable information, the code must be executed several times. A way to simplify the gathering of information with the OProfile tool, a batch file is in need to be created.

With that in mind, a batch file that can run the code the amount of times specified by the user was made and is the following:

```

1 #!/usr/bin/bash
  if [ $# -eq 2 ]; then
    sudo rm -rf oprofile_data/
    for i in $(seq 1 $2)
    do
6      echo interation $i
      sudo sudo perf --append --event cpu_clk_unhalted:$3 ./$1
    done
    oreport --demangle=smart --symbols $1
  fi

```

When executing the batch file, the user must provide three parameters. The executable file to profile, the number of times that will execute that same file, and the count number for the CPU_CLK_UNHALTED event. Then, in a loop, OProfile, with the command "perf" alongside the option "--append" (Old profile data in 'current' is left in place and new profile data will be added to it, and the 'previous' profile (if one existed) will remain untouched.). After the profiling is done, a report must be presented with the necessary information to interpret the code performance. For that, the

"opreport" is utilized. Alongside "opreport", the options "-demangle=smart" and "-symbols BRISK_exe" were used so that the user has a symbol summary for a single application including libraries, being the applications BRISK_exe.

4.7 Linux Image Creation and Memory Reservation

4.7.1 Linux Image Creation

The creation of the linux image for the Zybo z7-10 board is divided into several phases. Unlike other boards, in order to run linux on this board it is necessary to have a hardware description that is provided via the Vivado project bitstream.

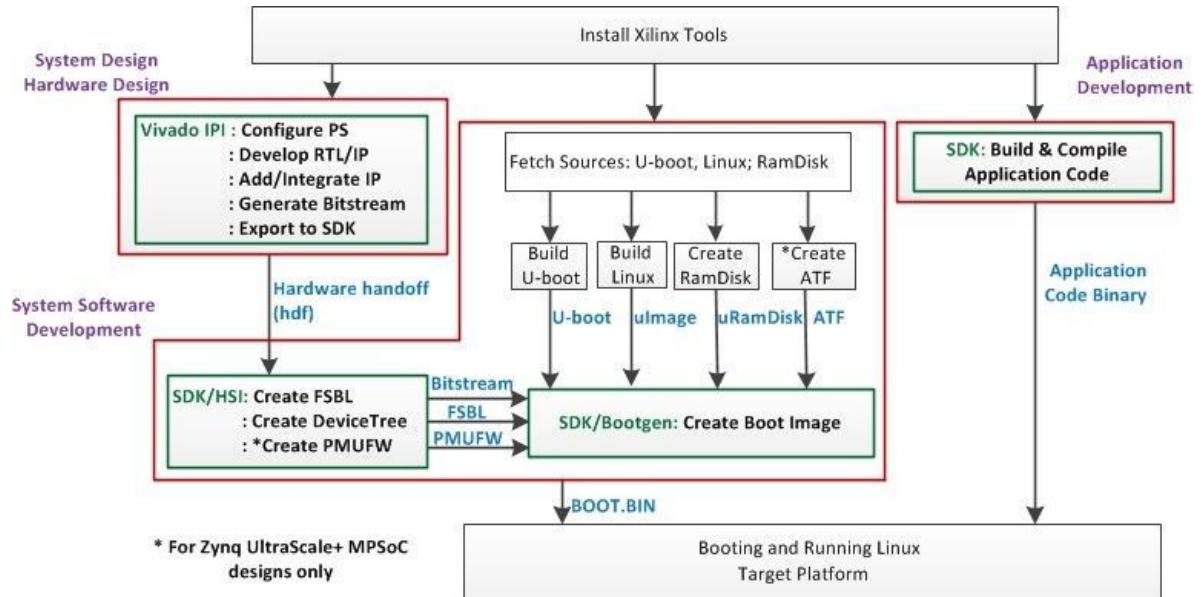


Figure 4.7.1: Steps to build an image linux for Zybo z7-10

The several components are depicted in figure 4.7.1. For the construction of all these steps, the petalinux tool was used in order to build, deploy, and boot embedded Linux. PetaLinux includes tools to customize the boot loader, Linux kernel, file system, libraries and system parameters.

To create an image for the board using petalinux, it is necessary to create a project using the board's BSP and perform all the necessary customizations. These configurations need to populate the filesystem with the packages needed for the project which are opencv and boost library. Finally, the image is generated using bitstream and FSBL. After creating the image, a SD card is used, dividing it into two partitions. The first partition is the boot partition that

contains the system files used to start the operating system. The second partition is the ROOT partition that will contain the filesystem used. In the figure 4.7.2 and 4.7.3 it is possible to see this two partitions and the respective components.

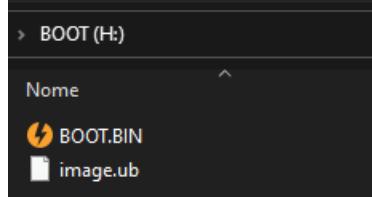


Figure 4.7.2: Boot Components for linux image

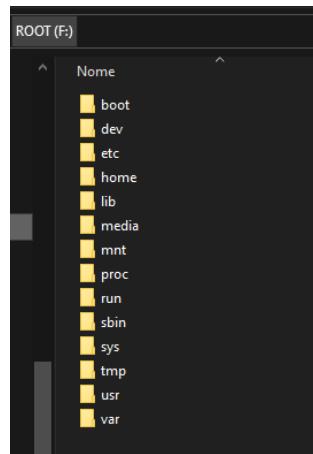


Figure 4.7.3: Root Components for linux image

4.7.2 Memory Allocation

One of the common requirements on Xilinx Zynq SoC based systems is to reserve memory regions for special usage, excluding it from the usage of Linux kernel and making it available only for a custom device driver.

This can be achieved following one of four methods: via device-tree, moving the kernel base address, changing the address where the u-boot loads the image or through the DMA API.

Considering petalinux is being used, to generate the boot image, the easiest way to reserve memory is to change the kernel base address in the default configuration menu, as shown in 4.7.4, issuing the command `petalinux-config`, then navigating through *Subsystem AUTO Hardware Settings -> Memory Settings -> Kernel base address*.

```

## Loading kernel from FIT Image at 10000000 ...
Using 'conf@1' configuration
Verifying Hash Integrity ... OK
Trying 'kernel@0' kernel subimage
  Description: Linux Kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x100000d4
  Data Size: 3813800 Bytes = 3.6 MiB
  Architecture: ARM
  OS: Linux
Load Address: 0x00008000
Entry Point: 0x00008000
hash algo: sha1
Hash value: 49363739e81d59417c3105246a8b3ef196e5e60
Verifying Hash Integrity ... OK

Primary Memory (ps7_ddr_0) --->
(0x0) System memory base address
(0x40000000) System memory size
(0xc800000) kernel base address
(0x400000) u-boot text base address offset to memory b

## Loading kernel from FIT Image at 10000000 ...
Using 'conf@1' configuration
Verifying Hash Integrity ... OK
Trying 'kernel@0' kernel subimage
  Description: Linux Kernel
  Type: Kernel Image
  Compression: uncompressed
  Data Start: 0x100000d4
  Data Size: 3813800 Bytes = 3.6 MiB
  Architecture: ARM
  OS: Linux
Load Address: 0x00c80000
Entry Point: 0x00c80000
hash algo: sha1
Hash value: 49363739e81d59417c3105246a8b3ef196e5e60
Verifying Hash Integrity ... sha1+ OK

```

Figure 4.7.4: Memory reservation through changing kernel base address

After this, it is also required to change the `XILINX_RAM_START` definition in the `xparameters.h` file to the new initial value and change the memory definition in the device tree `system.dtsi` file as shown in 4.130. The first value is the start address of the device, and the second is the size.

Listing 4.130: Device tree configuration

```

memory {
    device_type = "memory";
    reg = <0xC8000000 0x40000000>;
}

```

Finally, in order to execute u-boot from the 32 bit proxy address space a final change has to be made, in the `Config.mk` file in the `CONFIG_SYS_TEXT_BASE` definition.

4.8 Register-based Configuration

The Register File for the Register-based Configuration was created with resource to Vivado's AXI4 Peripheral generation feature. Then the automatically generated code was extended to include a second interface that would allow the Programmable Logic to access the values in the registers directly. This is shown in listing 4.131.

Listing 4.131: Register File - Direct Access Interface

```

1 module config_registers_v1_0_S00_AXI #
  (
    /* ... */
    /* Automatically generated parameters */
  )

```

```

6   (
7     // Users to add ports here
8     input      [0:0]  STATE_RUNNING,
9     output     [8:0]  CONF_WIDTH,
10    output     [7:0]  CONF_HEIGHT,
11    output     [0:0]  CONF_ENABLE,
12    output     [7:0]  CONF_THRESHOLD,
13    output     [31:0] MEM_BASE_ADDR,
14    output     [31:0] MEM_TOP_ADDR,
15    // User ports ends
16
17    /* ... */
18    /* AXI Interface */
19  )
20
21 /* ... */

```

After that, the REG_STATE register, in address 0x01 of the register table (fig. 4.8.1) was configured to respond to an input of the secondary interface instead of the AXI interface shared with the Processing System. This is so its first bit (coded RUNNING) may be changed by the Programmable Logic directly, as it is a read-only bit that indicates whether the operation is running or not.

Listing 4.132: Register File - RUNNING bit Interface Redirection

```

always @(*(posedge S_AXI_ACLK)
begin
  /* ... */
  /* Automatically generated code */
  if (slv_reg_wren)
    begin
      case (axi_awaddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB])
        /* ... */
        /* Automatically generated code */
        4'h1:      // STATE
          for (byte_index = 0; byte_index <= (C_S_AXI_DATA_WIDTH/8)-1; byte_index = byte_index+1)
            if (S_AXI_WSTRB[byte_index] == 1) begin
              // Respective byte enables are asserted as per write strobes
              // Slave register 1
              // slv_reg1[(byte_index*8) +: 8] <= S_AXI_WDATA[(byte_index*8) +: 8];
              slv_reg1['R_STATE_RUNNING] = STATE_RUNNING;
            end
        /* ... */
        /* Automatically generated code */

```

To finalise, the mapping of the registers to the outputs was done, as shown in listing 4.133.

Listing 4.133: Register File - Register Mapping

```

1  /* ... */
2    assign CONF_WIDTH      = slv_rego['R_CONF_WIDTH];
3    assign CONF_HEIGHT     = slv_rego['R_CONF_HEIGHT];
4    assign CONF_ENABLE     = slv_rego['R_CONF_ENABLE];
5    assign CONF_THRESHOLD  = slv_rego['R_CONF_THRESHOLD];
6
7    assign MEM_BASE_ADDR   = slv_reg2['R_MEM_BASE_ADDR];
8    assign MEM_TOP_ADDR    = slv_reg3['R_MEM_TOP_ADDR];
9
10   /* ... */

```

		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	REG_CTRL																		WIDTH[26:18]		HEIGHT[17:9]		EN		THRESHOLD[7:0]								
0x01	REG_STATE																								RUN								
0x02	REG_BASE_ADDR																								MEM_BASE_ADDR[31:0]								
0x03	REG_TOP_ADDR																								MEM_TOP_ADDR[31:0]								

Figure 4.8.1: Register Map

The not fully integrated schematic of the integration of the Register-based Control System with the Processing System and the BRISK Classifier is shown in fig. 4.8.2.

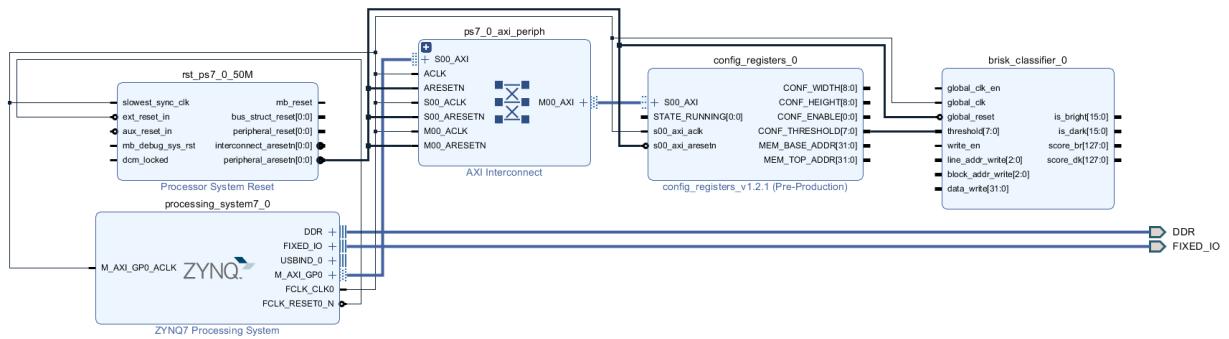


Figure 4.8.2: Block Design of the Integration of the Register-based Control System with the Processing System

5 | Testing and Result Assessment

This section will present all the unit and integration tests performed on the system's modules, ending with a final evaluation to compare actual performance with potential or desired performance for the system, mainly concerning software and hardware. Additionally, this chapter will also approach the project in terms of the behavioural simulations of the different modules, culminating in the simulation of the system as a whole. Note that, the ?? contains some the simulation snippets that will be explained and referenced throughout this section.

5.1 Generation

This section will portray the tests and results in terms of code generation and validation rules. As mentioned in the validation rules section, every time the user gives a value to a variable out of the predefined values, an error message is generated. In this section, were tried different scripts with some different types of errors to test if the system can find all of them and alert the user with an understandable message in real time.

In order to execute the tests described above was created a simple script in python that creates random scripts with a high probability of having syntactic errors. This script creates a file with the extension 'MyDsl' in order to be possible to be interpreted as a user script.

In the figures above are some of the error messages that are presented to the user when the grammatical rules are not followed. All the errors written in the script were detected.

The screenshot shows a code editor with a syntax-highlighted script. A red error icon is positioned next to the word 'Ope'. A tooltip box is overlaid on the error icon, containing the text: 'extraneous input 'visionSystem' expecting '{' Press 'F2' for focus'. The script content includes:

```
system visionSystem{  
    Ope  
    BRIEFL  
    Threshold: 23;  
};  
BRIEFL {  
    StringSize: 256;  
    PatchSize: 11;  
};
```

Figure 5.1.1: Error: Wrong object name (should be "System")

The screenshot shows a code editor with a syntax-highlighted script. A red error icon is positioned next to the word 'Software'. A tooltip box is overlaid on the error icon, containing the text: 'no viable alternative at input 'Software'' Press 'F2' for focus'. The script content includes:

```
System visionSystem{  
    OperationMode: Software;  
    BRISK {  
        N-FASTN: 5;  
        Threshold:  
    };  
    BRIEFL {  
        StringSize: 256;  
        PatchSize: 11;  
    };
```

Figure 5.1.2: Error: Wrong Operation mode (should be "Sw")

The screenshot shows a code editor with a syntax-highlighted script. A red error icon is positioned next to the word 'test'. A tooltip box is overlaid on the error icon, containing the text: 'no viable alternative at input 'test'' Press 'F2' for focus'. The script content includes:

```
System visionSystem{  
    OperationMode: Hybrid;  
    test {  
    };  
    BRIEFL {  
        StringSize: 256;  
        PatchSize: 11;  
    };  
};
```

Figure 5.1.3: Error: Wrong descriptor name (should be BRISK or ORB)

The screenshot shows a code editor with a syntax-highlighted script. A red error icon is positioned next to the word 'test'. A tooltip box is overlaid on the error icon, containing the text: 'mismatched input 'test' expecting 'BRIEFL'' Press 'F2' for focus'. The script content includes:

```
System visionSystem{  
    OperationMode: Hybrid;  
    BRISK {  
        N-FASTN: 5;  
        Threshold: 23;  
    };  
    test {  
    };  
};
```

Figure 5.1.4: Error: Wrong descriptor name (should be BRIEFL)

```

System visionSystem{
    OperationMode: Hybrid;
    BRISK {
        test: 5;
    };
    BRIEF {
        StringSize: 256;
        PatchSize: 11;
    };
};

```

The code editor highlights the 'test' variable in the BRISK block with a red error icon. A tooltip box says: "mismatched input 'test' expecting 'N-FASTN'" and "Press 'F2' for focus".

Figure 5.1.5: Error: Wrong parameter (should be N-FASTN)

```

System visionSystem{
    OperationMode: Hybrid;
    BRISK {
        N-FASTN: 5.2;
        Threshold: 23;
    };
    BRIEF {
        StringSize: 256;
        PatchSize: 11;
    };
};

```

The code editor highlights the decimal point in the 'N-FASTN' value with a red error icon. A tooltip box says: "mismatched input '.' expecting ;" and "Press 'F2' for focus".

Figure 5.1.6: Error: Wrong type (should be integer)

```

System visionSystem{
    OperationMode: Hybrid;
    BRISK {
        N-FASTN: 5;
        Threshold: 23.3;
    };
    BRIEF {
        StringSize: 2;
        PatchSize: 11;
    };
};

```

The code editor highlights the closing brace of the BRIEF block with a red error icon. A tooltip box says: "mismatched input ';' expecting ;" and "Press 'F2' for focus".

Figure 5.1.7: Error: Wrong type (should be integer)

```

System visionSystem{
    OperationMode: Hybrid;
    BRISK {
        N-FASTN: 5;
        Threshold: 23;
    };
    BRIEF {
        StringSize: 100;
        PatchSize: 1;
    };
};

```

The code editor highlights the 'StringSize' value of 100 with a red error icon. A tooltip box says: "ERROR: The value of the String size can't be 100. The String size must be a 64, 128 or 256!" and "Press 'F2' for focus".

Figure 5.1.8: Error: Wrong StringSize value (should be 64, 128 or 256)

```

System visionSystem{
    OperationMode: Hybrid;
    BRISK {
        N-FASTN: 5;
        Threshold: 23;
    };
    BRIEF {
        StringSize: 256;
        PatchSize: 9;
    };
};

```

The code editor highlights the 'PatchSize' value of 9 with a red error icon. A tooltip box says: "ERROR: The value of the Patch size can't be 9. The Patch size must be a 11, 15 or 19!" and "Press 'F2' for focus".

Figure 5.1.9: Error: Wrong PatchSize value (should be 11, 15 or 19)

```

System visionSystem{
    OperationMode: Hybrid;
    ORB {
        N-FASTN: 7;
        Threshold: 23;
    };
    BRIEF {
        StringSize: 256;
        PatchSize: 11;
    };
};

```

The code editor highlights the 'N-FASTN' value of 7 with a red error icon. A tooltip box says: "ERROR: The value of N-FASTN can't be 7. N-FASTN must be 5, 9 or 12!" and "Press 'F2' for focus".

Figure 5.1.10: Error: Wrong N-FASTN value (should be 5, 9 or 12)

To test the generation of the code firstly were used simple test scripts with different annotations and then it was used code developed by our colleges responsible from the various stages of the computer vision algorithm. In the end all these tests were successful.

In the figure 5.1.11, it is possible to see how to run the Runnable JAR file created at the end of the implementation, giving as input the file *sample.MyDsI* that is an example of a user script.

It is also possible to see the steps of the generation of the code. The implementation chosen is the *Hybrid*, and after the start of the generation of the files is possible to see which one is being generated.

```
C:\Users\João Rodrigo\Desktop\ESRGProject\Sample>java -jar generator.jar sample.MyDsl
Starting Hybrid generation...
    -Generating BRISK Detector...
        -Generating Detection...
        -Generating Scoring...
        -Generating Contiguity Test...
        -Generating Non Maximum Suppression...
    -Generating BRIEF Descriptor ...
        -Generating Patch...
        -Generating Binary Tests...
    -Generating Matching...

Code generation finished.
```

Figure 5.1.11: Command line result

The result of this project can be observed in the two figures above. In figure 5.1.12 are shown the folders for all the algorithms. Inside of them, are folders for all the operation modes. The image 5.1.13, is an example of all the files inside the folder BRISK.

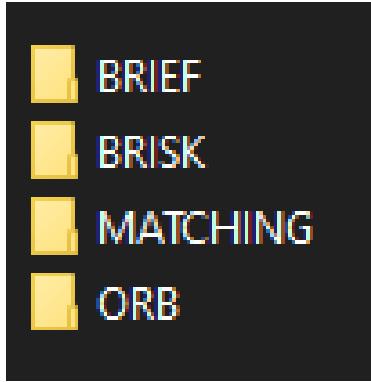


Figure 5.1.12: Folders for all the algorithms

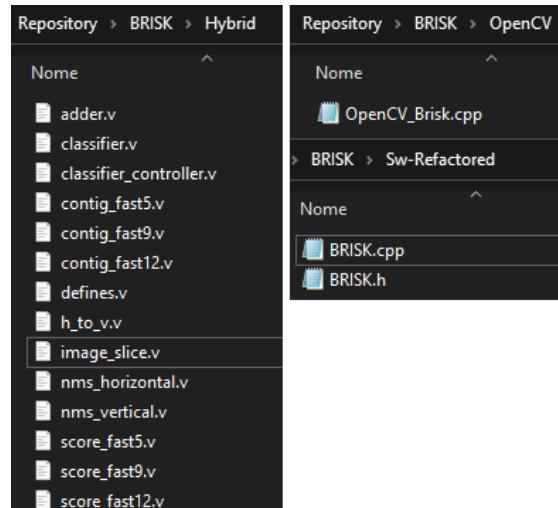


Figure 5.1.13: Project Repository for BRISK

The figure 5.1.14 contains generated files relative to a complete code generation. The user script defined that the operation mode was Hybrid so that the files of the Hybrid implementation were chosen according to the value of fast (in this case is 9) and the values of patch size (11) and string size (256).

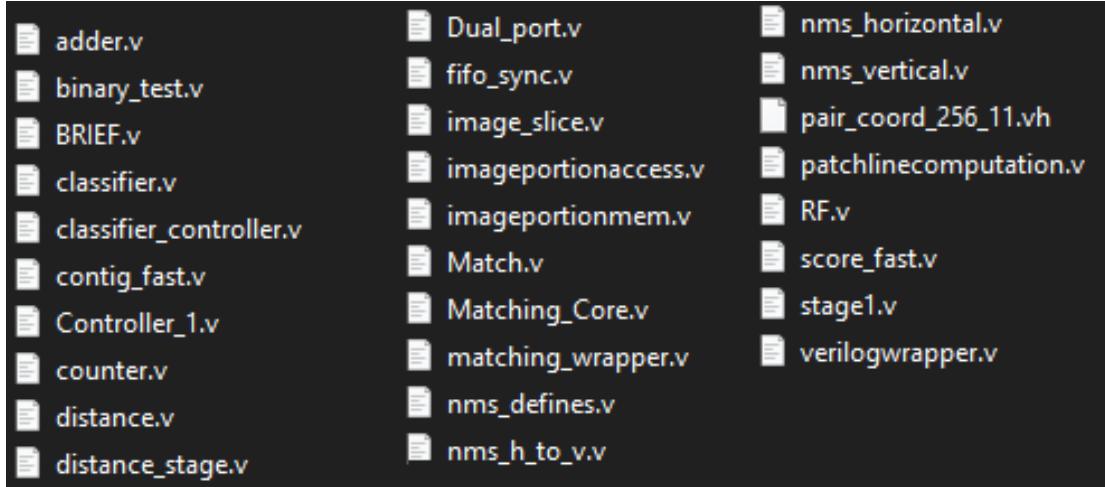


Figure 5.1.14: Final Result

5.2 Execution

This section will portray the tests and results in terms of system execution.

5.2.1 ORB Hardware

This section will refer to the tests and result assessment performed in terms of the ORB modules.

Patcher module

In order to better visualise the output of the module that generates the patch values, one created a testbench that takes the hexadecimal values of an image and uses these values to feed the patcher module. In order to validate the correct output of this module, the testbench used was developed so that it saves every output variable and important internal variable into a file for later evaluation.

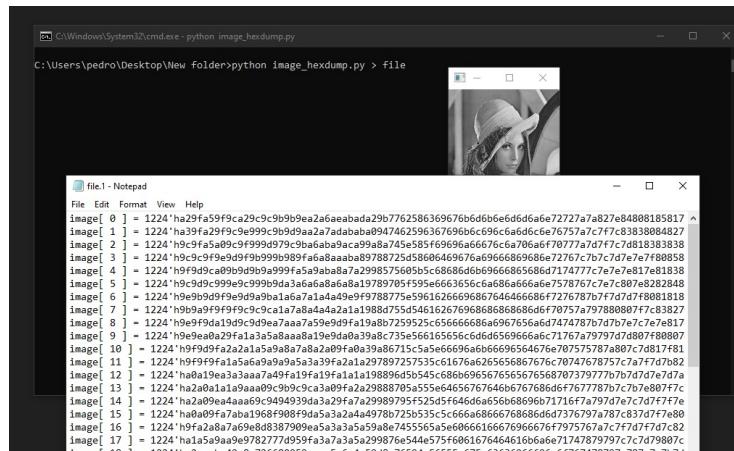


Figure 5.2.1: Conversion of the image to hexadecimal values

Patch generation. As can be seen in figure 3.3.19, only when the 7×7 file register matrix is filled does the valid flag rise. Additionally, and as was stated in previous chapters, the image appears inverted in the file registers. For different orders of FAST-n, the mechanism still behaves adequately.

```

9d 9e 9f 9d 9b 9e 2f
9c 9e 99 9c 9d 9c 34
9d 9b a0 9c 9d 9f 2d
9f 9d 9e 9f 9c 9c 2f
9f 9c a0 a5 9f 9c 83
9e 9c 9f a2 9f a3 9a
a2 9c 9f a5 9f a2 00

Center and Circle [ 9f, 9d, 9b, 9c, 2d, 2f, 83, a3, 9f, a5, 9f, 9c, 9f, 9f, 9d, 9f ]
Output Valid: 0
Current Keypoint Coordinates: row      3, col      2
Patch M10 value :      26634
Patch M01 value :      28763

```

```

9a 9d 9e 9f 9d 9b 9e
99 9c 9e 99 9c 9d 9c
9b 9d 9b a0 9c 9d 9f
9b 9f 9d 9e 9f 9c 9c
99 9f 9c a0 a5 9f 9c
99 9e 9c 9f a2 9f a3
9c a2 9c 9f a5 9f a2

Center and Circle [ 9e, 9f, 9d, 9d, 9f, 9c, 9c, 9f, a5, 9f, 9c, 9e, 99, 9b, 9b, 9e ]
Output Valid: 1
Current Keypoint Coordinates: row      3, col      3
Patch M10 value :      30997
Patch M01 value :      31006

```

Figure 5.2.2: Patch Valid for the first time for a Fast of order 12

```

a0 9c 9d 9f 2d 2c 36
9e 9f 9c 9c 2f 3b 6d
a0 a5 9f 9c 83 9f 7f
9f a2 9f a3 9e a7 7d
9f a5 9f a2 00 00 00
00 00 00 xx 00 00 00
00 00 00 xx 00 00 00

Center and Circle [ 9f, 9d, 9f, 2f, 83, 9e, a2, 9f, a5, 9f, a0, 9e, 9c, 00, 00, 00 ]
Output Valid: 0
Current Keypoint Coordinates: row      2, col      1
Patch M10 value :      9869
Patch M01 value :      10771

```

```

9b a0 9c 9d 9f 2d 2c
9d 9e 9f 9c 9c 2f 3b
9c a0 a5 9f 9c 83 9f
9c 9f a2 9f a3 9e a7
9c 9f a5 9f a2 00 00
00 00 00 00 xx 00 00
00 00 00 00 xx 00 00

Center and Circle [ a5, 9c, 9d, 9c, 9c, a3, 9f, a5, 9f, 9c, 9c, 9d, a0, 00, 00, 00 ]
Output Valid: 1
Current Keypoint Coordinates: row      2, col      2
Patch M10 value :      11933
Patch M01 value :      11948

```

Figure 5.2.3: Patch Valid for the first time for a Fast of order 9

```

9f 9c 83 9f 7f 78 76
9f a3 9e a7 7d 76 75
9f a2 00 00 00 00 00
00 xx 00 00 00 00 00

Center and Circle [ a3, 9c, 83, 9e, 00, a2, 9f, 9f, 9f, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 0
Current Keypoint Coordinates: row      1, col      0
Patch M10 value :      2306
Patch M01 value :      2369

```

```

a5 9f 9c 83 9f 7f 78
a2 9f a3 9e a7 7d 76
a5 9f a2 00 00 00 00
00 00 xx 00 00 00 00

Center and Circle [ 9f, 9f, 9c, a3, a2, 9f, a5, a2, a5, 00, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 1
Current Keypoint Coordinates: row      1, col      1
Patch M10 value :      2889
Patch M01 value :      2906

```

Figure 5.2.4: Patch Valid for the first time for a Fast of order 5

```

34 2f 2e 33 2c 32 5c
2f 30 2d 2f 31 5e 85
34 2f 38 34 63 86 83
2d 2c 36 6b 84 82 7c
2f 3b 6d 80 7f 7e 7f
83 9f 7f 78 76 7d 7f
9e a7 7d 76 75 80 7f

Center and Circle [ 6b, 33, 2c, 5e, 83, 7c, 7f, 7d, 75, 76, 7d, 9f, 2f, 2d, 34, 2e ]
Output Valid: 1
Current Keypoint Coordinates: row      3, col      149
Patch M10 value :      20031
Patch M01 value :      21202

```

```

9b 34 2f 2e 33 2c 32
9e 2f 30 2d 2f 31 5e
9c 34 2f 38 34 63 86
9f 2d 2c 36 6b 84 82
9c 2f 3b 6d 80 7f 7e
9c 83 9f 7f 78 76 7d
a3 9e a7 7d 76 75 80

Center and Circle [ 36, 2e, 33, 31, 86, 82, 7e, 76, 75, 7d, a7, 83, 9c, 9f, 9c, 2f ]
Output Valid: 0
Current Keypoint Coordinates: row      4, col 4294967293
Patch M10 value :      18868
Patch M01 value :      22101

```

Figure 5.2.5: Patch Valid to invalid transition for the first time for a Fast of order 12

```

34 2f 38 34 63 86 83
2d 2c 36 6b 84 82 7c
2f 3b 6d 80 7f 7e 7f
83 9f 7f 78 76 7d 7f
9e a7 7d 76 75 80 7f
00 00 00 00 00 00 00
00 00 00 00 00 00 00

Center and Circle [ 6d, 38, 34, 84, 7f, 76, 76, 7d, a7, 83, 2f, 2d, 2f, 00, 00, 00 ]
Output Valid: 1
Current Keypoint Coordinates: row      2, col      150
Patch M10 value :      7863
Patch M01 value :      8525

9c 34 2f 38 34 63 86
9f 2d 2c 36 6b 84 82
9c 2f 3b 6d 80 7f 7e
9c 83 9f 7f 78 76 7d
a3 9e a7 7d 76 75 80
a2 00 00 00 00 00 00
xx 00 00 00 00 00 00

Center and Circle [ 3b, 2f, 38, 6b, 80, 78, 7d, a7, 9e, 9c, 9f, 34, 00, 00, 00, 00 ]
Output Valid: 0
Current Keypoint Coordinates: row      3, col 4294967294
Patch M10 value :      7593
Patch M01 value :      9105

```

Figure 5.2.6: Patch Valid to invalid transition for the first time for a Fast of order 9

Valid flag toggling. In the next 3 Figures 5.2.8 to 5.2.10 it's seen that given the diminishing patch dimension according to the diminishing order of the FAST, the output is valid for more points, increasing the coordinates of the last valid keypoint for each line.

```

2f 3b 6d 80 7f 7e 7f
83 9f 7f 78 76 7d 7f
9e a7 7d 76 75 80 7f
00 00 00 00 00 00 00
00 00 00 00 00 00 00
00 00 00 00 00 00 00
00 00 00 00 00 00 00

Center and Circle [ 9f, 3b, 6d, 7f, 7d, a7, 9e, 83, 2f, 00, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 1
Current Keypoint Coordinates: row      1, col      151
Patch M10 value :      2189
Patch M01 value :      2399

9c 2f 3b 6d 80 7f 7e
9c 83 9f 7f 78 76 7d
a3 9e a7 7d 76 75 80
a2 00 00 00 00 00 00
xx 00 00 00 00 00 00
xx 00 00 00 00 00 00
xx 00 00 00 00 00 00

Center and Circle [ 83, 2f, 3b, 9f, a7, 9e, a3, 9c, 9f, 00, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 0
Current Keypoint Coordinates: row      2, col 4294967295
Patch M10 value :      2302
Patch M01 value :      2618

```

Figure 5.2.7: Patch Valid to invalid transition for the first time for a Fast of order 5

In the next 3 Figures 5.2.8 to 5.2.10 it's validated the values to which the keypoint needs to be in order for the output of the Patcher module to be considered valid.

```

9c 9f 9f 9f 9a 9b 34
9d 9e 9f 9d 9b 9e 2f
9c 9e 99 9c 9d 9c 34
9d 9b a0 9c 9d 9f 2d
9f 9d 9e 9f 9c 9c 2f
9f 9c a0 a5 9f 9c 83
9e 9c 9f a2 9f a3 9e

Center and Circle [ 9c, 9f, 9a, 9e, 34, 2d, 2f, 9c, 9f, a2, 9f, 9c, 9f, 9d, 9c, 9f ]
Output Valid: 0
Current Keypoint Coordinates: row      4, col      2
Patch M10 value :      26907
Patch M01 value :      29177

```

```

9c 9c 9f 9f 9f 9a 9b
9a 9d 9e 9f 9d 9b 9e
99 9c 9e 99 9c 9d 9c
9b 9d 9b a0 9c 9d 9f
9b 9f 9d 9e 9f 9c 9c
99 9f 9c a0 a5 9f 9c
99 9e 9c 9f a2 9f a3

Center and Circle [ a0, 9f, 9f, 9b, 9c, 9f, 9c, 9f, a2, 9f, 9c, 9f, 9b, 9b, 99, 9f ]
Output Valid: 1
Current Keypoint Coordinates: row      4, col      3
Patch M10 value :      30885
Patch M01 value :      30875

```

Figure 5.2.8: Patch invalid to valid second transition for the first time for a Fast of order 12

```

99 9c 9d 9c 34 2f 38
a0 9c 9d 9f 2d 2c 36
9e 9f 9c 9c 2f 3b 6d
a0 a5 9f 9c 83 9f 7f
9f a2 9f a3 9e a7 7d
9f a5 9f a2 00 00 00
00 00 00 xx 00 00 00

Center and Circle [ 9c, 9d, 9c, 2d, 2f, 83, a3, 9f, a2, a0, 9e, a0, 9c, 00, 00, 00 ]
Output Valid: 0
Current Keypoint Coordinates: row      3, col      1
Patch M10 value :      10075
Patch M01 value :      11145

```

```

9e 99 9c 9d 9c 34 2f
9b a0 9c 9d 9f 2d 2c
9d 9e 9f 9c 9c 2f 3b
9c a0 a5 9f 9c 83 9f
9c 9f a2 9f a3 9e a7
9c 9f a5 9f a2 00 00
00 00 00 00 xx 00 00

Center and Circle [ 9f, 9c, 9d, 9f, 9c, 9f, a2, 9f, 9c, 9d, 9b, 99, 00, 00, 00 ]
Output Valid: 1
Current Keypoint Coordinates: row      3, col      2
Patch M10 value :      11858
Patch M01 value :      11891

```

Figure 5.2.9: Patch invalid to valid second transition for the first time for a Fast of order 9

```

9c 9c 2f 3b 6d 80 7f
9f 9c 83 9f 7f 78 76
9f a3 9e a7 7d 76 75
9f a2 00 00 00 00 00
00 xx 00 00 00 00 00
00 xx 00 00 00 00 00
00 xx 00 00 00 00 00

Center and Circle [ 9c, 9c, 2f, 83, 9e, a3, 9f, 9f, 9c, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 0
Current KeyPoint Coordinates: row      2, col      0
Patch M10 value : 2432
Patch M01 value : 2691

9f 9c 9c 2f 3b 6d 80
a5 9f 9c 83 9f 7f 78
a2 9f a3 9e a7 7d 76
a5 9f a2 00 00 00 00
00 00 xx 00 00 00 00
00 00 xx 00 00 00 00
00 00 xx 00 00 00 00

Center and Circle [ 9f, 9c, 9c, a3, 9f, a2, a5, 9f, 00, 00, 00, 00, 00, 00, 00 ]
Output Valid: 1
Current KeyPoint Coordinates: row      2, col      1
Patch M10 value : 2859
Patch M01 value : 2883

```

Figure 5.2.10: Patch invalid to valid second transition for the first time for a Fast of order 5

Corner Score computation

Unit Integration. A testbench was designed to deliver full unity test coverage. By applying carefully chosen parameters to the input's one can register and compare its outputs to what one would expected, if a match is done, a higher confidence level of our code would be achieved.

The figure 5.2.11 present three cases were highlighted, in different colours, for different test groups.

- In **red**, it will be used to test the Scorer and the Contiguity Test, along with the Contiguity Counter.
- In **green**, it will be used to cover special cases with the Contiguity Counter.
- In **blue**, it is used to test the module's pipeline.

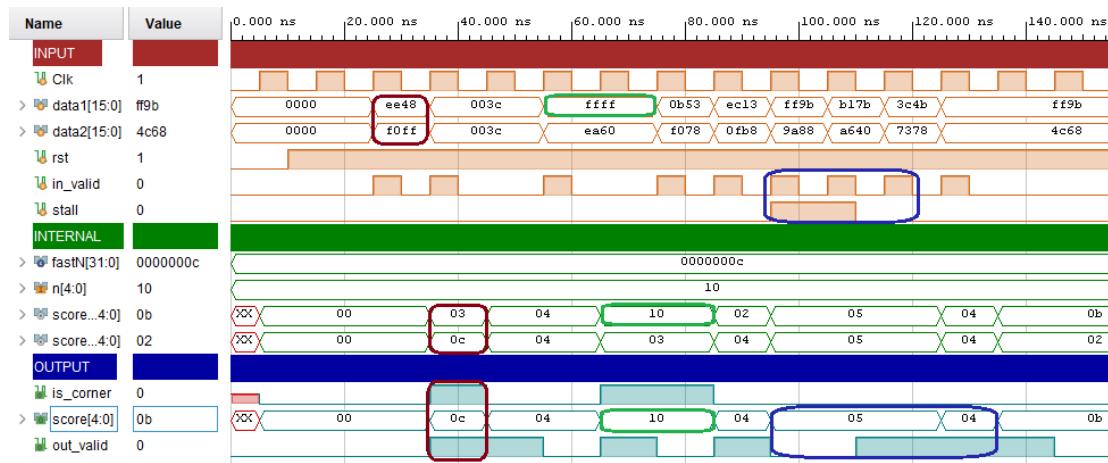


Figure 5.2.11: Corner Score Computation TestBench.

Marked in **red** are the both inputs received, and the Contiguity Counter performing properly its operation, as one can see by both of the scores. Next, the score selected was the higher one, as expected, and it passed the contiguity test (since the system is working with a boundary of 12-bit needed by the used of a 16-bit circle, represented by *fastN* and *n*). As well as *out_valid* was correctly asserted.

In **green** is used to check if it count correctly the array as if it was a circle, and if needed limited the score value.

Highlighted in **blue**, one can see that after a latency of 2 cycles, for each input one would get a new output, thus providing that the pipeline is working. Plus when stalled, the pipeline was stalled, and all new inputs were ignored, as wells as *out_valid* going to 0. And when *stall* goes back to 0, the module resumes were it left.

Resources. As said previously most of our resources are due to the use of the for loop, since it duplicates the hardware within the code block in order to perform the desired operation within a clock cycle.

The IO ports should be ignored since this is will be discarded when instantiated by another module.

Resource	Utilization	Available	Utilization %
LUT	513	17600	2.91
FF	44	35200	0.13
IO	50	100	50.00
BUFG	1	32	3.13

Figure 5.2.12: Resource Usage Table.

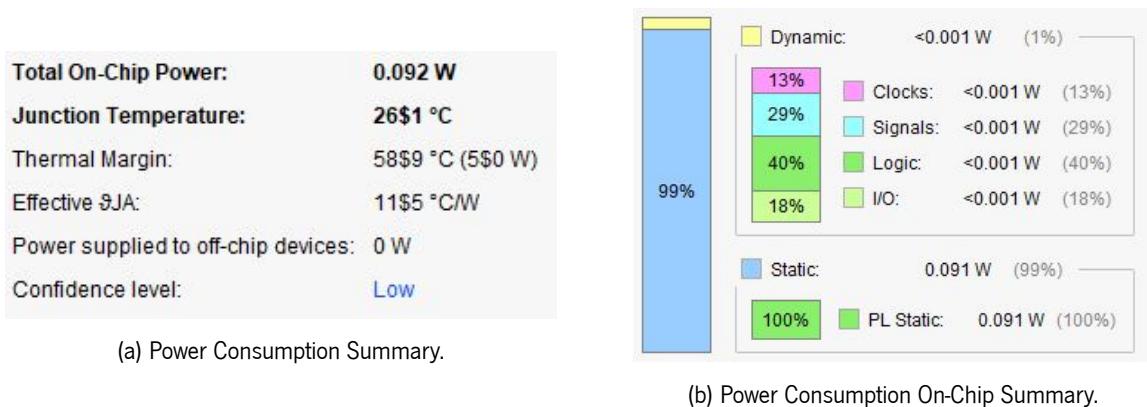
Power Consumption. Power Consumption is closely related to the amount of hardware used and the constraint's one may add to the circuit. With no constraints applied, one can observe that the total power exceeds the maximum amount. This is due to the for loop since in order to do that many calculations in a set time frame a lot of resources are used to be able to achieve so.

Total On-Chip Power:	7.801 W (Junction temp exceeded!)
Junction Temperature:	115\$0 °C
Thermal Margin:	-30\$0 °C (-2\$2 W)
Effective θ _{JA} :	11\$5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low
Implemented Power Report	

Figure 5.2.13: Power Consumption Summary.

But by tweaking the constraints one can diminish the power consumption, for example by changing one timing constraint for a larger one or choosing a lower clock frequency may vary significantly power consumption.

Below is presented the Power Consumption Summary for a clock frequency of 5MHz. As shown, the power decreases dramatically with that known, it is more than reasonable for what the board can handle. Even though the clock frequency is low the system can still out-perform its software counter-part by a long shot, as will be shown further down below.



(a) Power Consumption Summary.

(b) Power Consumption On-Chip Summary.

Corner Score Computation of Hardware VS Software Algorithms. In order to get more data to correctly evaluate our system performances, its software equivalent was developed and tested to get a better understanding of how much faster hardware acceleration is when compared to software.

Testing was done on a PC with the following specifications:

- Processor: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- RAM memory: 12.0GB
- 64 bits operative system, x64 based processor
- Product ID: 00325-81732-26297-AAOEM

And the final results are shown below. As one can observe with a CPU usage of 80% it took 17ms to compute the score along with the contiguity test.

Test Case	Frequency	Time
Software Refactored Execution Time	2.6GHz	17ms
Hardware Accelerator Execution Time	5MHz	0.4us

Table 5.1: HW vs SW test.

When compared to the group's implementations which takes 2 clock cycles to finish, and by assuming - for example - the clock frequency referenced in figure 5.2.14a - it would take 0.4us to finish. Which is around 42 thousand times faster than its counterpart, thus highlighting the hardware acceleration biggest advantage.

NMS

To perform the validation results, the inputs are fixed in 10 for the keypoint's x coordinate, 15 for the keypoint's y coordinate and the moments of the patch used are the ones obtained in the Theoretical Basis example (30997 for m01 and 30734 for m10). The expected output is the oriented x coordinate 18 and the oriented y coordinate 3.

NMS: Hybrid

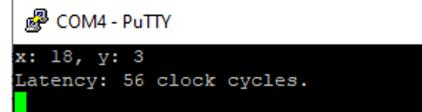


Figure 5.2.15: Hybrid approach validation.

NMS: Hardware with Floating-Point Numbers

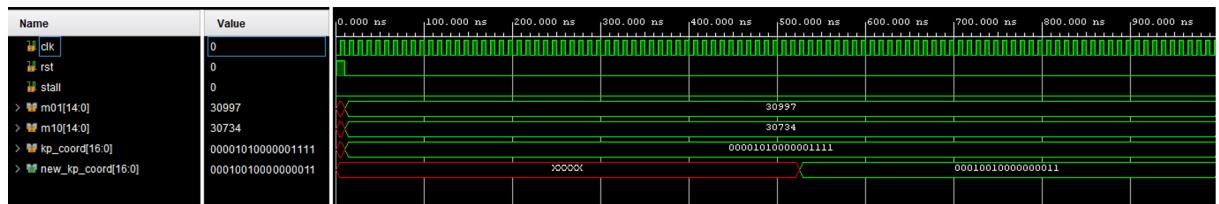


Figure 5.2.16: Hardware with floating-point numbers approach validation

NMS: Hybrid Hardware with Fixed-Point Numbers (N=100)

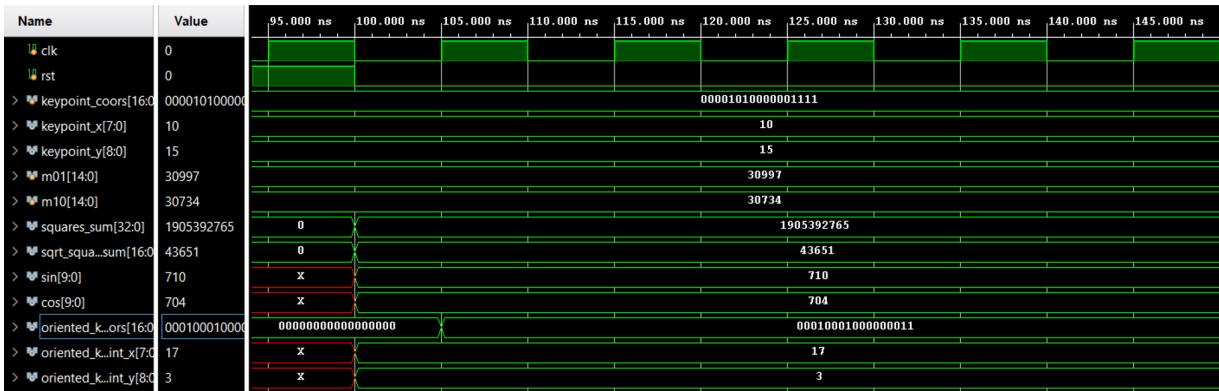


Figure 5.2.17: Results compilation

NMS: Results. In order to compare the three implementations, a table was created compiling all the metrics for each approach. The latency was measured in simulation for both hardware implementations as the Vivado software lacks reports for it, however, for the hybrid approach the global timer counter register was used. The Programmable Logic resources utilisation was obtained via implementation utilisation report provided by Vivado.

ORB Implementation. The implementation follows the natural flow of the ORB algorithm. The detection stage is followed by the scorer, next comes the NMS and finally the rotation invariant stage (figure 5.2.18).

Approach		Latency		PL Resources			
	Max.(Clock cycles)	Imp. (%)	LUT	LUTRAM	FF	DSP	
Hybrid	56 + (AXI Latency)	Base	-	-	-	-	-
Hardware	Floating-Point	49	2739	130	5656	10	
	Fixed-Point	1	1696	-	34	2	

Table 5.2: Hardware with fixed-point numbers approach validation

The rotation invariant stage was set as the last stage due to the non-deterministic nature of the NMS stage -due to it's workload depending on much clustered the keypoints are -, and so it's harder to stall the data needed in the rotation invariant stage from the detector. So the data is forward within the NMS stage, and only when ready it outputs the previously stored information.

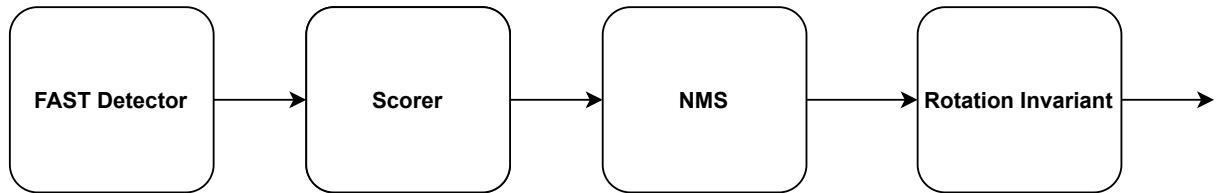


Figure 5.2.18: ORB Block Diagram

ORB Integration. With the unity tests being done one could provide a higher confidence level to each of our modules, and so proceed to their integration. At this stage in the implementation interfaces are being tested either being logically or temporally.

Due to the Detector and the NMS stages having longer and more complex execution only interface variables are presented to keep the simulation clear and simple.

When a keypoint candidate is detected by the Detection stage and the Bright/Dark classifier finishes the *is_dark* and *is_bright* are sent to the scorer. As well as the coordinates of the candidate and it's *m10* and *m01* values are forward to the pipeline registers until they reach the NMS stage at the same time as the scorer finishes. Therefore all data can be timely correct so that the NMS can use the information it needs along with the one which will be forwarded to the rotation invariant stage.

Three specific type of variables are being tracked to prove the correct function of the module.

- In **green** is highlighted the data that flow's through the scorer
- In **red** the information to be forwarded to the Rotation Invariant
- In **blue** the coordinates, information relevant to either the NMS and the Rotation Invariant stages

By analysing the picture above one could prove that:

- the scorer can correctly receive an input from the detector and provide one for the NMS
- the pipeline registers can correctly forward the data needed for the NMS stage

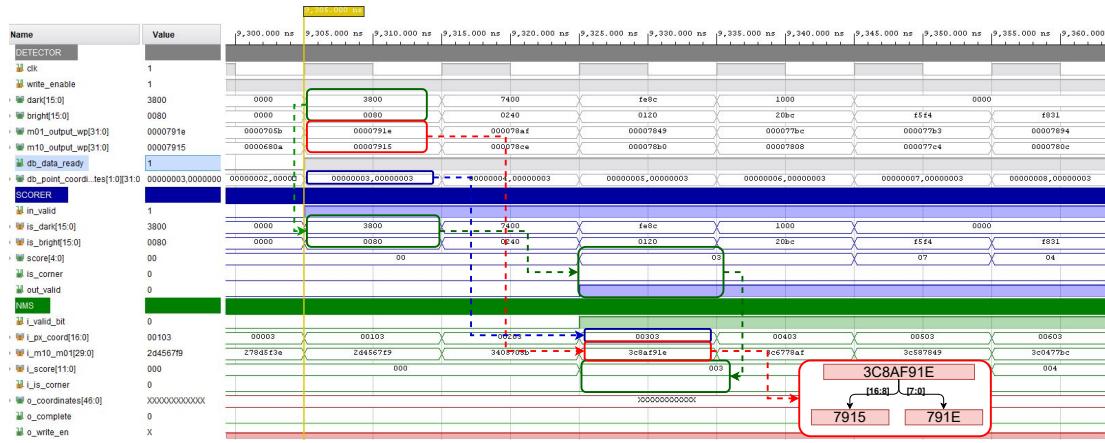


Figure 5.2.19: ORB Simulation - Detector, Scorer, NMS

- the pipeline registers can correctly forward the data to the NMS stage to be forward within it self, needed by the Rotation Invariant stage

Next, when the NMS already has achieved what can be considered as a keypoint it should proceed to output the coordinates of the said keypoint along with its $m10$ and $m01$. Next the Rotation Invariant stage will proceed to compute the rotated coordinates.

- In **blue** it's marked the data flow of the Rotation Invariant stage, either it being it's inputs or outputs

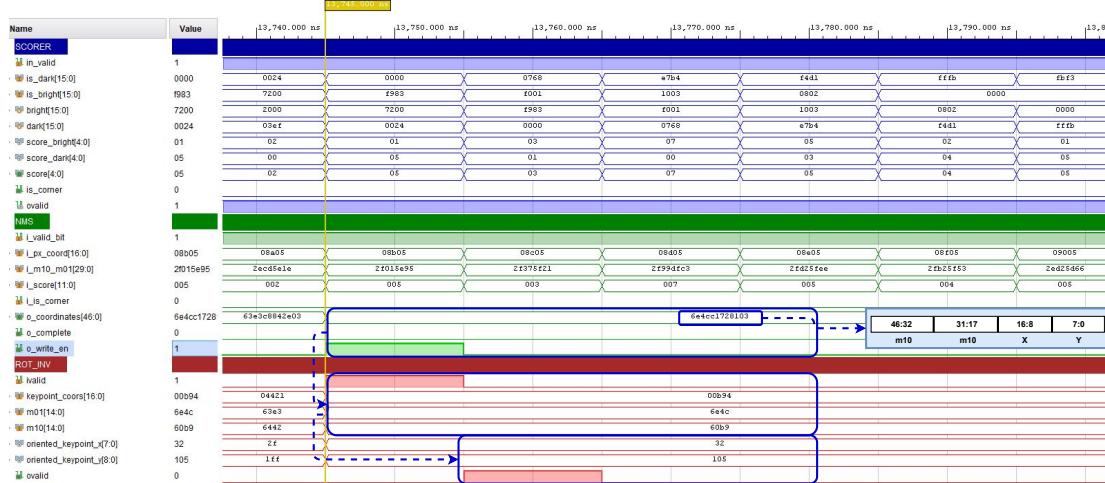


Figure 5.2.20: ORB - NMS and Rotation Invariant

As one could observe, when ready, the NMS stage delivers the correct keypoint coordinates along with the other information needed. Next to Rotation Invariant stage proceeds to operate and deliver the final rotated coordinates to be stored in a FIFO and later be used by the descriptor.

5.2.2 ORB Software

To see the results of the software implementation we created this small code where you can see the functions used to obtain the following image. This result is relative to FAST12 for a threshold of 20.

Listing 5.1: create() Method

```

1 int main() {
2     cv::Mat image, image1, grayImage, imageBlurred;
3     image = cv::imread("lion.jpeg", cv::IMREAD_COLOR);
4     cv::cvtColor(image, grayImage, cv::COLOR_RGB2GRAY);
5     GaussianBlur(grayImage, imageBlurred, cv::Size(3, 3), 0, 0);
6     if(! image.data ) {
7         std::cout << "Image not found or unable to open" << std::endl ;
8         return -1;
9     }
10
11     image1 = image.clone();
12     ORB detector(FAST12);
13     list<point_t> lista = detector.detection(imageBlurred, 20);
14
15     cv::Scalar green( 0, 255, 0);
16     for (auto it = lista.begin(); it != lista.end(); ++it){
17         cv::circle( image1, {it->x, it->y}, 1, green, 2);
18     }
19     cv::imshow( "fast12", image1 );
20     cv::waitKey(0);
21     return 0;
22 }
```

Using the same computer as described above, the time results obtained and the CPU(%) used were as follows:

Test Case	%CPU	Time
Software Refactored Execution Time	110%	120ms
OpenCV Execution Time	49%	4060ms

Table 5.3: OpenCV vs Software Refactored.

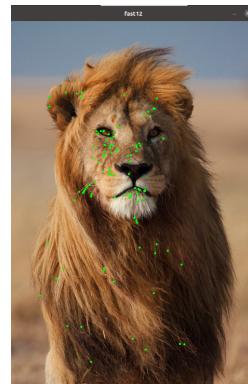


Figure 5.2.21: Final Result

5.2.3 BRISK

This section will refer to the tests and result assessment performed in terms of the BRISK modules.

Image Slicing and BRAM

To test the image slice module using Vidado's simulation tools, one generated a random 32x8 image in photoshop and used a python script (listing B.1) to convert it into an array that could be introduced in the testbench to perform several memory write operations to store the image slices in the BRAM. The generated array to store in the BRAM is represented in listing A.1. With the overall simulation, it's clear that one is performing successive memory writes from line zero to seven as depicted in fig. 5.2.22 (chosen for Writing). However, since the code for writing in the BRAM in blocks of 4 bytes was too long, one opted to use blocks of 16 bytes as proof of concept.

BRAM Write. Regarding this, one can see an example in fig. 5.2.22 of data being written to the first line of the BRAM, using the A port (dina) and in blocks of 16 bytes (highlighted in **yellow**), accordingly to what as specified in the testbench of listing A.1. Observe that the line chosen was line zero (first one) specified in **grey**, and considering the writing is done on blocks of 16 bytes, as forementioned, firstly one writes the 16 bytes in **black** and then the ones in **red**, as the image is 32 bytes in width (and 8 bytes in depth).

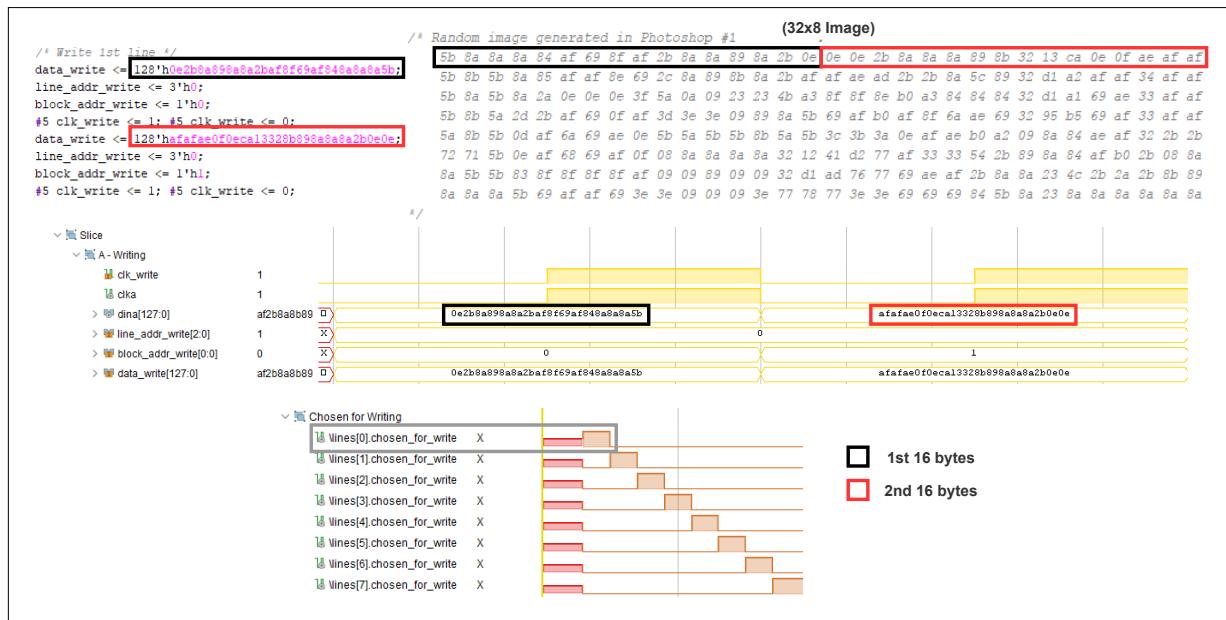


Figure 5.2.22: Image Slice Module Simulation: Details

Dynamic Mapping. In fig. 5.2.23, one can see the dynamic addresses changing as the simulation proceeds.

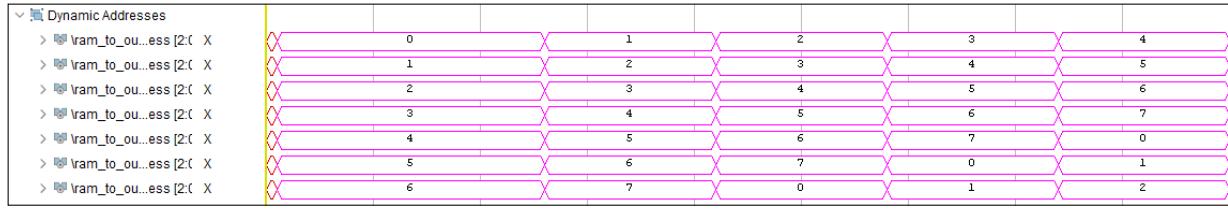


Figure 5.2.23: Image Slice Module Simulation: Dynamic Mapping

BRAM Read and Output Forwarding. After the eight line writes in the BRAM are done (highlighted in **orange** in fig. 5.2.22) and all of the 32x8 image is stored, the detection stage begins reading from memory to initiate the classifier, detection controller and register file data flux, until it reaches the scoring stage. This can be observed in fig. 5.2.24. Additionally, with proper analysis of the simulation results, one can see that at the time marked by the **yellow** cursor, the output of the BRAM is the one highlighted in **black**. This shows that the memory output at the B port (`doutb`) is represented through blocks of four bytes and also that the reading is done from right to left, as expected and specified in the design stage, fig. 3.3.3. The **red** highlighting intends to represent the lines read from the BRAM by the detection stage. Moreover, note that the **grey** rectangles depict four bytes of data and, as seen in fig. 4.2.1, each line of the BRAM is enabled but only the select ones are forwarded to `data_read`. In this case, the detector chose to forward the lines from 1 to 7.

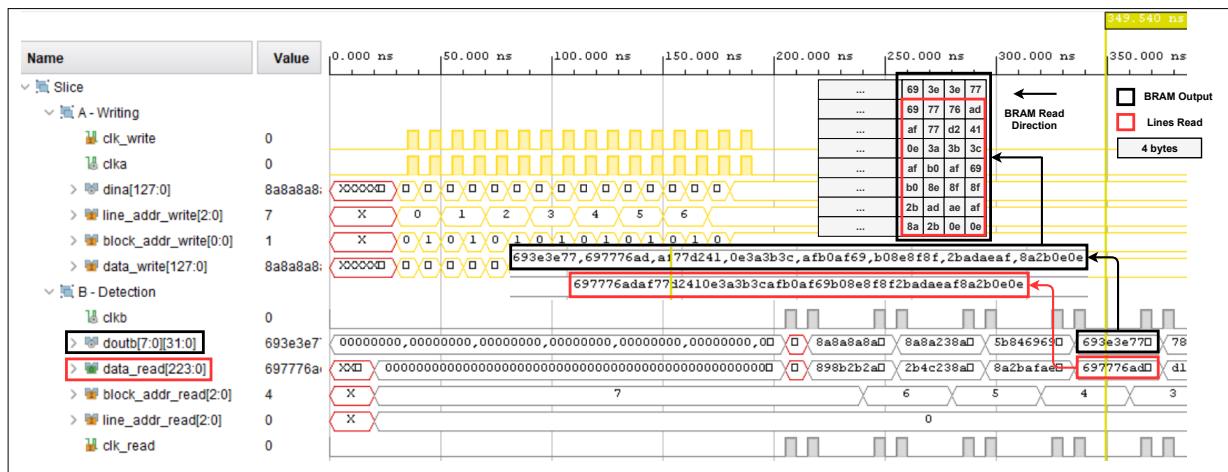


Figure 5.2.24: Simulation: Image slices read from BRAM

Register File, Classifier and Detection Controller

Register File. Following the same data portrayed in fig. 5.2.24, highlighted in **green**, one can see in fig. 5.2.25 that it enters the register file in a parallel manner through the Parallel In Serial Out (PISO) cells (in **blue**), when the `i_en_load` signal is asserted (high). Soon after, the data is shifted serially onto the Serial In Serial Out (SISO) cells as seen in **black**. When one reaches the SISOs, it is possible to see multiple and successive Bresenham circles. Each shift of the SISOs is represented in fig. 5.2.25 in a different colour.

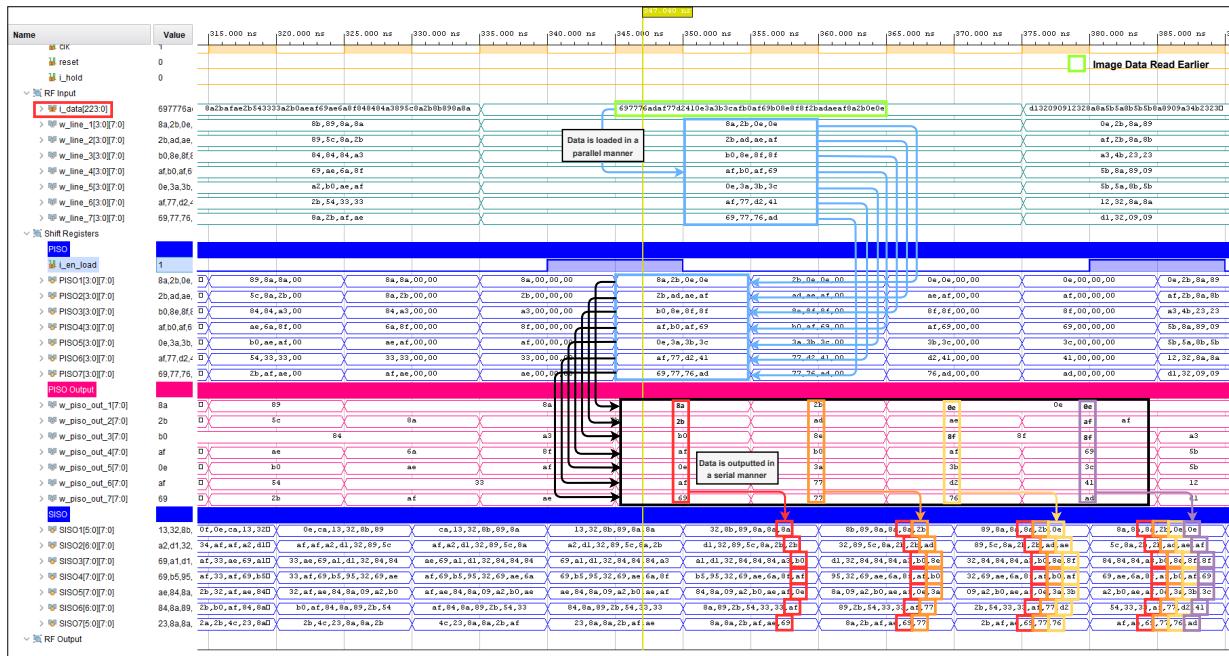


Figure 5.2.25: Simulation: Register File

Bresenham Circle Definition. Considering, for example, the first valid Bresenham circle (yellow cursor) given by the `o_valid_cnt_nms` signal marked in red, one can see that the data in the SISOs behaves as expected, describing a Bresenham circle that matches the register file output values (highlighted in black).

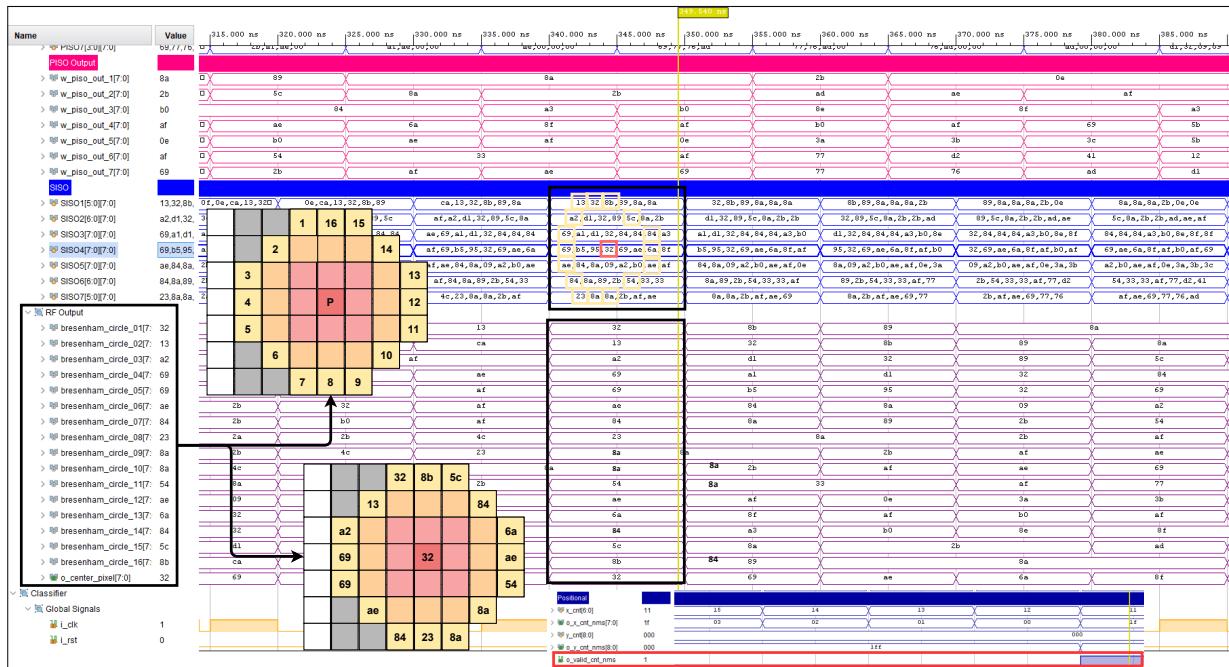


Figure 5.2.26: Simulation: Bresenham Circle FAST-12

Classifier. As it's possible to see in fig. 5.2.27, the Bresenham circle defined in fig. 5.2.26 in passed onto the classifier as input, as highlighted in **black**. The latter produced all the expected results, performing the operations necessary to populate the *is_bright* and *is_dark* arrays, and also the arrays referent to the score associated to those pixels. One can observe this and the transition of the pixel score values to the scoring stage of BRISK, both marked in **red** in fig. 5.2.27. Note that, only the pixels with an outcome greater than zero are put to high in the *is_bright* and *is_dark* arrays and are associated with their respective score. The threshold value used in fig. 5.2.27 is 10 (in hexadecimal).

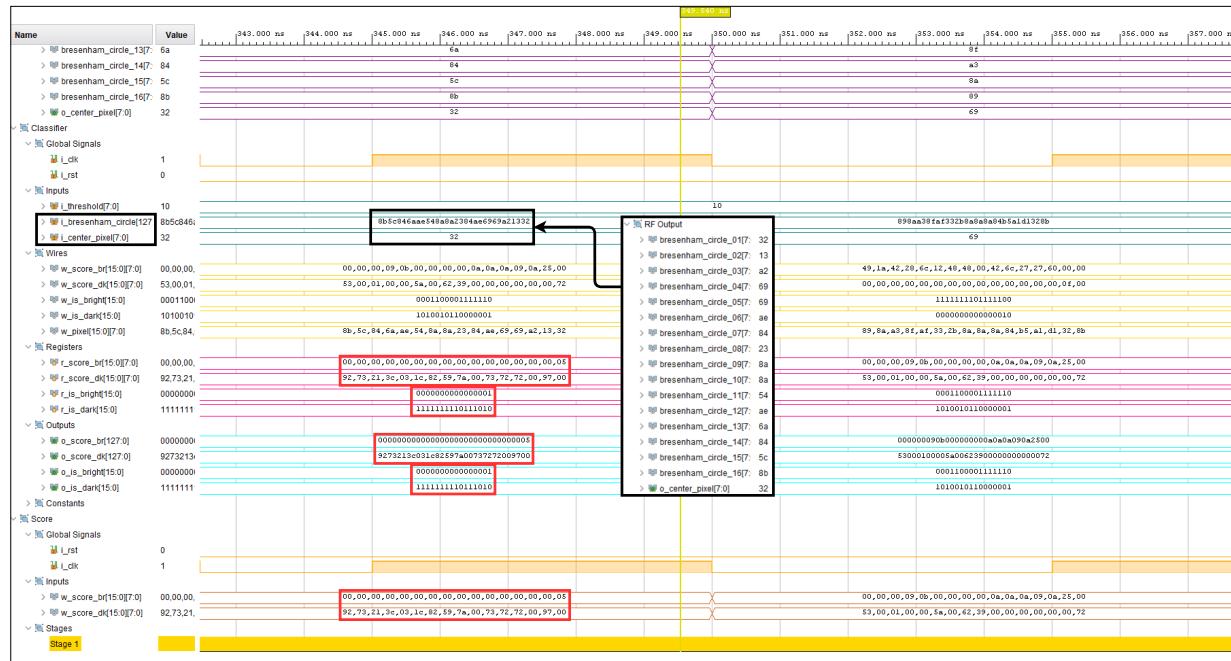


Figure 5.2.27: Simulation: Classifier

Detection Controller. In it the controller can be observed coordinating memory reads with the value in the 2-bit edge counter and the value of the coordinate addresses with the *x_cnt* and *y_cnt* counters. One can also observe how it coordinates shift and write operations with the Register File module - a close analysis of fig. 5.2.25 with attention to the time stamps will confirm that the timing is perfectly met. Another interesting observation may be confirming that the calculations for the coordinates for the Non-Maximum Suppression stage are done correctly, for a delay of 14 cycles, and the validity of the coordinates is well assessed.

As the goal with the controller is not to do the operations on the information but rather to coordinate them, the effect of its execution has been best analysed in the previous chapters.

Contiguity Test. In order to test the contiguity test modules for the different types of FAST, fictitious arrays of *is_bright* and *is_dark* were created as inputs where different contiguity states were included, alternating between a state where it is considered keypoint and a state where it is not considered keypoint.

In the figure 5.2.28, it is possible to see the correct behaviour of the three modules where when a valid state is entered *is_corner*, signalled in the red rectangle, takes logical value '1' on the next positive edge of the clock as it

is necessary to delay the test result to be in sync with the scoring stage as both are done in parallel and the results should be calculated at the same time.

This is done using two registers passing the value from one to the other as one can see in the simulation. When an invalid state is entered *is_corner* takes the logic value '0' signalling that that pixel is not considered a keypoint.

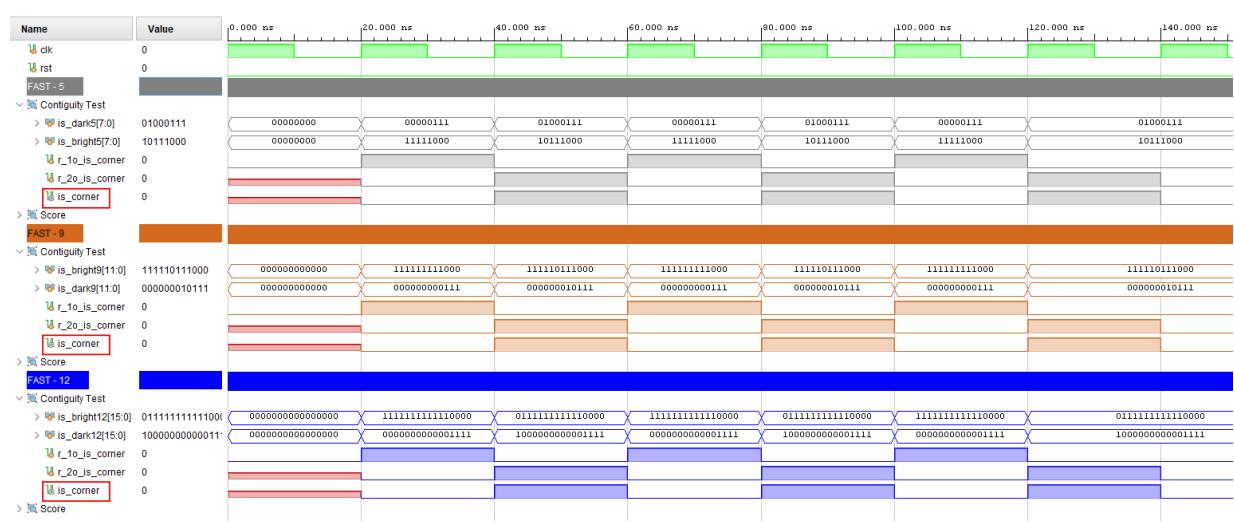


Figure 5.2.28: Contiguity Test Simulation for FAST-5,FAST-9 and FAST-12

Score Computation. In the simulation it is possible to verify that everything occurred as foreseen in the design phase. In case N is 5, 9 or 12, the score is always computed properly and correctly in time, in the sense of selecting the highest value of the sums among the input arrays. The simulation is divided into inputs, stages, and outputs, with the four stages representing the four chains of successive adders. In the image 5.2.33 is represented the behavioural simulation of FAST-12. The dark and bright pixels, *score_dk* and *score_br*, received as input, are divided in order to perform the sums successively. The size of the output array *score*, indirectly imposed by N, is also verified, being the result of the sums made by the various adders.

Horizontal NMS. The next figure, apart from proving the previous point, also shows the necessary data to guarantee the filter correction. There are represented the three assigned registers from the input pixels, and the output according to the comparison among them. The simulation radix is hexadecimal, which means that the pixel score corresponds to the three least significant numbers. Joining that with the position bits, we can observe that in the first two times the central corner wasn't suppressed. Then, it came a neighbour with a lower score (3'h200 < 3'h400), which is why the output wasn't updated with its value.

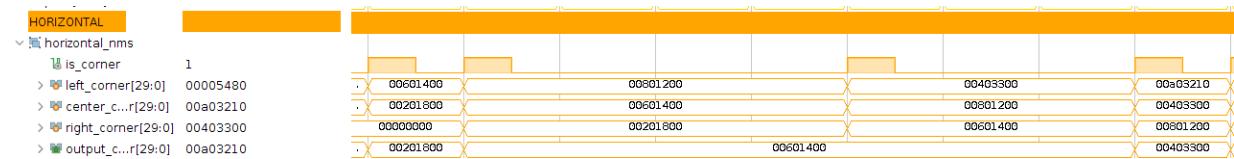


Figure 5.2.29: Horizontal filter suppression.

Vertical NMS. In the vertical simulation, it can be enforced the same analysis logic. The output was in concordance with the central corner (occurring algorithm forwarding) until the test bench arrived at the third column and has to deal with two neighbors. In that case, once the score (3'h210) was lower than the above corner score (3'h220), a suppression was correctly applied. This simulation print also shows a particularity of interest, which is the fact of a comparison start running as soon as there are at least two available corners.

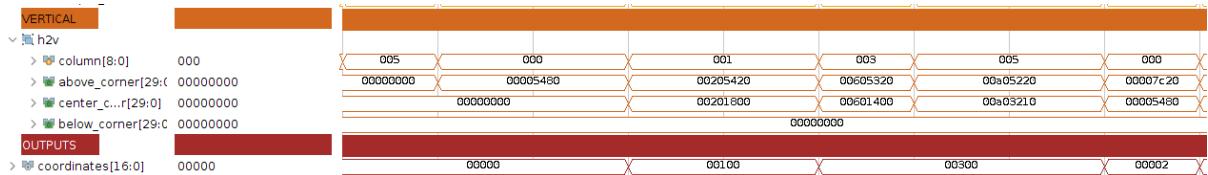


Figure 5.2.30: Vertical filter suppression.

Octaves NMS. No matter the octave, 3D suppression only starts if the above and below layers can already provide the correspondent pixel scores. To get that approval, as the following simulation shows, the last pixel coordinates from each layer are directly compared with the required ones. Thus, when the last saved pixel coordinates are greater than the desired one, the ready_a and ready_b from the closest layers get the logic level high. Finally, to execute the score comparison, we must assure both ready flags are active.

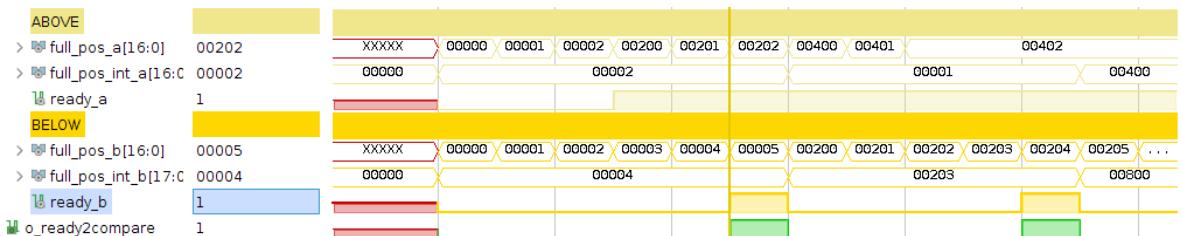


Figure 5.2.31: 3D flow control.

As expected, if the adjacent pixels score are both lower than the central layer one, the module will forward the pixel. Otherwise, it will be suppressed and the output present a null value, maintaining the o_read flag down.

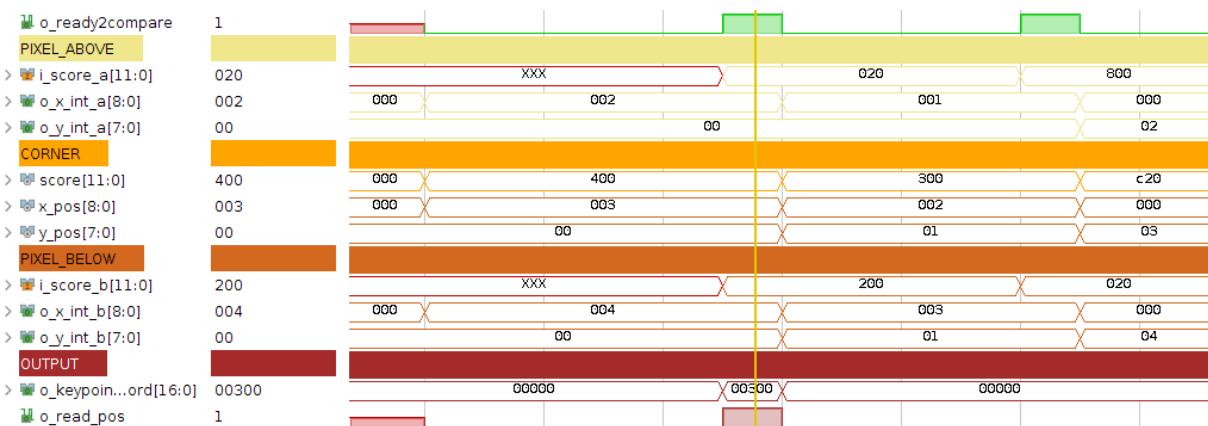


Figure 5.2.32: 3D comparison.

Unit and Stage Integration. Regarding what the aforementioned, one can observe that all the units performed as expected and, as the same piece of data was followed, there's clearly unit integration. Additionally, one step further was taken in the sense of transitioning to the next phase, the scoring stage, connecting it to the detection. This was already aforementioned and it can be seen in fig. 5.2.27 in red at the bottom. The fig. 5.2.33 represents the scoring stage, where the system sums all the bright pixel scores and all the dark pixel scores and chooses the one with the highest value. A simulation was made with both stages to consolidate the proper functioning of the scoring stage and also of the contiguity test. In fig. 5.2.34 it is possible to verify the correct function of the stage.

The outputs of the detection stage are marked in the pink rectangle. Marked with the blue rectangle, it is possible to visualise the four scoring stages where, in the last stage marked in green, it is possible to see the result in the r_o_score register. The contiguity test part is marked in peach and the result can be seen in the value of o_is_score.

With this it is possible to verify that everything is in sync because the result of the comparison that returns the highest score and the result of the contiguity test occur at the same time, proven by the rectangles marked in red. So it is possible to verify that everything is working as expected.

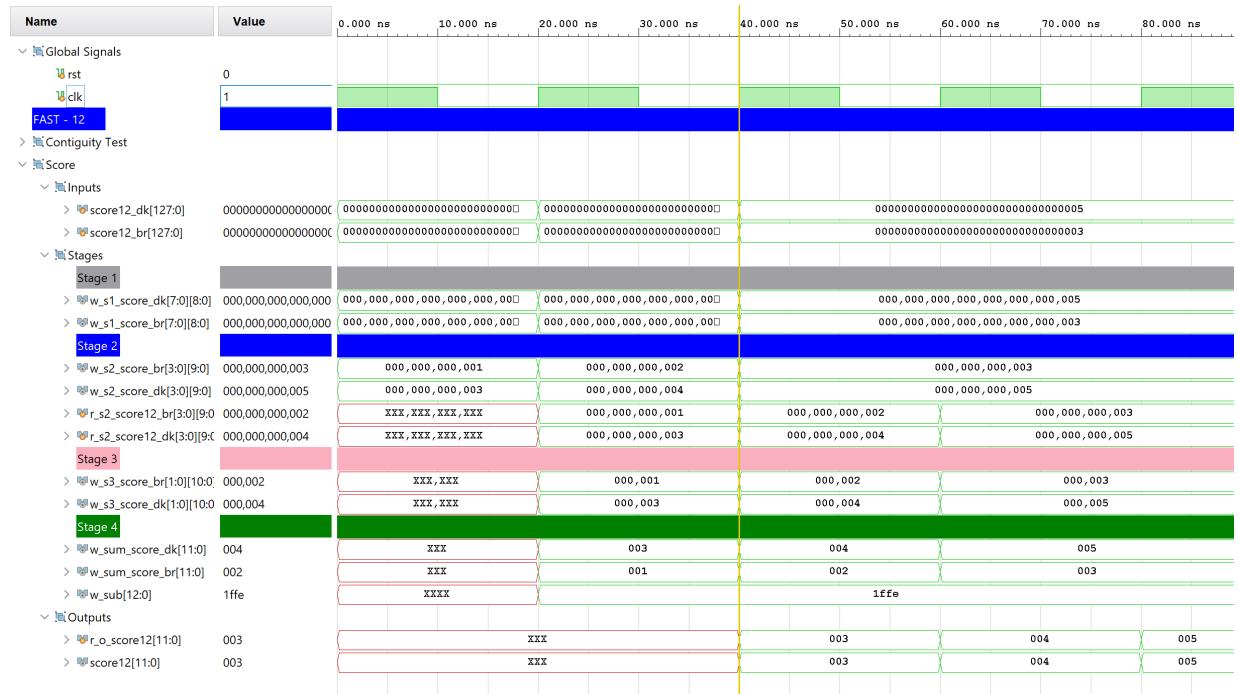


Figure 5.2.33: Simulation of FAST-12 Score

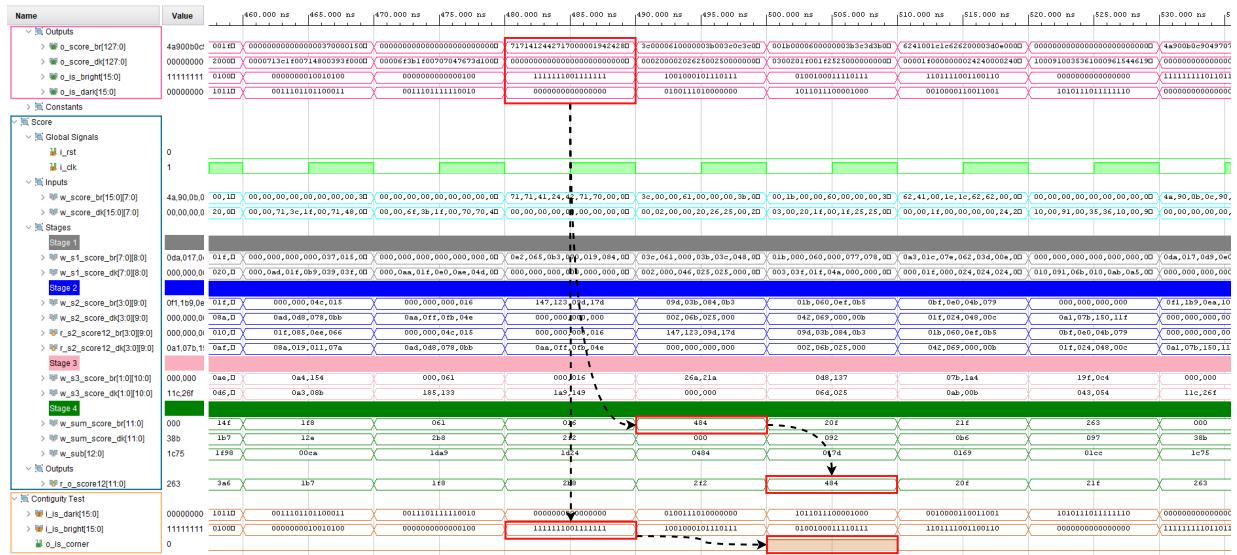


Figure 5.2.34: Integration Between Detection e Scoring Stages

To conclude the tests phase, the developed modules were all integrated with the next stage to verify the information dispatching. The last NMS stage is connected to the FIFO block which is read by the descriptor's first stage. The respective simulation results are also specified below.

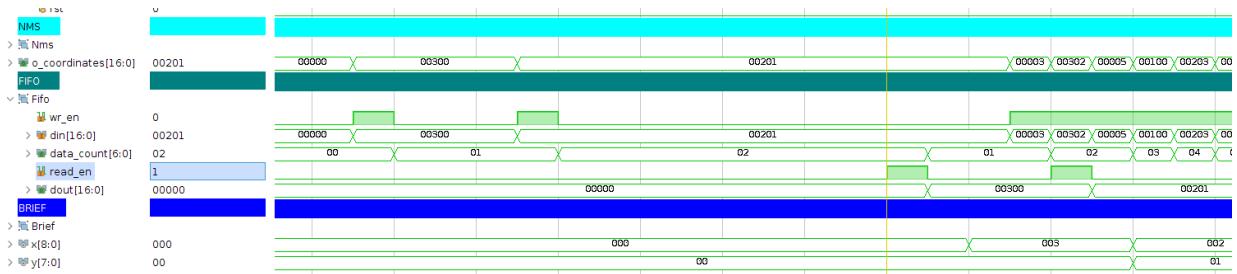


Figure 5.2.35: Output and BRIEF connection.

BRISK Software Refactoring: Detected Features. In the image 5.2.36 is presented the final result of the algorithm for the fast 12 and a threshold of 20. At green are the final keypoints that will be sent to the descriptor.

It is possible to verify that the keypoint are present in the locals where there is a variation in the intensity of the pixels.

BRISK Software Refactoring: Execution Time. The execution time was measured calculating the time between the initiation of the system and its finish.

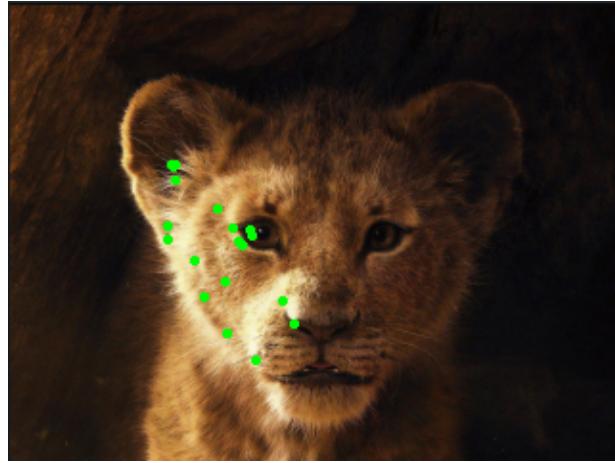


Figure 5.2.36: Final result

Listing 5.2: Platform-independent measurement of time with Boost

```

1 #include <iostream>
2 #include "BRISK_threads.h"
3 #include <boost/intrusive/list.hpp>
4 #include <boost/numeric/ublas/matrix.hpp>
5 #include <boost/date_time posix_time posix_time.hpp>
6 using namespace std;
7
8 int main()
9 {
10    BRISK_threads obj(matrixImageBlurred, 20, FAST12);
11    boost::posix_time::ptime start = boost::posix_time::microsec_clock::local_time();
12    Keypoint_l *lista4 = obj.compute_threads();
13    boost::posix_time::ptime end = boost::posix_time::microsec_clock::local_time();
14
15    boost::posix_time::time_duration timeTaken = end - start;
16    cout << "TIME: " << timeTaken.total_microseconds() << "us" << endl;
17 }
```

The system was executed several times in order to be possible to obtain the average of time that it takes to detect all the keypoints and to execute the NMS stage.

It was calculated the time that the initial code, the boost implementation and the final implementation (boost plus multi-threading) took.

In the table 5.4 are the values of the average execution time for the three implementation for the fast 12 using a threshold of 20. For smaller values of fast the execution time is lower and for a smaller value of the threshold the execution time is greater.

Table 5.4: Average Execution Time

Average Execution Time	Time (ms)
Initial Implementation	18.665
Boost Implementation	17.504
Final Implementation	7.996

5.2.4 BRIEF

This section will refer to the tests and result assessment performed in terms of the BRIEF modules.

FIFO Controller. To validate the correct operation of the FIFO controller module, it is possible to observe the Figure 5.2.37. As it can be seen, fifosync_valid is set from the moment the FIFO starts to have data and anytime mem_rdy is set, new keypoint coordinates are outputted, decreasing the FIFO internal counter. Since the flag i_new_img is set while the FIFO is receiving new inputs, it is also possible to verify the proper operation of the internal counter, being the output's 17th bit set when it reaches the value 1.

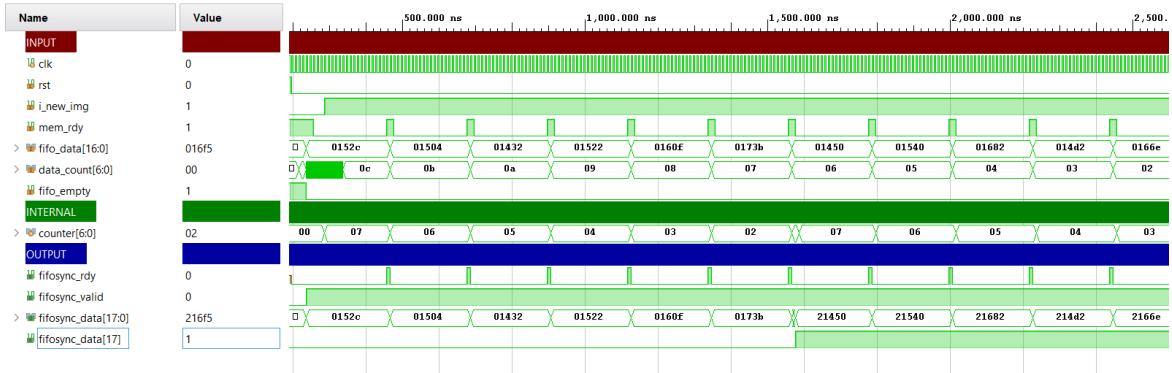


Figure 5.2.37: Simulation: FIFO Controller

Patch Computation. In order to validate the patch computation module, it is presented in the Figure 5.2.38 a simulation demonstrating its behaviour. To achieve the desired results, the FIFO preceding this module was filled with several keypoints after the reset, as can be seen by the persistent set value of the fifo_valid flag.

Analyzing the results, it is possible to observe that everytime mem_rdy is set, since fifo_valid is always set, the module receives new keypoint coordinates and lifts its flag mem_has_data, starting the processing of that input from that moment. When rf_rdy is set, the patch lines start being sent to the register file in the next clock cycle, as can be seen by the patchline output and the mem_valid flag, that evidently stays set for N clock cycles.



Figure 5.2.38: Simulation: Patch Computation

Register File. It is possible to verify in 5.2.39 the correct storing of the patch lines whenever the signals t_ready and t_valid are set. This simulation is a portion of a complete BRIEF module simulation where the patch size N is 15 and the description size is 256 bits.

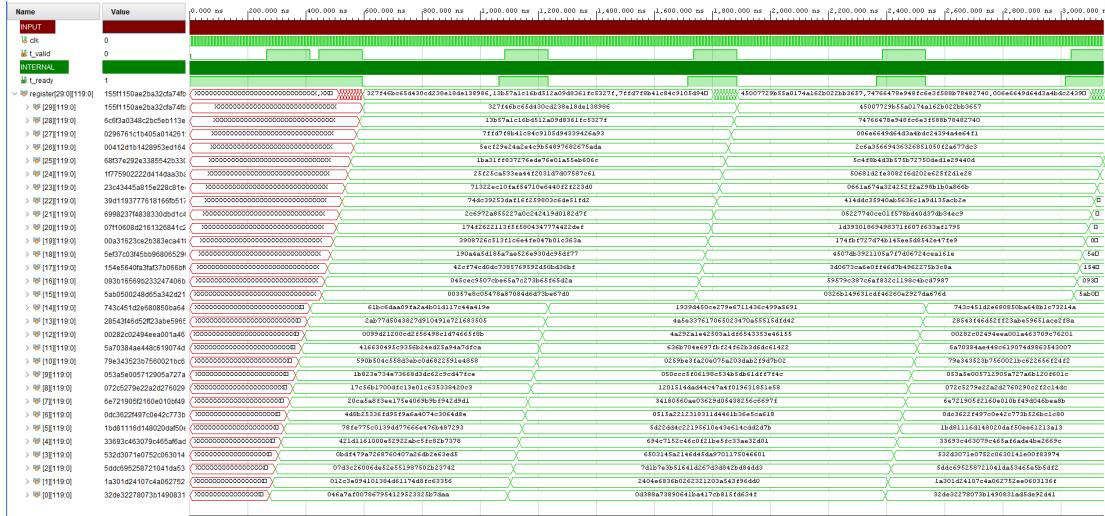


Figure 5.2.39: Simulation: Register File

To validate the output of the pairs as requested by the Binary Test module, a simulation is conducted where pairs are sent to the Register File module. As seen in the simulation 5.2.40, the pair1 sent is 2b521 which corresponds to point A being (5,13) and point B being (9,1). The pixel in the coordinates (5,13) is "fe" and can be seen concatenated in the output_a and the pixel correspondent to the coordinates (9,1) is "7f" which can be seen concatenated in the output_b. Only pair1 is shown in the simulation, but the same process applies to all four pairs which are all concatenated in the outputs a and b.

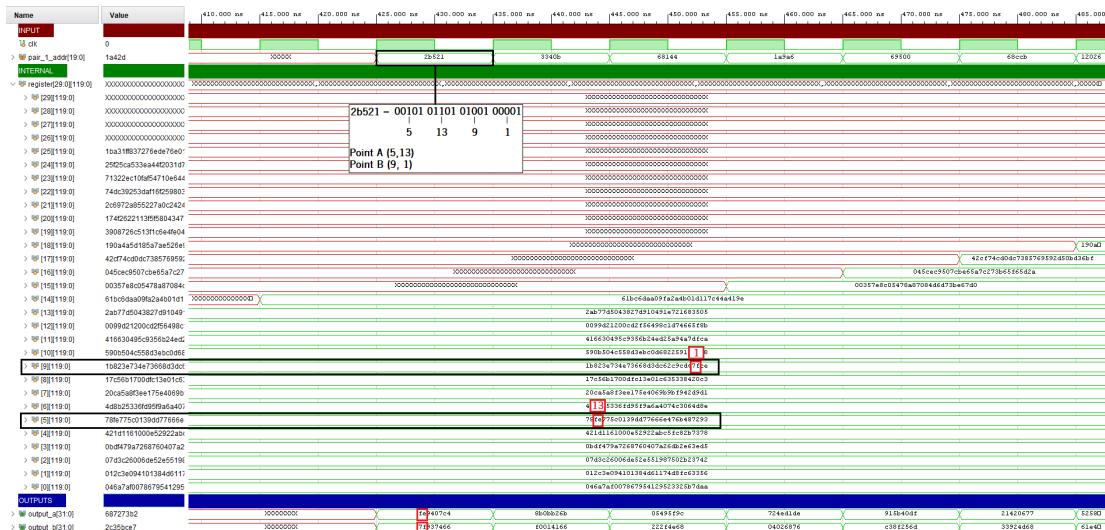


Figure 5.2.40: Simulation: Register File output pairs

Binary Test. For a descriptor length of 64 bits is expected that once a patch is available on the Register File, the binary test takes 16 clock cycles to complete its description. As one can see on the Simulation 5.2.41 when the flags patch_ready and patch_available become high, the binary test starts to generate the pair addresses and get them, at the same clock cycle, from the register file through pair_a and pair_b inputs. From that point, on each new clock cycle, four new binary tests are being made and the result of them concatenated to the descriptor. As is possible to observe, after 16 cycles, a description is complete.

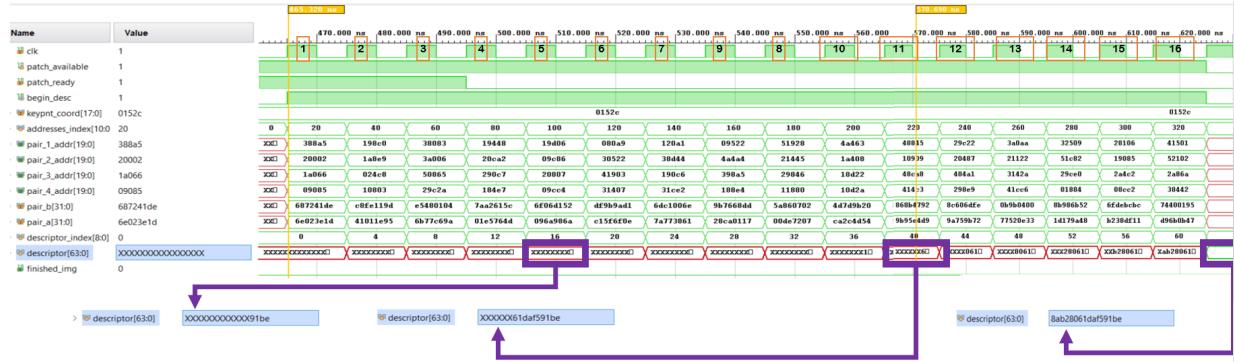


Figure 5.2.41: Simulation: Binary Test

Systolic Architecture. In order to verify the results of the systolic architecture is presented, in the Figure 5.2.42, the simulation of the Patch Computation module when inserted in this architecture. As it can be seen, when comparing this simulation with the simulation of the Figure 5.2.38, it is possible to see that there is always a register file ready to be filled with a new patch and thus, this module is only halted 3 clock cycles per keypoint, instead of the 42 clock cycles verified in the default architecture.

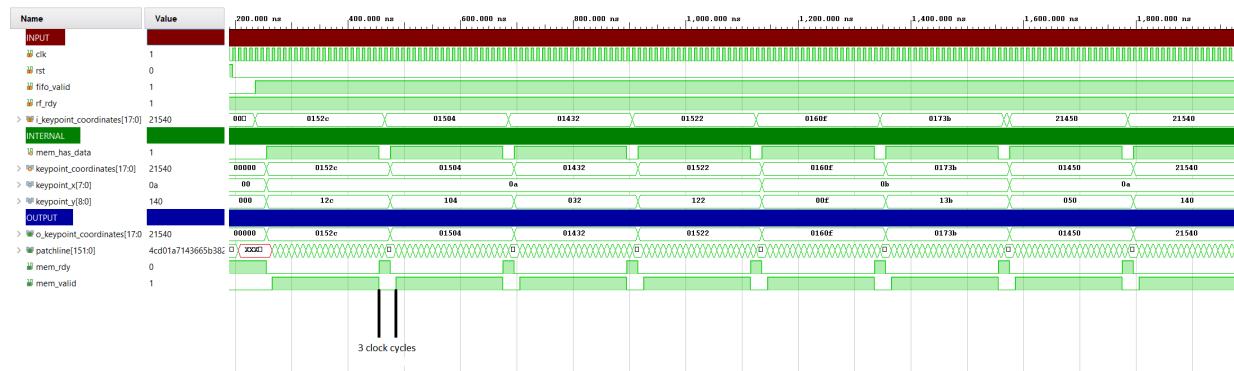


Figure 5.2.42: Simulation: Patch Computation with the systolic architecture

As for the whole BRIEF stage simulation, it is possible to verify that in the Figure 5.2.44, the descriptor of the first keypoint inputted takes 87 clock cycles to be outputted and then there is a latency of 64 clock cycles by keypoint.

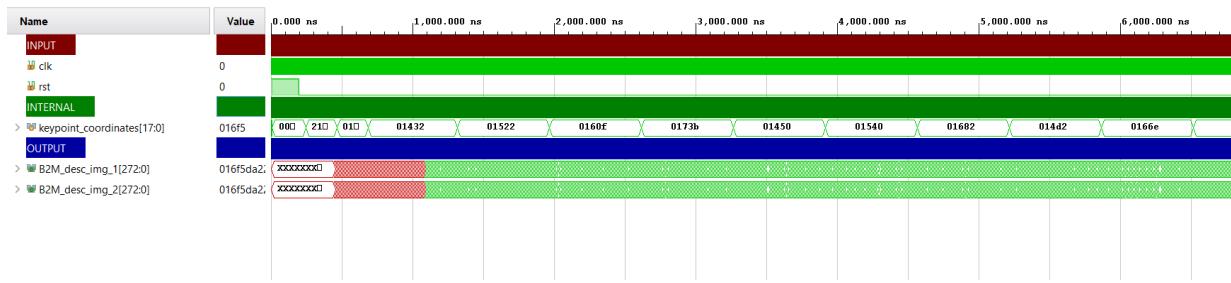


Figure 5.2.43: Simulation: BRIEF with the default architecture

On the other hand, when built with the systolic architecture mentioned before, the descriptor of the first keypoint inputted takes the same 87 clock cycles to be outputted but only 35 or 38 clock cycles in the subsequent keypoints.

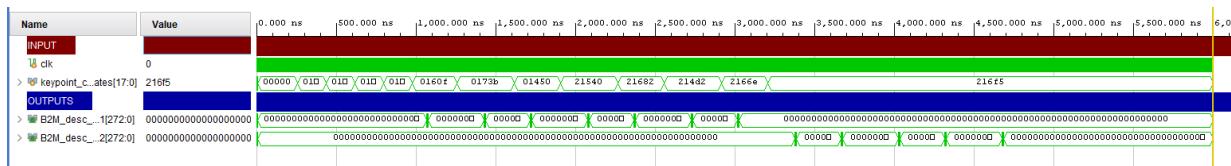


Figure 5.2.44: Simulation: BRIEF with the systolic architecture

Resource Utilisation

In this subsection the resource utilisation of both architectures is analysed for three combinations of the two customise parameters of the module, patch size and binary description string size. It is mandatory to analyse the extremes in order to obtain more trustworthy results, therefore the minimum and maximum resource utilisation scenario is analysed as well as a middle, more common scenario.

As seen in the next figures 5.2.45, 5.2.46 and 5.2.47, the resource utilisation gap between the default and systolic architectures is almost 50%. This is expected as the systolic architecture possesses two register file modules and two binary test modules which are both the ones who require the most resources. There are also two extra controller modules, however both consume negligible amount of resources.

In figure 5.2.45, it is possible to see the resource utilisation for both architectures in the minimum resource utilisation scenario.

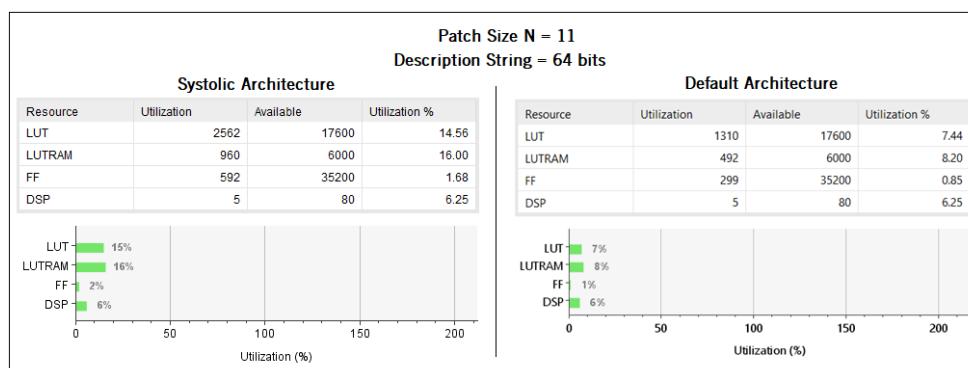


Figure 5.2.45: Resource Utilisation: N=11, 64 bit description string

In figure 5.2.46, it is possible to see the resource utilisation for both architectures in a reasonable scenario.

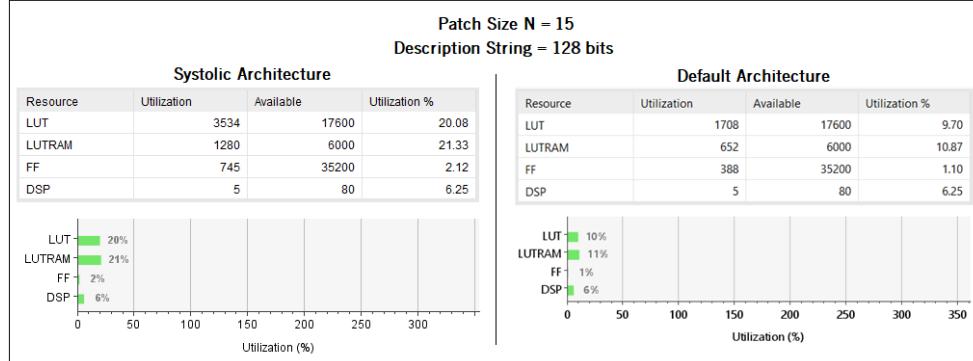


Figure 5.2.46: Resource Utilisation: N=15, 128 bit description string

In figure 5.2.47, it is possible to see the resource utilisation for both architectures in the maximum resource utilisation scenario.

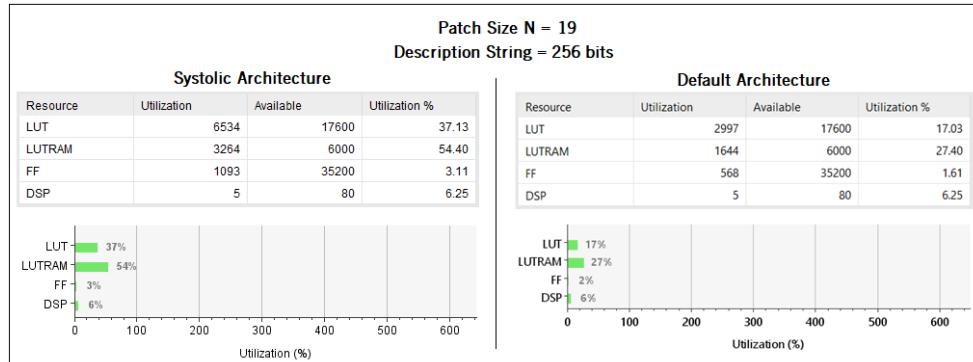


Figure 5.2.47: Resource Utilisation: N=19, 256 bit description string

BRIEF Software Refactoring: Verification of the pattern generation. For the verification of the pattern generation, the coordinates of each sampling pair calculated in the generation of the descriptor, was subtracted by the keypoint coordinates, the result obtained has verified by subtracting the values present on the txt, where if the final result is 0 the pattern generation was well made (as shown in the equation 5.1).

$$Values_{Calculated} - Keypoint_{Location} = Values_{file} \quad (5.1)$$

A practical example for this validation, is for example a sampling pair with the coordinates [18,37,20,33], the keypoint was the following coordinates [16,36], and the offsets presented in file are [2,1,4,-3]. As it can be seen in equation 5.2 the result of the subtraction of each sampling pair with the keypoint coordinate, gave the exact same offset as the one present in the txt file, indicating that the calculus was well made. This process was done more efficiently using excel as a medium (displayed in figure 5.2.49).

$$[18, 37, 20, 33] - [16, 36, 16, 36] = [2, 1, 4, -3] \leftrightarrow [2, 1, 4, -3] = [2, 1, 4, -3] \quad (5.2)$$

Keypoint		Points calculated by implementation				Points calculated by the implementation minus the keypoint				Offset presented in the file				OffsetFile-OffsetCalculated			
x	y	x	y	x	y	x	y	x	y	x	y	x	y	x	y	x	y
16	36	18	37	20	33	2	1	4	-3	2	1	4	-3	0	0	0	0
17	35	19	39	1	-1	3	3	1	-1	3	3	0	0	0	0	0	0
15	39	15	39	-1	3	-1	3	-1	3	-1	3	0	0	0	0	0	0
18	39	21	34	2	3	5	-2	2	3	5	-2	0	0	0	0	0	0
17	40	12	32	1	4	-4	-4	1	4	-4	-4	0	0	0	0	0	0
17	35	12	34	1	-1	-4	-2	1	-1	-4	-2	0	0	0	0	0	0
17	31	17	35	1	-5	1	-1	1	-5	1	-1	0	0	0	0	0	0
15	41	19	41	-1	5	3	5	-1	5	3	5	0	0	0	0	0	0
11	35	16	32	-5	-1	0	-4	-5	-1	0	-4	0	0	0	0	0	0
17	41	14	36	1	5	-2	0	1	5	-2	0	0	0	0	0	0	0
18	36	21	32	2	0	5	-4	2	0	5	-4	0	0	0	0	0	0
13	35	19	40	-3	-1	3	4	-3	-1	3	4	0	0	0	0	0	0

Figure 5.2.48: BRIEF software implementation validation of the sampling pattern

BRIEF Software Refactoring: Validation of the Descriptor Computation and Repeatability. Using Matlab and a pre-determined image, the keypoints selected and the sampling pattern used were saved into a txt file as well as the descriptor for each keypoint, allowing to later validate the implementation. Then the code implemented was run, and the values were compared.

Figure 5.2.49: BRIEF software implementation validation of the descriptors

As seen in the image above, the result given by the implemented code is always exactly the same verifying the correct implementation of the BRIEF feature descriptor, this test was made multiple times allowing to verify that not only the computation is done as intended but that it consistently generates the same description allowing two different systems to run the process and obtain the exact same result.

BRIEF Software Refactoring: Execution Time. Generally the execution time from a program is measured from program initiation at presentation of some inputs to termination at the delivery of the last outputs. Several different measures of software performance are of interest.

- Worst-case execution time (**WCET**)—the longest execution time for any possible combination of inputs:

- Best-case execution time (**BCET**)—the shortest execution time for any possible combination of inputs;
- Average-case execution time for typical inputs.

Average-case performance is used for very different purposes than are worst-case and best-case execution times, being normally used to tune software. With this in mind and the resources provided by the boost library the execution time was measured using the code displayed below in listing 5.3.

Listing 5.3: Platform-independent measurement of time with Boost

```
#include <boost/date_time posix_time posix_time.hpp>
/* [...] */

3 int main() {
    boost::posix_time::ptime start = boost::posix_time::microsec_clock::local_time();
    // do something time-consuming
    boost::posix_time::ptime end = boost::posix_time::microsec_clock::local_time();
8
    boost::posix_time::time_duration timeTaken = end - start;

    std::cout << std::endl;
    std::cout << "TIME: " << timeTaken.total_seconds()      << " s" << std::endl;
13   std::cout << "TIME: " << timeTaken.total_milliseconds() << " ms" << std::endl;
    std::cout << "TIME: " << timeTaken.total_microseconds() << " us" << std::endl;
    std::cout << "TIME: " << timeTaken.total_nanoseconds()  << " ns" << std::endl;

    return 0;
18 }
```

Running the implemented code with the code listed above integrated in it, a average value for the execution time of each stage the of development was done. In table 5.5, is possible to notice that although the boost library accelerates the process is not that significant when compared to the implementation without it. The implementation of multi-threading gave a significant improve to the execution time being almost twice as fast when compared with the previous implementation.

Table 5.5: Average Execution Time of BRIEF implementations

Average Execution Time	Seconds (μs)
Initial Code	15801
Boost Implemented	12691
Threads Implemented	6495

Throughout the course of the tests for the execution time the software implemented was making 202 descriptions, so taking into account the average time of the last implementation (with the multi-threading), each keypoint took on average $32.15 \mu s$.

5.2.5 Matching

This section will refer to the tests and result assessment performed in terms of matching.

Matching Core Simulation. As a way to simulate and test the matching core in hardware, a simulation was generated in which its inputs were set to 102 and 80 (pattern 1 = 102 and pattern 2 = 80). The figure 5.2.50 represents the generated simulation.

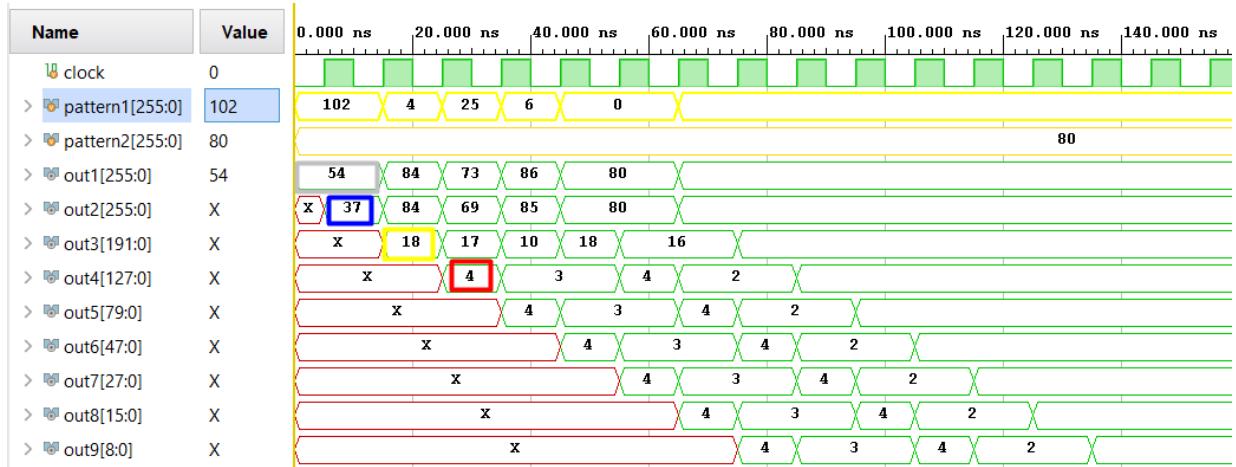


Figure 5.2.50: Matching Core Simulation

As can be seen in figure 5.2.50 the variable out1 will result in the XOR operation between the two patterns and the remaining variables (out2, out3, out4) will contain the value of the sums explained above. For a better understanding of the operations performed on the matching core the fig. 5.2.51 was developed. In fig. 5.2.51 it is possible to see the result of the xor between the values 102 and 80, as well as the several sums that the xor result will suffer.

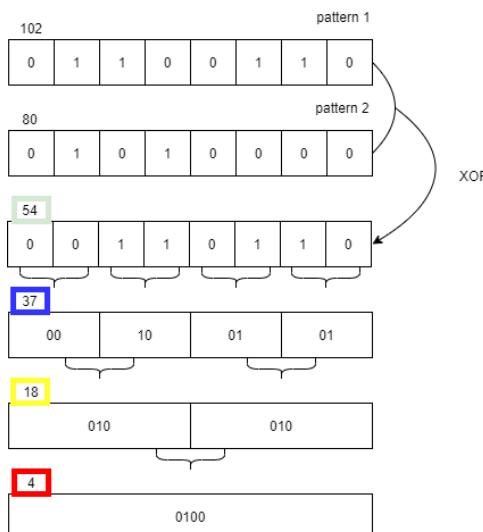


Figure 5.2.51: Matching Core Simulation

After calculating the hamming distance it is necessary to save the index of the descriptor with the smallest hamming distance and as a way to show this process the simulation of fig. 5.2.52 was generated.

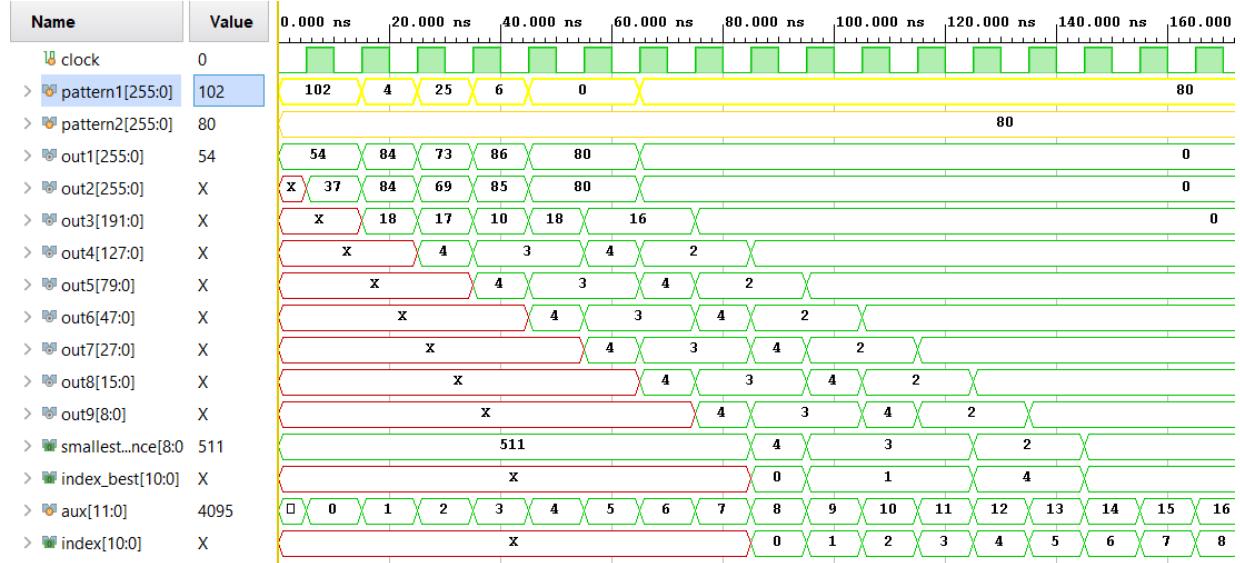


Figure 5.2.52: Matching Core Simulation with index of the best selection

In the simulation (fig. 5.2.52) it is possible to verify the change of the `index_best` variable that stores the index value of the descriptor with the smallest hamming distance from the common part being compared. This variable is updated with a new value whenever a descriptor with a new smallest hamming distance is found.

In the tests performed in the simulation (fig. 5.2.52) it is possible to see the change of the `index_best` from 0 to 4 since the descriptor with index 4 has a smaller hamming distance than the descriptor with index 0, for the pattern being compared.

BRAM Simulation. In order to be able to verify the implementation, a test was developed, in which different data are assigned, in order to simulate the descriptors, and where it will be possible to see the memory storing those same values.

On the following figures is showed a simulation where some values into BRAM one and two are stored.



Figure 5.2.53: BRAM Simulation

With the following simulations it is possible to verify that after the reset and after the enable image 1 is on it is possible to store values in the memory, having tested the values 53, 89 and 300. It is possible to see in the fig. 5.2.53 that the value 53 is stored at address 0, the other values being stored in the same way.

The same reasoning is done for the BRAM two. In that case the next fig. 5.2.54 shows that the enable of the image two is on, and so, the data is saved on the specified address.

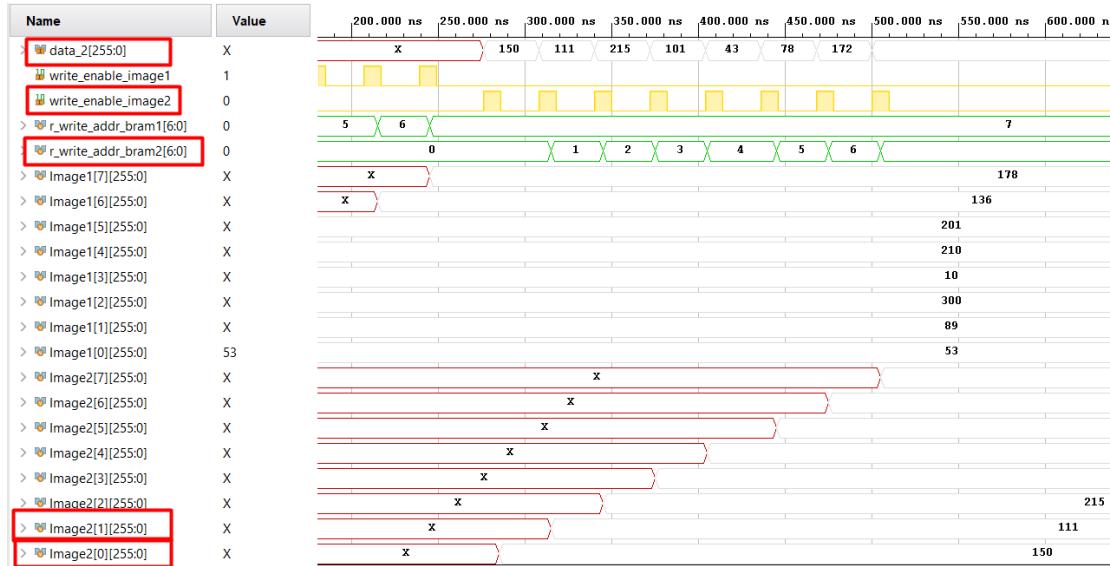


Figure 5.2.54: BRAM Simulation

Matching Controller Simulation. The patterns inserted to the BRAM are presented on the next table. The Best matches are calculated as well as the smallest differences, to later be proved on simulation. Every descriptor of image 1 is compared to all the descriptors of image two.

Descriptor Image 1	Descriptor Image 2	Smallest Difference	Index Best
150	53	2	4
111	89	4	0
215	300	2	4
101	10	2	0
43	210	2	3
78	201	2	3
172	136	2	2
205	178	1	5

Table 5.6: Smallest Difference and Index of best Matches

In the figure 5.2.55 is possible to see that when the result ready comes to one, the first pattern (150) when compared to all the others, presents the index 4 from the common part as the best match, being 2 the smallest difference.

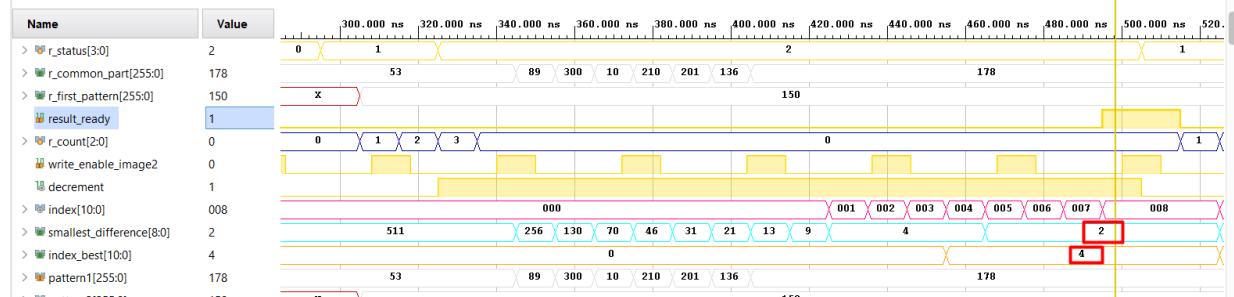


Figure 5.2.55: Controller Simulation

The next simulation 5.2.56 represents the brute force algorithm between the descriptor 111 and all the others. The index of best and smallest difference are in accordance with the table presented before.

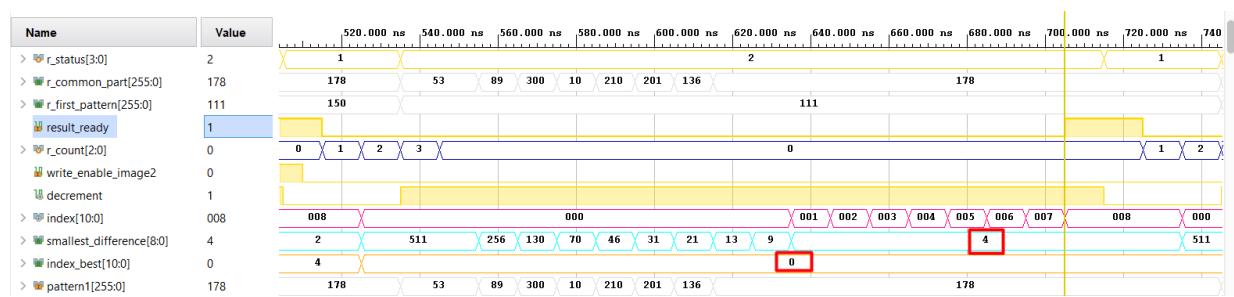


Figure 5.2.56: Controller Simulation

Cross Checking Simulation. The controller has the ability to perform cross checking in order to remove the various false matches that may exist after the matching process.

As a way to separate true descriptors from false ones, a technique was adopted in which the descriptor that is considered a false match will be stored in the best_match register with the valid bit (most significant bit of the register) equal to zero. In other words, a descriptor that is a true match will be stored in the respective register with the valid bit set to 1. As one can see in the simulation of fig. 5.2.57, the registers marked in red indicate true matches (valid bit = 1).

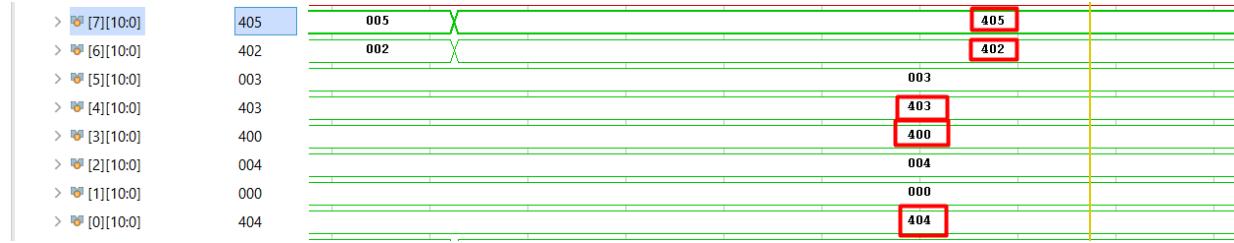


Figure 5.2.57: Cross Checking simulation

In the simulation of ?? it is possible to see that the best matching for the descriptor 150 of image 1 is the descriptor 10 of image 2 (index = 4 -> 404). As the valid bit is one, it can be said that this matching point is valid.

Matching Controller Simulation Parallelism. A comparison was developed between the controller when only one matching core is used and when 3 cores are used. The comparison should reflect on the values in terms of speed, and when 3 cores are used, the speed at which the algorithm finishes should be higher, taking less time to execute the algorithm. With this, it is possible to observe in the following figures 5.2.58, 5.2.59 that the use of 3 matching cores reduced the execution time of the algorithm.

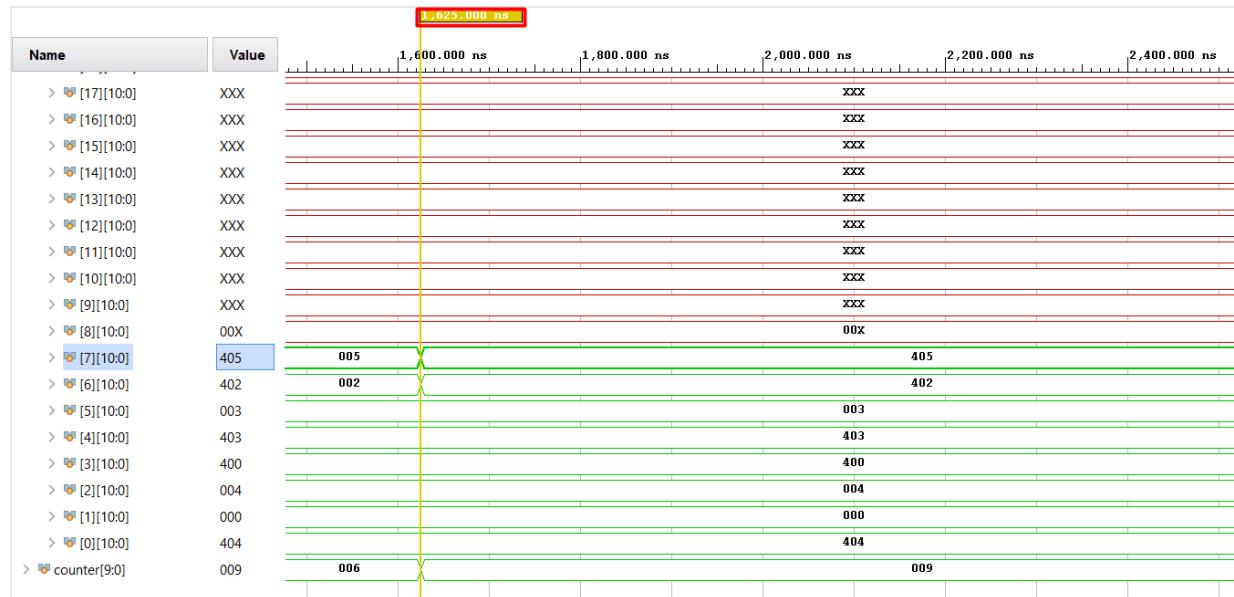


Figure 5.2.58: Parallelism simulation

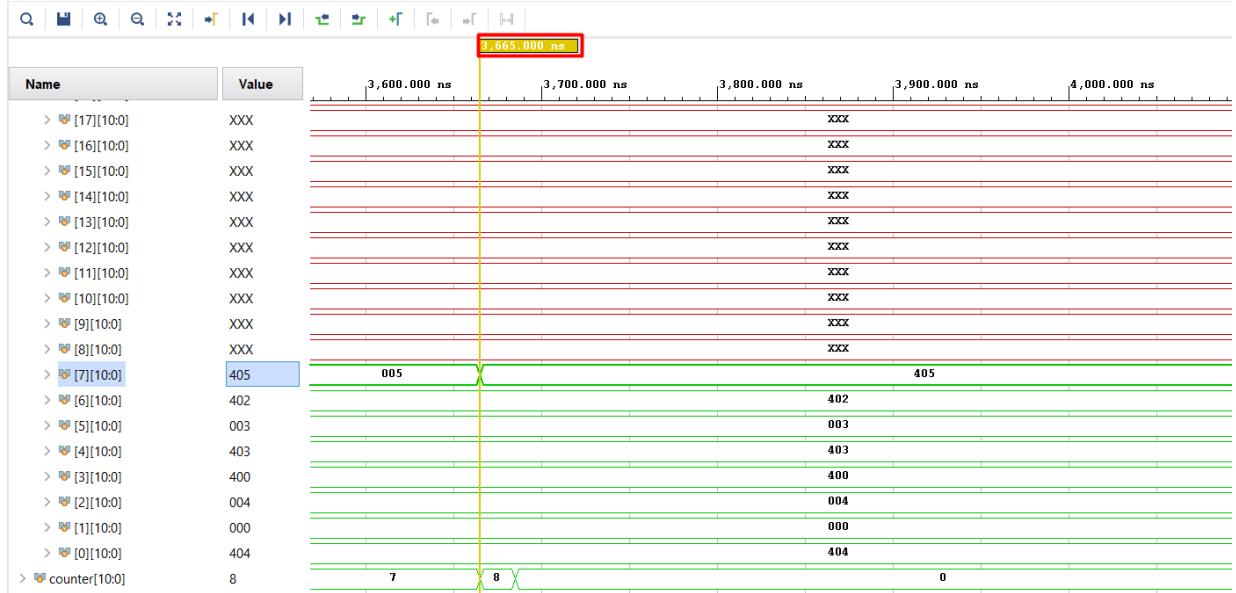


Figure 5.2.59: Parallelism simulation

Matching Software Refactoring: Validation. The validation will consist of, as mentioned in the beginning, converting the obtained keypoints and matches into *openCV* objects so that the **drawMatches** function can create a graphic with the image with lines for each match.

Converting the objects into *openCV* objects and creating said graphic is done by the **drawImage** function:

Listing 5.4: drawImage function

```

2   void drawMatchingImage(cv::Mat image1, cv::Mat image2, matching_l& matchingList) {
3     /* [...] */
4     for(auto & it : matchingList) {
5       /* [...] */
6       keyPointsVector1.push_back(keyPoint1);
7       keyPointsVector2.push_back(keyPoint2);
8       /* [...] */
9       matchesVector.push_back(matches);
10      /* [...] */
11    }
12    cv::Mat img_matches;
13    drawMatches(image1, keyPointsVector1, image2, keyPointsVector2, matchesVector, img_matches);
14    cv::namedWindow("Matches", cv::WINDOW_NORMAL);
15    imwrite("../IMAGE/img_matched.png", img_matches);
16    imshow("Matches", img_matches );
17  }

```

Figures 5.3.1 and 5.3.6 consist of, in the upper area, the output of the scenario where *openCV* is used completely and the lower area the scenario where the refactored software is used. In terms of quality and performance, it should be noted that a few outliers are not filtered in the refactored version, this is due to the outlier removal algorithm that either does not have the optimal threshold or is not the most suited to the operation. It should also be noted that the refactored version is faster than the *openCV* which can justify the small drop in quality.

5.3 Software Refactoring Integration

Listing 5.5: Software Refactoring Integration

```

int main() {
    std::cout << "Software Implementation - ESRG PROJECT" << std::endl;
    /* [...] */
    cv::Mat image1 = cv::imread(imagePath1, cv::IMREAD_COLOR);
    cv::Mat image2 = cv::imread(imagePath2, cv::IMREAD_COLOR);
    /* [...] */
    boost::numeric::ublas::matrix<uchar> imageMatrix1 = convert2matrix(image1Blurred);
    boost::numeric::ublas::matrix<uchar> imageMatrix2 = convert2matrix(image2Blurred);
    /* [...] */

    /***** Detect Phase *****/
    * ----- Detect Phase ----- *
    *****/
    auto* detector1 = new BRISK_threads(imageMatrix1, 25, FAST12, 2);
    auto* detector2 = new BRISK_threads(imageMatrix2, 25, FAST12, 2);
    /* [...] */
    Keypoint_l* keyPointsList1 = detector1->compute_threads();
    Keypoint_l* keyPointsList2 = detector2->compute_threads();
    /* [...] */

    /***** Description Phase *****/
    * ----- Description Phase ----- *
    *****/
    auto *descriptor1 = new briefThreads(PATCH11, BRIEF8, imageMatrix1, 2, keyPointsList1);
    auto *descriptor2 = new briefThreads(PATCH11, BRIEF8, imageMatrix2, 2, keyPointsList2);
    /* [...] */
    BRIEF_l* descriptorsList1 = &descriptor1->compute_threads();
    BRIEF_l* descriptorsList2 = &descriptor2->compute_threads();
    /* [...] */

    /***** Matching Phase *****/
    * ----- Matching Phase ----- *
    *****/
    auto *matching = new matcher(3);
    /* [...] */
    matching_l* matchingList = &matching->compute(descriptorsList1, descriptorsList2);
    /* [...] */
    drawMatchingImage(image1, image2, *matchingList);

    delete detector1;
    delete detector2;
    delete descriptor1;
    delete descriptor2;
    delete matching;
    /* [...] */
    std::cout << std::endl;
    std::cout << "----- Detection Time -----" << std::endl;
    std::cout << "TIME: " << timeTaken_detection.total_seconds() << " s" << std::endl;
    std::cout << "TIME: " << timeTaken_detection.total_milliseconds() << " ms" << std::endl;
    std::cout << "TIME: " << timeTaken_detection.total_microseconds() << " us" << std::endl;
    std::cout << "TIME: " << timeTaken_detection.total_nanoseconds() << " ns" << std::endl;
    /* [...] */
    return EXIT_SUCCESS;
}

```

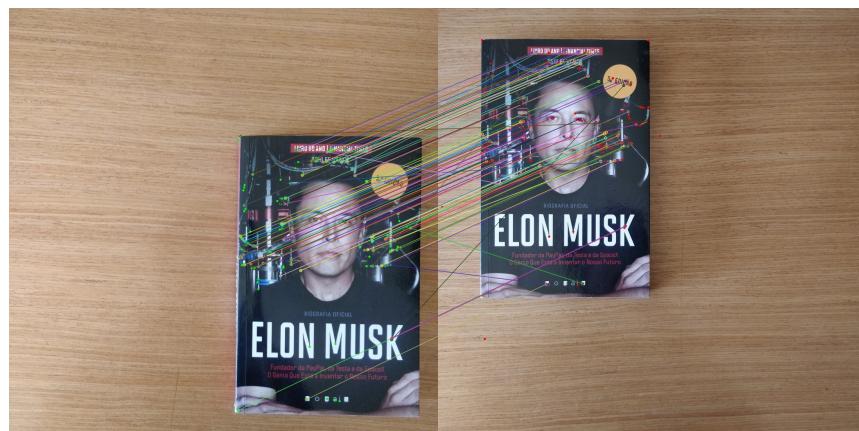


Figure 5.3.1: Test 1

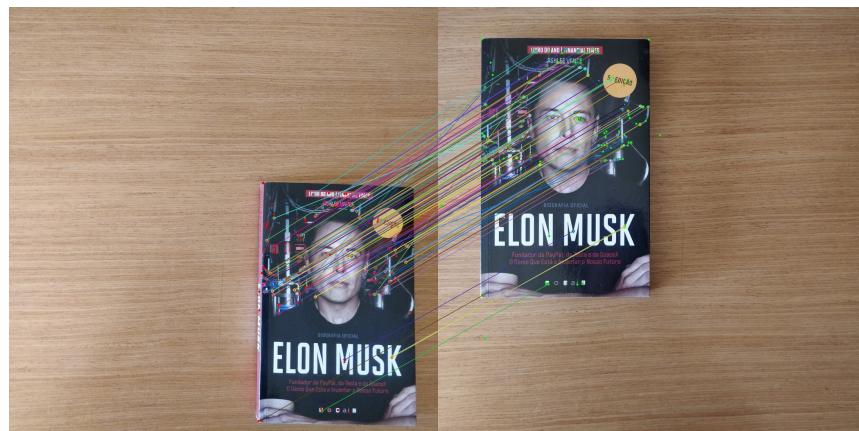


Figure 5.3.2: Test 2



Figure 5.3.3: Test 3

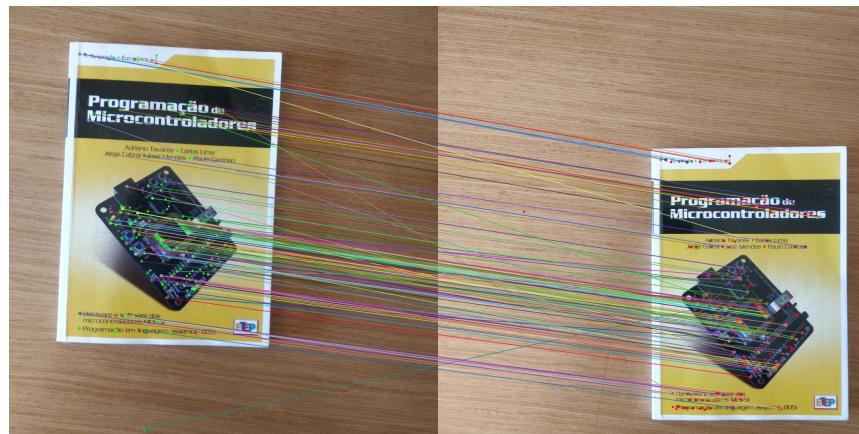


Figure 5.3.4: Test 4



Figure 5.3.5: Test 5

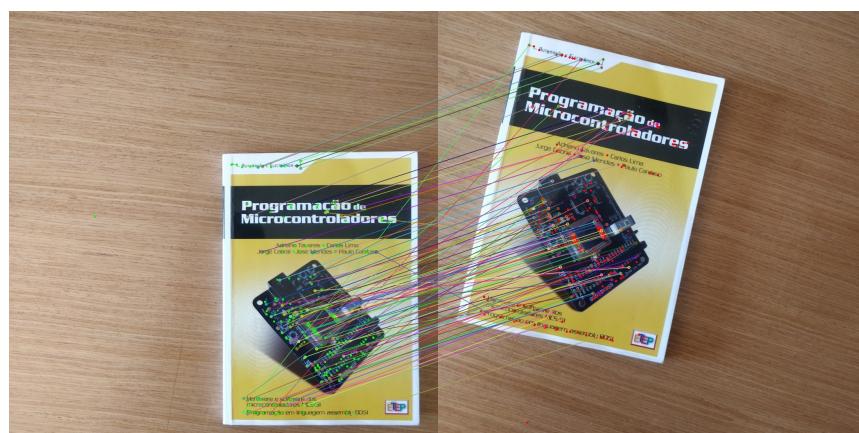


Figure 5.3.6: Test 6

Listing 5.6: Software Refactoring Integration Timing Outputs

```
1 ----- Detection Time -----
TIME: 0 s
TIME: 263 ms
TIME: 263185 us
TIME: 263185000 ns
6
----- Description Time -----
TIME: 0 s
TIME: 55 ms
TIME: 55594 us
11 TIME: 55594000 ns

----- Matching Time -----
TIME: 0 s
TIME: 779 ms
16 TIME: 779828 us
TIME: 779828000 ns
```

5.4 Profiling and Benchmarking

5.4.1 VTune Profiling of the ORB Descriptor

VTune has many tools that allow the user to analyse the behaviour of the system during a period of time, this type of data can be relevant to study the influence of some parameters/routines on the whole system.

Influence of Keypoint Number. One started by studying the influence of the amount of keypoints detected by the system on the whole program. The following points explain this test's procedure. It started by executing the program with VTune running on background. During this execution the program was presented two images, one with many keypoints to be detected and one with few keypoints. After a few time the profiling process was stopped and the results were analysed.

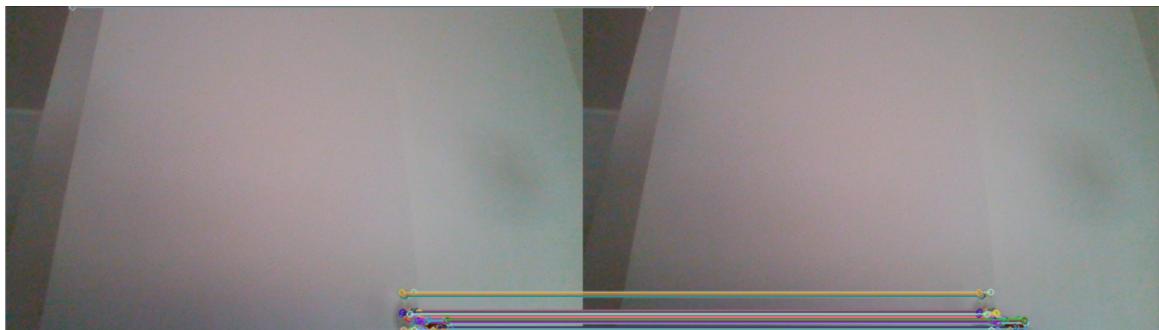


Figure 5.4.1: Test: Fewer Keypoints



Figure 5.4.2: Test: Additional Keypoints

Test Results. Before finish the results, one had already concluded a few things. First would be the influence of the VTune overhead and the amount of keypoints detected on the execution of the program. The execution of the program with VTune running was much much slower, this was an effect that wasn't verified with OProfile, with OProfile the execution didn't seem to get affected much. The amount of keypoints also played a major role on the execution of the program, with Few keypoints the execution of the program was much faster than the execution with many keypoints. Following are represented the results of this test.

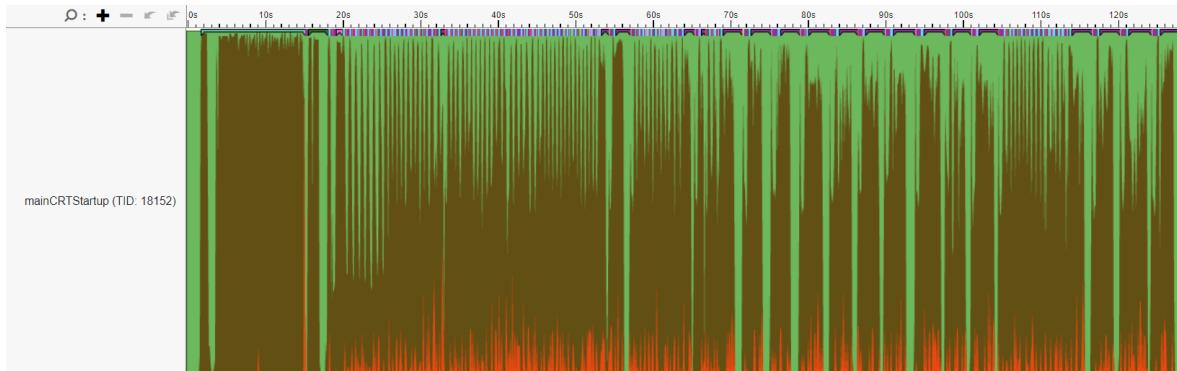


Figure 5.4.3: Test: Execution with Additional Keypoints

With this analysis it was possible to confirm our suspicions that the number of keypoints detected was making the process much slower. Has can be seen in the picture bellow three different stages of the program execution were detected. The stage represented with purple is the startup of the program, the stage represented with yellow is the moments where the picture had few keypoints, the stage represented with blue is the moment where the picture had many keypoints. The process is much slower when there are many keypoints.

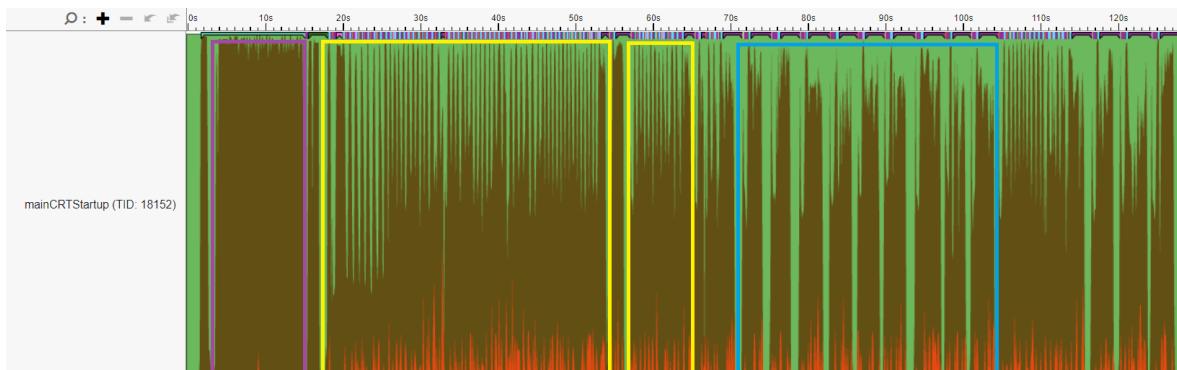
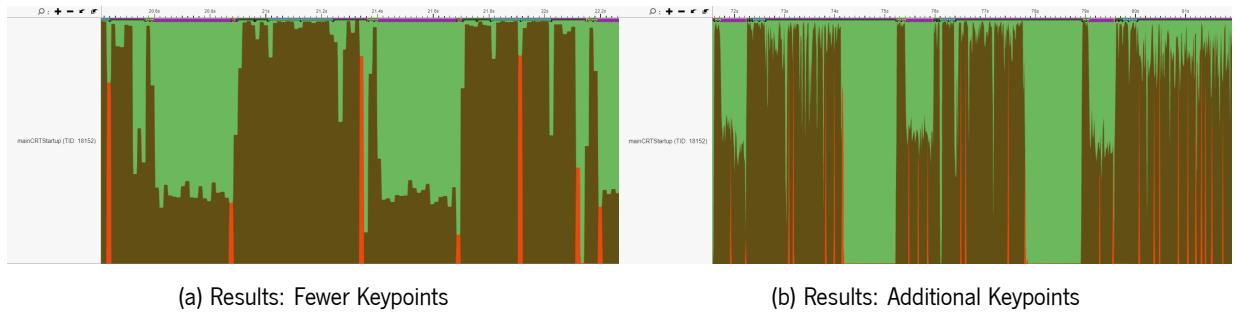


Figure 5.4.4: Results: Execution with Additional Keypoints vs Fewer Keypoints

To understand what was making the process so much slower when there were many keypoints, one proceeded to better analyse the results by zooming in in each stage of the program.



In the top slider can be seen some colour which identify which routine is executing in each moment of the execution, the following list labels each colour to each routine.

- **Dark Purple** - Matching Routine
 - **Light Purple** - Draw Lines and Circles
 - **Light Blue** - Compute Routine (BRIEF)
 - **Dark Green** - Detect Routine (FAST)

Evaluation. With this data it is possible to conclude what was the origin of the problem. Basically the matching process gets very slow when there are too many keypoints. This can be seen as a Bottleneck because the matching process is slowing down the flow of the whole system. This problem is easily solved by limiting the number of points so this leads to one conclusion, this algorithm could be greatly improved by adding an algorithm like non-maximal suppression before proceeding to the matching routine. The non maximal suppression algorithm is used to reduce the number of keypoints detected by analysing the keypoints which are too close and then only output the ones that have the highest score.

BRIEF CPU load. VTune gives data to the user about how much CPU load each function is consuming. One used this to find-out what were the routines from the BRIEF algorithm that were consuming more CPU load.

▼ ORB_Class::Brief		3.4%	0s	opencv...	ORB_Class::...	opencv-te...	0x140001c70
▼ cv::Feature2D::compute		3.4%	0s	opencv...	cv::Feature2...	feature2d...	0x182ee12d0
▼ cv::ORB_Impl::detectAndCompute		3.4%	0s	opencv...	cv::ORB_Impl...	orb.cpp	0x182f32f00
► cv::GaussianBlur		2.1%	0s	opencv...	cv::Gaussian...	smooth.d...	0x182893f00
► cv::resize		0.9%	0s	opencv...	cv::resize(dla...	resize.cpp	0x18284de90
► cv::cvtColor		0.2%	0s	opencv...	cv::cvtColor(...	color.cpp	0x1824c1ba0
► cv::computeOrbDescriptors		0.1%	0.154s	opencv...	cv::compute...	orb.cpp	0x182f2f360
► cv::copyMakeBorder		0.0%	0s	opencv...	cv::copyMak...	copy.cpp	0x18204a880
► cv::Mat::create		0.0%	0s	opencv...	cv::Mat::crea...	mat.inl.hpp	0x181f4ac10
► cv::utils::trace::details::Region::Region		0.0%	0s	opencv...	cv::utils::trac...	trace.cpp	0x182348550
► cv::Mat::Mat		0.0%	0s	opencv...	cv::Mat::Ma...	mat.inl.hpp	0x181f3df50
► std::copy<cv::Point_<int>, const *,std::back_insert_iterator<std::vector<cv::Point_<int>, class std::allocator<cv::Point_<int>> >		0.0%	0.030s	opencv...	std::copy<cv::...	xutility	0x182f2fa60
► cv::debug_build_guard::InputArray::empty		0.0%	0s	opencv...	cv::debug_b...	matrix_wr...	0x1821fa860
► std::vector<class cv::Point_<int>, class std::allocator<class cv::Point_<int>> >		0.0%	0s	opencv...	std::vector<cl...	vector	0x182057a60

Figure 5.4.6: Brief CPU Load

As it can be seen in the result the calculus of the Gaussian filter is the function that consumes the most CPU load.

ORB Hotspots. VTune gives a general view of the routines that are consuming the most CPU load (Hotspots). By making this analysis it was possible to confirm that the routine which is consuming the most CPU load is the matching routine.

Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Time	Task Count	Average Task Time
void __cdecl cv::DescriptorMatcher::match(const class cv::debug_build_guard::InputArray &,const class cv::debug_build_guard::InputArray &,class std::vector<class cv::DMatch,class std::allocator<class cv::DMatch>> &)	52.200s	98	0.533s
void __cdecl cv::Feature2D::compute(const class cv::debug_build_guard::InputArray &,class std::vector<class cv::KeyPoint,class std::allocator<class cv::KeyPoint>> &,const class cv::debug_build_guard::OutputArray &)	22.601s	196	0.115s
bool __cdecl cv::VideoCapture::open(int,int)	13.478s	1	13.478s
void __cdecl cv::Feature2D::detect(const class cv::debug_build_guard::InputArray &,class std::vector<class cv::KeyPoint,class std::allocator<class cv::KeyPoint>> &,const class cv::debug_build_guard::InputArray &)	12.892s	196	0.066s
void __cdecl cv::circle(const class cv::debug_build_guard::InputOutputArray &,class cv::Point_<int>,int,const class cv::Scalar_<double> &,int,int,int)	12.449s	34,092	0.000s
[Others]	12.867s	17,837	0.001s

*N/A is applied to non-summable metrics.

Figure 5.4.7: ORB Hotspots

5.4.2 MSVC/ICC Results for BRISK Detection

With VTune, several analysis were made. The code was analysed by VTune Hotspot Feature for three different timescales. First for 60 seconds, then for 600 seconds, and lastly for 1800 seconds. This was made for a more reliable set of information and terms of comparison between tests.

MSVC Results

This section will present the results the for the profiling and benchmarking using MSVC.

Timing Analysis. This paragraph will present the results the for the profiling and benchmarking using MSVC, in terms of timing analysis.

Table 5.7: MSVC Time Table

	10 Tasks	100 Tasks	1000 Tasks	10000 Tasks
\bar{x} (ms)	43.82994	48.87546	54.55777	52.14287
s (ms)	4.982791	5.920759	17.10567	13.63817

Elapsed Time	60.787	599.755	1800.715
CPU Time	53.636	572.009	1711.819
Effective Time	53.375	569.380	1704.188
Idle	1.041	2.067	8.826
Poor	52.335	567.313	1695.362
Ok	0.000	0.000	0.000
Ideal	0.000	0.000	0.000
Over	0.000	0.000	0.000
Spin Time	0.260	2.629	7.631
Overhead Time	0.000	0.000	0.000
Thread Count	1.000	1.000	1.000
Paused Time	0.000	0.000	0.000

everything expressed in seconds.

Figure 5.4.8: BRISK Detection Time Analysis

Top Hotspots. This paragraph will present the results for the profiling and benchmarking using MSVC, in terms of the hotspots/bottlenecks found.

🕒 Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time ⓘ
<code>std::vector<float, class std::allocator<float> >::operator[]</code>	opencv_world420d.dll	693.751s
<code>std::vector<int, class std::allocator<int> >::operator[]</code>	opencv_world420d.dll	562.238s
<code>cv::BRISK_Impl::generateKernel</code>	opencv_world420d.dll	90.174s
<code>sin</code>	ucrtbased.dll	53.807s
<code>cos</code>	ucrtbased.dll	40.818s
[Others]	N/A*	271.030s

*N/A is applied to non-summable metrics.

Figure 5.4.9: BRISK Detection Top Hotspots

Top Tasks. This paragraph will present the results for the profiling and benchmarking using MSVC, in terms of the most recurrent tasks within BRISK.

🕒 Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Ⓢ Time	Task Ⓢ Count	Average Ⓢ Task Time
<code>void __cdecl cv::DescriptorMatcher::match(const class cv::debug_build_guard::_InputArray &const class cv::debug_build_guard::_InputArray &class std::vector<class cv::DMatch, class std::allocator<class cv::DMatch> > & const class cv::debug_build_guard::_InputArray &const class cv::debug_build_guard::_InputArray &const class cv::DMatch &)</code>	166.916s	512	0.326s
<code>void __cdecl cv::circle(const class cv::debug_build_guard::_InputOutputArray &class cv::Point<__int>, __int, const class cv::Scalar<__double> & __int, __int, __int)</code>	57.840s	190,464	0.000s
<code>void __cdecl cv::line(const class cv::debug_build_guard::_InputOutputArray &class cv::Point<__int>, class cv::Point<__int>, const class cv::Scalar<__double> & __int, __int, __int)</code>	19.796s	97,280	0.000s
<code>void __cdecl cv::AgastFeatureDetector::Impl::detect(const class cv::debug_build_guard::_InputArray &vector<class cv::KeyPoint, class std::allocator<class cv::KeyPoint> & const class cv::debug_build_guard::_InputArray &)</code>	6.439s	6,144	0.001s
<code>void __cdecl cv::resize(const class cv::debug_build_guard::_InputArray &const class cv::debug_build_guard::_OutputArray &class cv::Size<__int>, __double, __double, __double)</code>	4.657s	5,120	0.001s
[Others]	7.573s	34,820	0.000s

*N/A is applied to non-summable metrics.

Figure 5.4.10: BRISK Detection Top Tasks

ICC Results

This section will present the results for the profiling and benchmarking using ICC.

Timing Analysis. This paragraph will present the results for the profiling and benchmarking using ICC, in terms of timing analysis.

Table 5.8: ICC Time Table

	10 Tasks	100 Tasks	1000 Tasks	10000 Tasks
\bar{x} (ms)	49.50668	45.49597	49.5654153	49.32315569
s (ms)	4.63055986	5.916665812	8.90300453	10.68847064

Elapsed Time	60.986	600.719	1790.692
CPU Time	54.290	564.041	1721.264
Effective Time	54.076	561.654	1713.728
Idle	0.955	4.608	11.554
Poor	53.121	557.046	1702.174
Ok	0.000	0.000	0.000
Ideal	0.000	0.000	0.000
Over	0.000	0.000	0.000
Spin Time	0.214	2.387	7.536
Overhead Time	0.000	0.000	0.000
Thread Count	1.000	1.000	1.000
Paused Time	0.000	0.000	0.000

everything expressed in seconds.

Figure 5.4.11: BRISK Detection Time Analysis

Top Hotspots. This paragraph will present the results the for the profiling and benchmarking using ICC, in terms of the hotspots/bottlenecks found.

🕒 Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time ⓘ
<code>std::vector<float, class std::allocator<float> >::operator[]</code>	opencv_world420d.dll	697.464s
<code>std::vector<int, class std::allocator<int> >::operator[]</code>	opencv_world420d.dll	564.693s
<code>cv::BRISK_Impl::generateKernel</code>	opencv_world420d.dll	91.304s
<code>sin</code>	ucrtbased.dll	52.730s
<code>cos</code>	ucrtbased.dll	41.210s
[Others]	N/A*	273.863s

*N/A is applied to non-summable metrics

Figure 5.4.12: BRISK Detection Top Hotspots

Top Tasks. This paragraph will present the results the for the profiling and benchmarking using ICC, in terms of the most recurrent tasks within BRISK.

🕒 Top Tasks

This section lists the most active tasks in your application.

Task Type	Task Time ⓘ	Task Count ⓘ	Average Task Time ⓘ
<code>void __cdecl cv::DescriptorMatcher::match(const class cv::debug_build_guard_<InputArray &, const class cv::debug_build_guard_<InputArray &, class std::vector<class cv::DMatch, class std::allocator<class cv::DMatch> > &, const class cv::debug_build_guard_<InputArray &)</code>	166.29s	548	0.303s
<code>void __cdecl cv::circle(const class cv::debug_build_guard_<InputOutputArray &, class cv::Point_<int>, int, const class cv::Scalar_<double>, &int, int, int)</code>	59.276s	203.856	0.000s
<code>void __cdecl cv::line(const class cv::debug_build_guard_<InputOutputArray &, class cv::Point_<int>, const class cv::Point_<int>, const class cv::Scalar_<double>, &int, int, int)</code>	20.731s	104.120	0.000s
<code>void __cdecl cv::AgastFeatureDetector::Impl::detect(const class cv::debug_build_guard_<InputArray &, class std::vector<class cv::KeyPoint, class std::allocator<class cv::KeyPoint> &, const class cv::debug_build_guard_<InputArray &)</code>	6.430s	6,576	0.001s
<code>class cv::Mat __cdecl cv::imread(const class std::basic_string<char>, struct std::char_traits<char>, class std::allocator<char> > &, int)</code>	5.059s	1,098	0.000s
[Others]	7.684s	41,650	0.000s

*N/A is applied to non-summable metrics

Figure 5.4.13: BRISK Detection Top Tasks

5.4.3 OProfile Profiling

First, to start profiling it was used the `opperf` command, followed by the command `opreport` to show all the information retrieved from the code. To be sure of the obtained results, this process was carried out but now with a larger number of iterations. To do this, a bash file was used, as it can be seen below (Source: Vitor Silva):

```

#!/bin/bash
#!/usr/bin/env python3
3  if [ $# -eq 3 ]; then

    sudo rm -rf oprofile_data/
    echo -----
    echo "Profiling $1 for $2 iteration(s)"
8     echo -----
     for i in $(seq 1 $2)
     do
        echo -----
        echo iteration $i
13       echo -----
        sudo opperf --append --event CPU_CLK_UNHALTED:$3:0:0:1 ./$1
        echo -----
     done
     opreport --accumulated\
18       --exclude-dependent\
       --exclude-symbols=_GLOBAL_OFFSET_TABLE_ \
       --symbols > iterationGlobal.perf
else
    echo "Wrong command <executable> <n_iter> <sample_rate>"
23 fi

```

Now, with the code and bash script implemented, one has everything ready to start testing. For a matter of coherency, every test will include a number of executions of the code in question.

5.4.4 ORB Detection

To have a good testing phase it was decided to do the profile with three different photos, with different keypoints and image sizes.



Figure 5.4.14: Three images used to profile the ORB Detection phase

Test 1: Eagle Image. The test for the image of the Eagle was carried out. The results obtained are in the fig. 5.4.15.

```
carlos@carlos-VivoBook-ASUSlaptop-X571LI-F571LT:~/Desktop/opencv$ sudo operf --append --event CPU_CLK_UNHALTED:100000:0:0:1 ./opencv
opref: Profiler started
1317
Profiling done.
carlos@carlos-VivoBook-ASUSlaptop-X571LI-F571LT:~/Desktop/opencv$ oreport
Using /home/carlos/Desktop/opencv/oprofile_data/samples/ for samples directory.
CPU: Intel Architectural Perfmon, speed 5000 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
571 81.4551 opencv
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
341 59.7198 libjpeg.so.8.2.2
/1 12.4343 ld-2.31.so
/2 9.1668 libnvenc.so.4.5.2
40 7.0053 libopencv_features2d.so.4.5.2
27 6.9996 libopencv_imgproc.so.4.5.2
17 2.9772 libnvidia-opencl.so.460.80
10 1.7513 kallsyms
5 0.8757 libopencv_core.so.4.5.2
3 0.5254 libstdc++.so.6.0.28
2 0.3503 libpthread-2.31.so
2 0.1751 libopencv_imgcodecs.so.4.5.2
1 0.1751 libm-2.31.so
130 18.5449 modprobe
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
57 43.8462 ld-2.31.so
32 25.3846 libc-2.31.so
26 28.0000 kmalloc
18 1.6923 kallsyms
2 1.5385 libcrypto.so.1.1
1 0.7692 liblzma.so.5.2.4
1 0.7692 libpthread-2.31.so
```

Figure 5.4.15: Results of oProfile in the Eagle image.

This image is the largest image with 205Kb, and at the same time is the image with less keypoints found (1317), marked in red. Marked in green is the samples in the JPEG library (341), that is the library used to read the image, it is a large number because the image size is also large. Marked in purple it is the library features2d that is the library where the ORB is included, and for 1317 keypoints found it as 48 samples.

Test 2: Castle Image. The test for the image of the castle was carried out. The results obtained are in the fig. 5.4.16.

```
1809
Profiling done.
carlos@carlos-VivoBook-ASUSlaptop-X571LI-F571LT:~/Desktop/opencv$ oreport
Using /home/carlos/Desktop/opencv/oprofile_data/samples/ for samples directory.
CPU: Intel Architectural Perfmon, speed 5000 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 100000
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
341 72.2458 opencv
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
124 36.3636 ld-2.31.so
54 15.8358 libjpeg.so.8.2.2
53 15.5425 libopencv_features2d.so.4.5.2
48 14.0762 libc-2.31.so
19 5.5718 libopencv_imgproc.so.4.5.2
17 4.9853 libnvidia-opencl.so.460.80
10 2.9326 kallsyms
7 2.0526 libopencv_core.so.4.5.2
2 0.5865 libglbl-2.0.so.0.6400.6
2 0.5861 libopencv_imgcodecs.so.4.5.2
1 0.2933 libglm-2.3.so.24.0.0
1 0.2933 libgobject-2.0.so.0.6400.6
1 0.2933 libpango-1.0.so.0.4400.7
1 0.2933 libpthread-2.31.so
1 0.2933 libstdc++.so.6.0.28
131 27.7542 modprobe
CPU_CLK_UNHALT...| samples| %|
-----|-----|-----|
59 45.0382 ld-2.31.so
27 28.6107 kmalloc
27 28.6107 libc-2.31.so
15 11.4564 kallsyms
```

Figure 5.4.16: Results of oProfile in the Castle image.

This image was the smallest one with 105Kb so as expected the samples in the JPEG library dropped to only 54 samples, marked in green. In this image was found 1809 keypoints, marked in red, so in this test the samples in the features2d went to 53, marked in pink.

Test 3: Lion Image. The test for the image of the lion was carried out. The results obtained are in the fig. 5.4.17.

```
carlos@carlos-VivoBook-ASUSLaptop-X571LI-F571LI:~/Desktop/opencv$ sudo operv --append --event CPU_CLK_UNHALTED:00000:0:0:1 ./opencv
operv: Profiler started
2473
Profiling done.
carlos@carlos-VivoBook-ASUSLaptop-X571LI-F571LI:~/Desktop/opencv$ oreport
using /home/carlos/Desktop/opencv/profile_data/samples/ for samples directory.
CPU: Intel Architectural Perfmon, speed 5000 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) co
unt 100000
CPU_CLK_UNHALT...|
samples| %|
-----
643 83.1824 opencv
CPU_CLK_UNHALT...|
samples| %|
-----
278 43.2348 libjpeg.so.8.2.2
183 28.4603 ld-2.31.so
60 9.3313 libopencv_features2d.so.4.5.2
48 7.4650 libc-2.31.so
18 2.7994 libopencv_imgproc.so.4.5.2
17 2.6439 libnvidia-opencl.so.460.80
14 2.1773 libopencv_core.so.4.5.2
12 1.8663 kallsyms
2 0.3110 libgobject-2.0.so.0.6400.6
2 0.3110 libpthread-2.31.so
2 0.3110 libopencv_imgcodecs.so.4.5.2
1 0.1555 opencv
1 0.1555 libILmImf-2.3.so.24.0.0
1 0.1555 libOpenCL.so.1.0.0
1 0.1555 libblkid.so.1.1.0
1 0.1555 libglib-2.0.so.0.6400.6
1 0.1555 libstdc++.so.6.0.28
1 0.1555 libxcb-shm.so.0.0.0
130 16.8176 modprobe
CPU_CLK_UNHALT...|
samples| %|
-----
61 46.9231 ld-2.31.so
30 23.0769 libc-2.31.so
28 21.5385 kmmod
9 6.9231 kallsyms
1 0.7692 libdl-2.31.so
1 0.7692 libpthread-2.31.so
```

Figure 5.4.17: Results of oProfile in the Lion image.

This image has a size of 150Kb and it obtained 278 samples in the JPEG library. This images was the highest number of keypoints (2473) found so it was the highest number of samples in the features2d library (60). To have a better understanding of the results the table 5.9 was created.

Table 5.9: oProfile results - ORB Detection Phase

	Keypoints	Image Size	Samples lib JPEG	Samples lib features2D
Eagle Image	1317	205 Kb	341	40
Castle Image	1809	105 Kb	54	53
Lion Image	2473	150 Kb	278	60

Test 4: 500 iterations. The percentage of samples performed by the jpeg and features2d libraries were practically the same. It can be concluded then that when performing only one iteration, the results did not mislead us.

5.4.5 ORB Descriptor

Test 1. First one executed the code complete, and by the report one concluded that the function cv::LineAA was a hotspot, and that function is responsible to draw the lines in the pictures after the matching. So, one removed all the image showing and the matching from the code. The matching was also taking a big part of the execution, one can see in the picture in the functions "normHamming". The time execution media of this test was 147,7 ms.

```
CPU: Intel Sandy Bridge microarchitecture, speed 3500 MHz (estimated)
Counted CPU_CLK_UNHALTED events (Clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 10000
samples % image name symbol name
8290 9.1734 libopencv_core.so.4.5.3 cv::WorkerThread::thread_body()
6897 7.6320 libopencv_imgproc.so.4.5.3 void hlineResizeCn<unsigned char, ufixedpoint16, 2, true, 1>(unsigned char*, int, int)
5656 6.2587 libglib-2.0.so.0.5600.4 /usr/lib/x86_64-linux-gnu/libglib-2.0.so.0.5600.4
5328 5.8958 libopencv_imgproc.so.4.5.3 cv::LineA(cv::Mat8, cv::Point<long>, cv::Point<long>, void const*)
5053 5.5915 libz.so.1.2.11 /lib/x86_64-linux-gnu/libz.so.1.2.11
4317 4.7770 libopencv_imgproc.so.4.5.3 cv::hal::opt_SSE4_1::cvtColorLoop_Invoker<cv::hal::opt_SSE4_1::RGB2Gray<cv::Mat8>, cv::Mat8>
4316 4.7759 ld-2.27.so do_lookup_x
3834 4.2426 libgtk-3.so.0.2200.30 /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2200.30
3182 3.5211 libopencv_imgproc.so.4.5.3 cv::opt_SSE4_1::RowFilter<unsigned char, float, cv::opt_SSE4_1::RowVec<cv::Mat8>, int, int>
2930 3.2422 libopencv_imgproc.so.4.5.3 icv_y8_ownSwapChannels_8u_C3R
2815 3.1150 libgobjection-2.0.so.0.5600.4 /usr/lib/x86_64-linux-gnu/libgobjection-2.0.so.0.5600.4
2579 2.8538 libopencv_imgproc.so.4.5.3 cv::opt_SSE4_1::SymmColumnFilter<cv::opt_SSE4_1::Cast<float, unsigned char const**, unsigned char*, int, int>
2525 2.7941 libopencv_features2d.so.4.5.3 void cv::FAST_t<16>(cv::_InputArray const, vector<cv::KeyPoint> &, int)
2096 2.3194 libopencv_imgproc.so.4.5.3 resize<bitExactInvoker<unsigned char, ufixedpoint16, 2>::operator()>(cv::UMat, cv::UMat)
1756 1.9431 libx11.so.6.3.0 /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0
1552 1.7174 ld-2.27.so strcmp
1538 1.7019 libopencv_core.so.4.5.3 cv::hal::normHamming(unsigned char const*, unsigned char const*, int)
1532 1.6953 libopencv_core.so.4.5.3 cv::hal::opt_SSE4_2::normHamming(unsigned char const*, unsigned char const*, int)
1065 1.4708 libcurl-4.14.1.so.0.2200.30 /usr/lib/x86_64-linux-gnu/libcurl-4.14.1.so.0.2200.30
```

Figure 5.4.18: Results: Test 1

Test 1

Number of Executions	Mean (ms)
10	147.7

Table 5.10: Results: Test 1 Timing

Test 2. In test 2 one can verify that the functions mentioned in Test 1 disappeared and the time of execution dropped significantly, but that was already expected, because, as one saw in Vtune profiling, the matching takes a big part on the execution greatly because of the absence of a non-max suppression filter. In this report one can see that the main hotspot is the function hlineResize, where the program makes the successive resizes of the picture in the detection stage (Oriented Fast), but surprisingly, in fourth place is the image colour conversions to detect and describe keypoints. So, in the next test one will read the images on a greyscale. The time execution media of this test was 63,7 ms.

Figure 5.4.19: Results: Test 2

Test 2

Number of Executions	Mean (ms)
10	63.7

Table 5.11: Results: Test 2 Timing

Test 3. As one can see, the convert colour function is no longer an hotspot, and the time of execution improved, but now one must pay attention to the row filter and column filter functions, used by the descriptor. In a chance to improve the program one will replace the orb descriptor for a pure brief descriptor. The time execution media of this test was 51,4 ms.

```
CPU: Intel Sandy Bridge microarchitecture, speed 3500 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 10000
samples % image name symbol name
8066 25.1066 libopencv_imgproc.so.4.5.3 void hlineResizeCn<unsigned char, ufixedpoint16, 2, true, 1>(unsigned
, int, int)
5390 16.7772 libopencv_core.so.4.5.3 cv::iWorkerThread::thread_body()
2842 8.8461 libopencv_imgproc.so.4.5.3 cv::iOpt_SSE4_1::RowFilter<unsigned char, float, cv::iOpt_SSE4_1::RowNoV
int, int>
2468 7.6820 libopencv_imgproc.so.4.5.3 resize_bitExactInvoker<unsigned char, ufixedpoint16, 2>::operator()(cv
2294 7.1404 libopencv_imgproc.so.4.5.3 cv::iOpt_SSE4_1::SymmColumnFilter<cv::iOpt_SSE4_1::Cast<float, unsigned
ned char const**, unsigned char*, int, int>
2107 6.5583 libopencv_features2d.so.4.5.3 void cv::FAST_t<16>(cv::_InputArray const&, vector<cv::KeyPoint> &,
2015 6.2720 libjpeg.so.8.1.2 /usr/lib/x86_64-linux-gnu/libjpeg.so.8.1.2
785 2.4434 libopencv_features2d.so.4.5.3 cv::ORB_Impl::detectAndCompute(cv::_InputArray const&, cv::_InputAr
onst&, bool)
648 2.0170 ld-2.27.so do_lookup_x
491 1.5283 ldbc-2.27.so sched_yield
424 1.3198 libopencv_core.so.4.5.3 cv::copyMakeBorder(cv::_InputArray const&, cv::_OutputArray const&, int,
421 1.3104 ld-2.27.so _dl_relocate_object
390 1.2139 ld-2.27.so _dl_lookup_symbol_x
361 1.1237 libopencv_features2d.so.4.5.3 cv::computeKeyPoints(cv::Mat const&, cv::UMat const&, cv::Mat const
ctor<float> const&, vector<cv::KeyPoint> &, int, double, int, int, cv::ORB::ScoreType, bool, int)
280 0.8715 libopencv_core.so.4.5.3 cv::softfdouble::operator*(cv::softdouble const&) const
258 0.8031 libopencv_core.so.4.5.3 cv::softfdouble::operator-(cv::softdouble const&) const
```

Figure 5.4.20: Results: Test 3

Test 3

Number of Executions	Mean (ms)
10	51.4

Table 5.12: Results: Test 3 Timing

Test 4. In test 4 the time improved again, and if one notice, the function compute from the pure brief collects 500 samples versus the 785 from the orb brief. The time execution media of this test was 32,7 ms.

```
CPU: Intel Sandy Bridge microarchitecture, speed 3500 MHz (estimated)
Counted CPU_CLK_UNHALTED events (clock cycles when not halted) with a unit mask of 0x00 (No unit mask) count 10000
samples % image name symbol name
4189 21.0608 libopencv_imgproc.so.4.5.3 void hlineResizeCn<unsigned char, ufixedpoint16, 2, true, 1>(unsigned
, int, int)
2707 13.6099 libopencv_core.so.4.5.3 cv::iWorkerThread::thread_body()
243 11.9781 libopencv_features2d.so.4.5.3 void cv::FAST_t<16>(cv::_InputArray const&, vector<cv::KeyPoint> &,
114 10.7783 libjpeg.so.8.1.2 /usr/lib/x86_64-linux-gnu/libjpeg.so.8.1.2
1281 6.4404 libopencv_imgproc.so.4.5.3 resize_bitExactInvoker<unsigned char, ufixedpoint16, 2>::operator()(cv
907 4.5601 libopencv_xfeatures2d.so.4.5.3 cv::xFeatures2d::smoothedSum(cv::Mat const&, cv::KeyPoint const&, int
862 4.3338 ld-2.27.so do_lookup_x
500 2.5138 libopencv_features2d.so.4.5.3 cv::computeKeyPoints(cv::Mat const&, cv::UMat const&, cv::Mat const&
ctor<float> const&, vector<cv::KeyPoint> &, int, double, int, int, cv::ORB::ScoreType, bool, int)
448 2.0170 ld-2.27.so do_lookup_x
408 1.9597 ld-2.27.so _dl_relocate_object
324 1.6290 libopencv_core.so.4.5.3 cv::copyMakeBorder(cv::_InputArray const&, cv::_OutputArray const&, int, int
322 1.6189 libopencv_imgproc.so.4.5.3 cv::initIntterTab2D(int, bool)
312 1.5686 ld-2.27.so strcmp
288 1.4077 libopencv_imgproc.so.4.5.3 lcv_y8_ownIntegral_8u32s_CIR_U8
234 1.1765 ldbc-2.27.so sched_yield
114 1.0686 libopencv_core.so.4.5.3 cv::softfdouble::operator*(cv::softdouble const&) const
103 0.9783 libopencv_core.so.4.5.3 cv::softfdouble::operator*(cv::softdouble const&) const
155 0.7793 libopencv_xfeatures2d.so.4.5.3 cv::xFeatures2d::pixelTests32(cv::_InputArray const&, vector<cv::Key
136 0.6838 lib-2.27.so _memcpy_sse2_unaligned_erns
127 0.6385 libopencv_core.so.4.5.3 cv::softfdouble::softfdouble(int)
```

Figure 5.4.21: Results: Test 4

Test 4

Number of Executions	Mean (ms)
10	32.7

Table 5.13: Results: Test 4 Timing

5.4.6 BRIEF Refactored

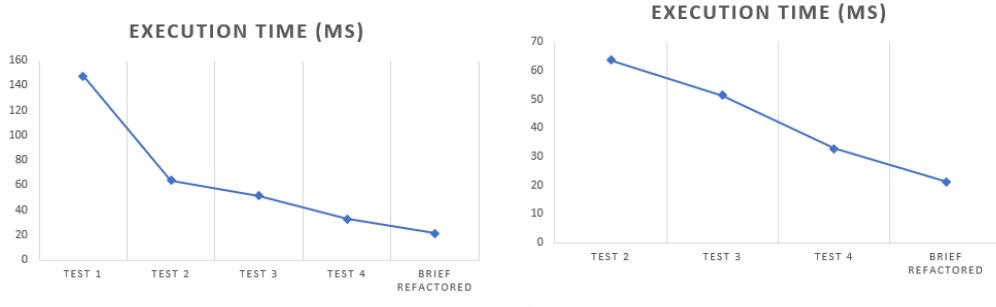
To profile the BRIEF refactored code was used the same process used to profile the opencv program with OProfile. Since the I/O operations were not very useful the first test was done already without this functions. The following picture shows the result of this test.

Figure 5.4.22: Profile Refactored

Test 5

Number of Executions	Mean (ms)
10	21.3

Table 5.14: Results: Test 5 Timing



(a) Execution time comparison - Test 1 with matching

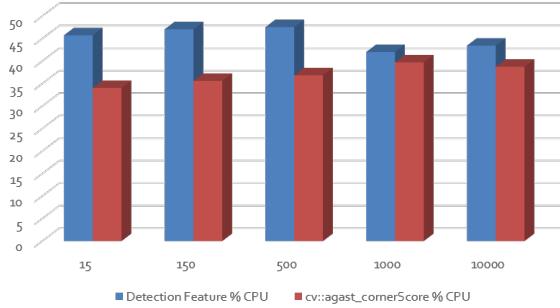
(b) Execution time comparison without matching -
only detection and description

5.4.7 BRISK Detection

Table 5.15: OProfile Time Table

	10 Tasks	100 Tasks	1000 Tasks	10000 Tasks
\bar{x} (ms)	6.912053	6.7391235	6.84362548	6.925233395
s (ms)	0.45519986	0.312206301	0.25002655	0.414904837

Detection Feature % CPU vs. Most % CPU Consuming Task



(a) Results: Consumption of CPU vs Most Consuming Task

Nº of Iterations	15	150	500	1000	10000
Elapsed Time	2.411	23.46	78.853	167.039	1501.91
Samples	43	448	1539	2411	28506
% CPU Utilization	45.7446	47.2076	47.8514	48.1862	47.4815

(b) Results: CPU vs Time

5.5 BRISK MSVC vs. ICC vs. OProfile

Table 5.16: MSVC vs. ICC vs. OProfile

\bar{x} (ms)	10 Tasks			100 Tasks			1000 Tasks			10000 Tasks		
	Vtune - ICC	Vtune - MSVC	Oprofile	Vtune - ICC	Vtune - MSVC	Oprofile	Vtune - ICC	Vtune - MSVC	Oprofile	Vtune - ICC	Vtune - MSVC	Oprofile
\bar{x} (ms)	49.50668	43.82994	6.912053	45.49597	48.87546	6.739124	49.56542	54.55777	6.843625	49.32316	52.14287	6.925233
s (ms)	4.63055986	4.982791225	0.45519986	5.916666	5.920759	0.312206	8.903005	17.10567	0.250027	10.68847	13.63817	0.414905

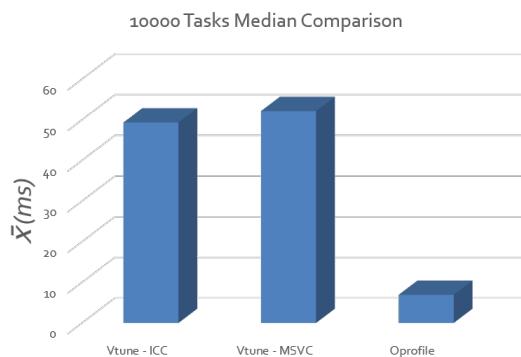


Figure 5.5.1: MSVC vs. ICC vs. OProfile Graph

5.6 Verification

5.6.1 DSL Unit and Integration Tests

The unit and integration tests for the DSL detection are presented in table 5.17 and table 5.18.

Unit Tests	Expected Results	Real Results
Test all instructions individually	All instructions function correctly	All instructions function correctly
Use DSL code to check for lexical errors	All lexical errors were detected	All lexical errors were detected
Use DSL code to check syntactic errors	All syntactic errors were detected	All syntactic errors were detected
Use DSL code to check semantic errors	All semantic errors were detected	All semantic errors were detected
Use DSL code to check the code generator	The C++ and verilog code was generated	The C++ and verilog code was generated correctly

Table 5.17: DSL Unit Tests

Integrated Tests	Expected Results	Real Results
Use configured system written in DSL to generate code	The code was generated correctly	The code was generated correctly as expected

Table 5.18: DSL Integrated Tests

5.6.2 ORB Detection Unit and Integration Tests

The unit and integration tests for ORB are presented in table 5.19.

Unit and Integrated Tests	Expected Result	Real Result
Verify the patch generation	Correctly generate the patch	As expected
Invalid data signalling	Correctly identify edge cases in patcher	As expected
Verify dark/bright classifier stage	The potential keypoints were classified as expected	As expected
Verify contiguity test and scoring stage	The keypoints were all detected	As expected
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	As expected
Verify lists organisation	The lists were organised as expected	As expected
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	As expected

Table 5.19: ORB Unit and Integration Tests

5.6.3 ORB- Software Refactoring

Unit and Integrated Tests	Expected Results	Real Results
Verify detection stage	The potential keypoints were detected as expected	As expected
Verify contiguity test and scoring stage	The keypoints were all detected	As expected
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	As expected
Verify lists organisation	The lists were organised as expected	As expected
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	As expected
Compare execution times	Lower execution time	As expected

Table 5.20: ORB - Software Implementation

5.6.4 BRISK Detection Unit and Integration Tests

The unit and integration tests for BRISK detection are presented in table 5.21.

Unit Test	Expected Result	Real Result
Request image fetch	Image is fetched and written to the internal buffer	It's possible to store an image on slice buffer
Request detection	Pixels correctly identified as bright or dark	Pixels correctly identified as bright or dark
Program image location and size	Configure memory access controller	Not tested
Measure the detector's throughput and latency with Vivado	Low latency and high throughput	Low latency (12 cycles) and high throughput (1 cycle per pixel)
Compare BRISK detection in Sw and Hw	Less time on the Hw	Only achieved the Hw time
Integrated Test	Expected Result	Real Result
Trigger detection	Pixel classification and other scoring and NMS inputs	Pixel classification and other scoring and NMS inputs

Table 5.21: Unit and Integrated Tests

5.6.5 BRISK Scoring and Contiguity Test Unit and Integration Tests

The unit and integration tests for BRISK scoring are presented in table table 5.22.

5.6.6 BRISK NMS Unit and Integration Tests

In the table 5.23 are the NMS tests that will be needed to verify if the algorithm is working as expected.

Unit Test	Expected Result	Real Result
Input a contiguous state	Valid keypoint	As expected
Input a non contiguous state	Not a keypoint	As expected
Input bright/dark pixels	Get score value	As expected
Compare pixels scores	Get maximum score value	As expected
Test throughput and latency	Less latency on the Hw	High throughput(1 cycle per pixel) and low latency(2 cycles)
Integrated Test	Expected Result	Real Result
Combining Detection and Scoring Stages	Get Score and Contiguity Test at the same time	As expected

Table 5.22: BRISK Scoring and Contiguity Test Unit and Integration Tests

Unit and Integrated Tests	Expected Result	Real Result
Verify detection stage	The potential keypoints were detected as expected	As expected
Verify contiguity test and scoring stage	The keypoints were all detected	As expected
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	As expected
Verify lists organisation	The lists were organised as expected	As expected
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	As expected
Compare execution times	Lower execution time	As expected

Table 5.23: BRISK NMS Unit and Integration Tests

5.6.7 BRISK Software Refactoring

In the table 5.24 are the results of all the tests proposed. All the tests were successful.

Unit and Integrated Tests	Expected Results	Real Results
Verify detection stage	The potential keypoints were detected as expected	The potential keypoints were detected as expected
Verify contiguity test and scoring stage	The keypoints were all detected	The keypoints were all detected
Verify horizontal NMS stage	The keypoints with horizontal neighbours that have a bigger score were suppressed	The keypoints with horizontal neighbours that have a bigger score were suppressed
Verify lists organisation	The lists were organised as expected	The lists were organised as expected
Verify vertical NMS stage	The keypoints with vertical neighbours that have a bigger score were suppressed	The keypoints with vertical neighbours that have a bigger score were suppressed
Compare execution times	Lower execution time	Almost half the execution time

Table 5.24: BRISK Unit and Integration Test Cases for Software Implementation

5.6.8 BRIEF

The unit and integration tests for BRIEF are presented in section 5.6.8 and table 5.26.

Unit and Integrated Tests	Expected Results	Real Results
Keypoint coordinates reception	Receive keypoint coordinates from detection stage	As expected
Pair generation	The patch-pairs are successfully generated	As expected
Descriptor computation	The binary string for the descriptor is generated from the binary tests	As expected
Verify the space allocation	The correct space was allocated	As expected
Verify the patch	The correct area was selected with the keypoint at the centre	As expected
Verify the pattern generation	The pattern was generated has intended	As expected
Descriptor dispatch	The descriptor is sent to the matching stage	As expected
Descriptor repeatability	Returns the same vector	As expected

Table 5.25: BRIEF Unit and Integration Test Cases for Hardware Implementation

Test	Expected Result	Real Result
Verify the pattern generation	The pattern was generated has intended	As expected
Verify the Descriptor Computation	Generates the expected descriptor	As expected
Descriptor Repeatably	Returns the same string	As expected
Compare execution times	Lower execution time	Half of the time

Table 5.26: BRIEF Unit and Integration Test Cases for Software Implementation

5.6.9 Matching

The test cases for the matching can be consulted in tables 5.27 and 5.28 and almost all were successful, with the connection between PS and PL not working, due to AXI problems.

Unit and Integrated Tests	Expected Results	Real Results
Matching Core best descriptor detection	Best descriptor detection	As expected
Reading and writing in the BRAM	Read and write data in the BRAM at the correct address	As expected
Coordination between the controller and BRAM	Connection between the controller and the BRAM	As expected
Information exchange between the controller and 1 matching core	Load the descriptors to the matching Core and receive the best matches and signals in the controller	As expected
Information exchange between the controller and the 3 matching cores	Load the descriptors to the 3 matching Core and receive the best matches and signals in the controller	As expected
Cross checking	False Matching signalled	As expected
Applying more than one matching Core	Reduction of time in the matching process	As expected
AXI Connection between the PS and the PL	Exchange information between the PS and the PL	Not as expected

Table 5.27: Matching Unit and Integration Test Cases for Hardware Implementation

Unit and Integrated Tests	Expected Results	Real Results
Load linked list descriptors	Correct reading of all descriptors	As expected
Slope based rejection	Remove false matches	As expected
Comparison of the results obtained with the implemented algorithm and the OpenCV	High accuracy	As expected
Brute force	Find the best matches	As expected
Cross Checking	False Matching removed	As expected

Table 5.28: Matching Unit and Integration Test Cases for Software Implementation

6 | Conclusion

At the beginning of the project two goals were set. One, developing a system capable of matching two images on the ZYBO board, and two, developing an AXI interface between it and the Processing System, and in parallel refactor a software counterpart for the sake of comparison. While the first one was successfully met in terms of functionality in simulation, the second one was cut short.

Although one was able to understand the protocol and successfully test the technology, the lack of experience implementing this kind of interfaces had a big impact on the workflow of the interface, which explains the lack of information about it in the implementation and testing chapters. Nonetheless, as the primary goal was remarkably achieved, while gaining considerable knowledge in both aspects, the project outcome was undoubtedly positive.

Despite the setback, the groundwork necessary for the second goal was also prepared successfully, which encompassed the implementation of a register-based configuration, the creation of a Petalinux image for the board and the shared-memory allocation for the eventual interface.

However, it is also important to be aware that, achieving the second goal would allow for real-world implementation and testing, for comparison with the refactored code within the ZYBO, which would certainly reveal some design flaws and potentially contribute even more for the class' growth as engineers.

6.1 Future Works

As aforementioned, although the class is left satisfied with the state of the project at the time of delivery, there's also room for improvement and some work could certainly be done in the future.

- Implement and test the AXI-DDR interface;
- Use a different method for memory allocation;
- Properly integrate the hardware modules;
- Develop the core program that would call the hardware routines;
- Develop a device driver for the aforementioned main program.

Bibliography

- [1] Fularz, M., Kraft, M., Schmidt, A. and Kasiłski, A., 2015. A High-Performance FPGA-Based Image Feature Detector and Matcher Based on the FAST and BRIEF Algorithms. International Journal of Advanced Robotic Systems, 12(10), p.141.
- [2] Moyer, B., 2013. Real world multicore embedded systems. Oxford: Newnes.
- [3] Leutenegger, Stefan & Chli, Margarita & Siegwart, Roland. (2011). BRISK: Binary Robust invariant scalable keypoints. Proceedings of the IEEE International Conference on Computer Vision. 2548-2555. 10.1109/ICCV.2011.6126542.
- [4] Azimi, E., Behrad, A., Ghaznavi-Ghoushchi, M. and Shanbehzadeh, J., 2017. A fully pipelined and parallel hardware architecture for real-time BRISK salient point extraction. Journal of Real-Time Image Processing, 16(5), pp.1859-1879.
- [5] E. Mair, G. D. Hager, D. Burschka, M. Suppa, and G. Hirzinger. Adaptive and generic corner detection based on the accelerated segment test. In Proceedings of the European Conference on Computer Vision (ECCV), 2010.
- [6] Grandviewresearch.com. 2021. Computer Vision Market Size Share Report, 2020-2027. [online] Available at: <<https://www.grandviewresearch.com/industry-analysis/computer-vision-market>> [Accessed 11 May 2021].
- [7] Hu, Y., G. H. Geng, Q. M. Zhou, and X. F. Wang. 2010.“Stereo Matching Algorithm for 3D Reconstruction Based on Un-Calibrated Images.” Application Research of Computers 27 (10):3964–3967
- [8] Ji, T., Q. H. Sheng, and P. Zong. 2010.“Precision Analysis of Correlation Coefficient Matching for High Overlap Images Matching.” Journal of Computer Applications 30–2: 57–59.
- [9] Jingjin Huang , Guoqing Zhou, "On-Board Detection and Matching of Feature Points", Research Gate
- [10] Jingjin Huang, Guoqing Zhou, Dianjun Zhang, Guangyun Zhang, Rongting Zhang & Oktay Baysal , "An FPGA-based implementation of corner detection and matching with outlier rejection", International Journal of Remote Sensing
- [11] ORB Feature Extraction and Matching in Hardware
- [12] A High-Performance FPGA-Based Image Feature Detector and Matcher Based on the FAST and BRIEF Algorithms
- [13] Jakubovic, Amila & Velagic, Jasmin. (2018). Image Feature Matching and Object Detection Using Brute-Force Matchers. 83-86. 10.23919/ELMAR.2018.8534641.
- [14] Calonder, M., Lepetit, V., Ozuysal, M., Trzcinski, T., Strecha, C. and Fua, P., 2012. BRIEF: Computing a Local Binary Descriptor Very Fast. IEEE Transactions on Pattern Analysis and Machine Intelligence, 34(7), pp.1281-1298.
- [15] Fang, W., Zhang, Y., Yuy, B. and Liuy, S., 2018. FPGA-based ORB Feature Extraction for Real-Time Visual SLAM. Institute of Application Specific Instruction Set Processor.
- [16] Naia, N., 2021. Real-Time Linux and Hardware Accelerated Systems on QEMU.
- [17] Mesquita, J., 2021. Hybrid Linux System Modelling with Mixed-Level Simulation.
- [18] Maxfield, C. FPGAs: World Class Designs. Newnes, 2009, vol. 1.
- [19] Dekker, R., 2021. What's the Difference Between VHDL, Verilog, and SystemVerilog?. [online] Electronicdesign.com. Available at: <<https://www.electronicdesign.com/resources/whats-the-difference-between/article/21800239/whats-the-difference-between-vhdl-verilog-and-systemverilog>> [Accessed 4 June 2021].

- [20] Xilinx. 2021. What is an FPGA? Field Programmable Gate Array. [online] Available at: <<https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>> [Accessed 4 June 2021].
- [21] Forums.xilinx.com. 2021. [online] Available at: <<https://forums.xilinx.com/t5/Design-and-Debug-Techniques-Blog/AXI-Basics-1-Introduction-to-AXI/ba-p/1053914>> [Accessed 4 June 2021].
- [22] Silva, V., 2021. AXI Interface Slides.
- [23] Developer.arm.com. 2021. Documentation – Arm Developer. [online] Available at: <<https://developer.arm.com/documentation/ihio051/a/Interface-Signals/Transfer-signaling/Handshake-process?lang=en>> [Accessed 4 June 2021].
- [24] Xilinx.com. 2021. [online] Available at: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug1118-vivado-creating-packaging-custom-ip.pdf> [Accessed 12 June 2021].
- [25] Bettini, L., 2021. Implementing Domain-Specific Languages with Xtext and Xtend.
- [26] Salgado, F., Martins, A., Almeida, D., Gomes, T., Monteiro, J. and Tavares, A., 2021. MODELADBT: Model-Driven Elaboration Language Applied to Dynamic Binary Translation.

Appendices

A | Testbenches

Listing A.1: Detector Testbench

```
'timescale 1ns / 1ps
2
module detector_tb_general ();
7
    reg      global_hold;
    reg      global_clk;
7
    reg      global_reset;
7
    reg      [7:0]   threshold;
12
    reg      [$clog2(/*'RAM_DEPTH*/8)-1:0]           write_en;
    reg      [$clog2(/*'ADDRESS_WRITE_BLOCKS*/8)-1:0]   line_addr_write;
    reg      [/*'ADDRESS_WRITE_BLOCK_SIZE*/4*8-1:0]       block_addr_write;
    reg      [/*'ADDRESS_WRITE_BLOCK_SIZE*/4*8-1:0]       data_write;
17
    fast
        #(.FAST_VARIANT(5))
        fast5( .global_clk_en(~global_hold), .global_clk(global_clk), .global_reset(global_reset),
                .threshold(threshold), .write_en(write_en), .line_addr_write(line_addr_write),
                .block_addr_write(block_addr_write), .data_write(data_write));
22
    fast
        #(.FAST_VARIANT(9))
        fast9( .global_clk_en(~global_hold), .global_clk(global_clk), .global_reset(global_reset),
                .threshold(threshold), .write_en(write_en), .line_addr_write(line_addr_write),
                .block_addr_write(block_addr_write), .data_write(data_write));
27
    fast
        #(.FAST_VARIANT(12))
        fast12( .global_clk_en(~global_hold), .global_clk(global_clk), .global_reset(global_reset),
                .threshold(threshold), .write_en(write_en), .line_addr_write(line_addr_write),
                .block_addr_write(block_addr_write), .data_write(data_write));
32
37
    /* Reset */
    initial begin
        global_reset = 0;
        #10
        global_reset = 1;
        #12
        global_reset = 0;
42
        end
47
    /* Global clocking */
    initial begin
        #10
        forever
            #5 global_clk = ~global_clk;
        end
52
    /* Prepare memory */
    initial begin
57
        /* Reset clocks and change input data */
        threshold <= 8'h10;
        // clk <= 0;          /* doesn't matter, will be left off */
        global_clk <= 0;
```

```

62      write_en <= 1;
      global_hold <= 0;
#18
      global_hold <= 1;
#7

67      /* Write 1st line */
      data_write <= 128'h8a8a8a5b;      /* 3->0 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd0;         /* Block 0 */
#10;    /* Clock in */
      data_write <= 128'h8f69af84;      /* 7->4 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd1;         /* Block 1 */
#10;    /* Clock in */
      data_write <= 128'h8a8a2baf;      /* 11->8 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd2;         /* Block 2 */
#10;    /* Clock in */
      data_write <= 128'h0e2b8a89;      /* 15->12 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd3;         /* Block 3 */
#10;    /* Clock in */
      data_write <= 128'h8a2b0e0e;      /* 19->16 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd4;         /* Block 4 */
#10;    /* Clock in */
      data_write <= 128'h8b898a8a;      /* 23->20 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd5;         /* Block 5 */
#10;    /* Clock in */
      data_write <= 128'hoeca1332;      /* 27->24 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd6;         /* Block 6 */
#10;    /* Clock in */
      data_write <= 128'hafafafae0f;     /* 31->28 */
      line_addr_write <= 3'd0;          /* Line 0 */
      block_addr_write <= 3'd7;         /* Block 7 */
#10;    /* Clock in */

      /* Write 2nd line */
      data_write <= 128'h8a5b8b5b;      /* 3->0 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd0;         /* Block 0 */
#10;    /* Clock in */
      data_write <= 128'h8eafaf85;      /* 7->4 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd1;         /* Block 1 */
#10;    /* Clock in */
      data_write <= 128'h898a2c69;      /* 11->8 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd2;         /* Block 2 */
#10;    /* Clock in */
      data_write <= 128'hafzb8a8b;      /* 15->12 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd3;         /* Block 3 */
#10;    /* Clock in */
      data_write <= 128'h2badaea;      /* 19->16 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd4;         /* Block 4 */
#10;    /* Clock in */
      data_write <= 128'h895c8a2b;      /* 23->20 */
      line_addr_write <= 3'd1;          /* Line 1 */
      block_addr_write <= 3'd5;         /* Block 5 */

```

```

127      #10; /* Clock in */
      data_write <= 128'hafa2d132; /* 27->24 */
      line_addr_write <= 3'd1; /* Line 1 */
      block_addr_write <= 3'd6; /* Block 6 */
      #10; /* Clock in */
      data_write <= 128'hafaf34af; /* 31->28 */
      line_addr_write <= 3'd1; /* Line 1 */
      block_addr_write <= 3'd7; /* Block 7 */
      #10; /* Clock in */

      /* Write 3rd line */
      data_write <= 128'h8a5b8a5b; /* 3->0 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd0; /* Block 0 */
      #10; /* Clock in */
      data_write <= 128'h0e0e0e0e2a; /* 7->4 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd1; /* Block 1 */
      #10; /* Clock in */
      data_write <= 128'h090a5a3f; /* 11->8 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd2; /* Block 2 */
      #10; /* Clock in */
      data_write <= 128'ha34b2323; /* 15->12 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd3; /* Block 3 */
      #10; /* Clock in */
      data_write <= 128'hb08e8f8f; /* 19->16 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd4; /* Block 4 */
      #10; /* Clock in */
      data_write <= 128'h848484a3; /* 23->20 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd5; /* Block 5 */
      #10; /* Clock in */
      data_write <= 128'h69a1d132; /* 27->24 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd6; /* Block 6 */
      #10; /* Clock in */
      data_write <= 128'hafaf33ae; /* 31->28 */
      line_addr_write <= 3'd2; /* Line 2 */
      block_addr_write <= 3'd7; /* Block 7 */
      #10; /* Clock in */

      /* Write 4th line */
      data_write <= 128'h2d5a8b5b; /* 3->0 */
      line_addr_write <= 3'd3; /* Line 3 */
      block_addr_write <= 3'd0; /* Block 0 */
      #10; /* Clock in */
      data_write <= 128'h0ef69af2b; /* 7->4 */
      line_addr_write <= 3'd3; /* Line 3 */
      block_addr_write <= 3'd1; /* Block 1 */
      #10; /* Clock in */
      data_write <= 128'h3e3e3daf; /* 11->8 */
      line_addr_write <= 3'd3; /* Line 3 */
      block_addr_write <= 3'd2; /* Block 2 */
      #10; /* Clock in */
      data_write <= 128'h5b8a8909; /* 15->12 */
      line_addr_write <= 3'd3; /* Line 3 */
      block_addr_write <= 3'd3; /* Block 3 */
      #10; /* Clock in */
      data_write <= 128'hafbeaf69; /* 19->16 */
      line_addr_write <= 3'd3; /* Line 3 */
      block_addr_write <= 3'd4; /* Block 4 */
      #10; /* Clock in */

```

```

192      data_write <= 128'h69ae6a8f; /* 23->20 */
          line_addr_write <= 3'd3; /* Line 3 */
          block_addr_write <= 3'd5; /* Block 5 */
          #10; /* Clock in */
          data_write <= 128'h69b59532; /* 27->24 */
          line_addr_write <= 3'd3; /* Line 3 */
          block_addr_write <= 3'd6; /* Block 6 */
          #10; /* Clock in */
          data_write <= 128'hafaf33af; /* 31->28 */
          line_addr_write <= 3'd3; /* Line 3 */
          block_addr_write <= 3'd7; /* Block 7 */
          #10; /* Clock in */
202      /* Write 5th line */
          data_write <= 128'hed5b8b5a; /* 3->0 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd0; /* Block 0 */
          #10; /* Clock in */
          data_write <= 128'hae696aaaf; /* 7->4 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd1; /* Block 1 */
          #10; /* Clock in */
          data_write <= 128'h5b5a5b0e; /* 11->8 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd2; /* Block 2 */
          #10; /* Clock in */
          data_write <= 128'h5b5a8b5b; /* 15->12 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd3; /* Block 3 */
          #10; /* Clock in */
          data_write <= 128'hoe3a3b3c; /* 19->16 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd4; /* Block 4 */
          #10; /* Clock in */
          data_write <= 128'ha2boaeaaf; /* 23->20 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd5; /* Block 5 */
          #10; /* Clock in */
          data_write <= 128'hae848a09; /* 27->24 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd6; /* Block 6 */
          #10; /* Clock in */
          data_write <= 128'h2b2b32af; /* 31->28 */
          line_addr_write <= 3'd4; /* Line 4 */
          block_addr_write <= 3'd7; /* Block 7 */
          #10; /* Clock in */
237      /* Write 6th line */
          data_write <= 128'hoe5b7172; /* 3->0 */
          line_addr_write <= 3'd5; /* Line 5 */
          block_addr_write <= 3'd0; /* Block 0 */
          #10; /* Clock in */
          data_write <= 128'haf6968af; /* 7->4 */
          line_addr_write <= 3'd5; /* Line 5 */
          block_addr_write <= 3'd1; /* Block 1 */
          #10; /* Clock in */
          data_write <= 128'h8a8a080f; /* 11->8 */
          line_addr_write <= 3'd5; /* Line 5 */
          block_addr_write <= 3'd2; /* Block 2 */
          #10; /* Clock in */
          data_write <= 128'h12328a8a; /* 15->12 */
          line_addr_write <= 3'd5; /* Line 5 */
          block_addr_write <= 3'd3; /* Block 3 */
          #10; /* Clock in */
          data_write <= 128'haf77d241; /* 19->16 */

```

```

    line_addr_write <= 3'd5;          /* Line 5 */
    block_addr_write <= 3'd4;         /* Block 4 */
257   #10;   /* Clock in */
    data_write <= 128'h2b543333;    /* 23->20 */
    line_addr_write <= 3'd5;          /* Line 5 */
    block_addr_write <= 3'd5;         /* Block 5 */
#10;   /* Clock in */
262   data_write <= 128'haf848a89;    /* 27->24 */
    line_addr_write <= 3'd5;          /* Line 5 */
    block_addr_write <= 3'd6;         /* Block 6 */
#10;   /* Clock in */
    data_write <= 128'h8a0e82b80;    /* 31->28 */
267   line_addr_write <= 3'd5;          /* Line 5 */
    block_addr_write <= 3'd7;         /* Block 7 */
#10;   /* Clock in */

/* Write 7th line */
272   data_write <= 128'h835b5b8a;    /* 3->0 */
    line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd0;         /* Block 0 */
#10;   /* Clock in */
    data_write <= 128'hf8f8f8f8f;    /* 7->4 */
277   line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd1;         /* Block 1 */
#10;   /* Clock in */
    data_write <= 128'h890909af;    /* 11->8 */
    line_addr_write <= 3'd6;          /* Line 6 */
282   block_addr_write <= 3'd2;         /* Block 2 */
#10;   /* Clock in */
    data_write <= 128'hd1320909;    /* 15->12 */
    line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd3;         /* Block 3 */
287   #10;   /* Clock in */
    data_write <= 128'h697776ad;    /* 19->16 */
    line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd4;         /* Block 4 */
#10;   /* Clock in */
    data_write <= 128'h8a2bafae;    /* 23->20 */
292   line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd5;         /* Block 5 */
#10;   /* Clock in */
    data_write <= 128'h2b4c238a;    /* 27->24 */
297   line_addr_write <= 3'd6;          /* Line 6 */
    block_addr_write <= 3'd6;         /* Block 6 */
#10;   /* Clock in */
    data_write <= 128'h898b2b2a;    /* 31->28 */
    line_addr_write <= 3'd6;          /* Line 6 */
302   block_addr_write <= 3'd7;         /* Block 7 */
#10;   /* Clock in */

/* Write 8th line */
307   data_write <= 128'h5b8a8a8a;    /* 3->0 */
    line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd0;         /* Block 0 */
#10;   /* Clock in */
    data_write <= 128'h69afaf69;    /* 7->4 */
312   line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd1;         /* Block 1 */
#10;   /* Clock in */
    data_write <= 128'h09093e3e;    /* 11->8 */
    line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd2;         /* Block 2 */
#10;   /* Clock in */
    data_write <= 128'h78773e09;    /* 15->12 */
    line_addr_write <= 3'd7;          /* Line 7 */

```

```

    block_addr_write <= 3'd3;           /* Block 3 */
    #10;      /* Clock in */
322   data_write <= 128'h693e3e77;    /* 19->16 */
    line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd4;          /* Block 4 */
    #10;      /* Clock in */
    data_write <= 128'h5b846969;    /* 23->20 */
327   line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd5;          /* Block 5 */
    #10;      /* Clock in */
    data_write <= 128'h8a8a238a;    /* 27->24 */
    line_addr_write <= 3'd7;          /* Line 7 */
332   block_addr_write <= 3'd6;          /* Block 6 */
    #10;      /* Clock in */
    data_write <= 128'h8a8a8a8a;    /* 31->28 */
    line_addr_write <= 3'd7;          /* Line 7 */
    block_addr_write <= 3'd7;          /* Block 7 */
337   #10;      /* Clock in */

    write_en <= 0;

    global_hold <= 0;
342 end

initial begin
  #5000
  $finish;
end

endmodule

```

Listing A.2: Matching Testbench

```

'timescale 1ns / 1ps

module TestBenchControllerStates;
4
parameter DATA_WIDTH=256;
parameter ADDR_WIDTH=7;
reg clk; //Clock Signal
reg reset; //Reset Signal
9 reg write_enable_image1; //Used to write in BRAM1
reg write_enable_image2; //Used to write in BRAM2
reg [DATA_WIDTH-1:0] data_1; //Data that will be introduced in BRAM1
reg [DATA_WIDTH-1:0] data_2; //Data that will be introduced in BRAM2
wire [DATA_WIDTH-1:0] common_part; //Common part
14 reg result_ready; //Signal from Matching Stage to load new patterns
reg result_image1; //Signal From Brief to inform that image1 is over

matching_wrapper uut (
    .reset(reset),
    .clk(clk),
    .data_1(data_1),
    .data_2(data_2),
    .result_image1(result_image1),
    .write_enable_image1(write_enable_image1),
    .write_enable_image2(write_enable_image2),
    .r_smallest_difference(),
    .r_index_best()
);

```

```

29 | initial begin
// Initialize Inputs
    clk = 0;
    reset=1;
    write_enable_image1 = 0;
34 |     write_enable_image2 = 0;
    result_image1=0;
    result_ready=0;
#10;
    reset=0;
39 |     write_enable_image1 = 1;
    data_1=53;
#10;
    write_enable_image1 = 0;
#22;
44 |     write_enable_image1 = 1;
    data_1=80;
#10;
    write_enable_image1 = 0;
#22;
49 |     write_enable_image1 = 1;
    data_1=300;
#10;
    write_enable_image1 = 0;
#22;
54 |     write_enable_image1 = 1;
    data_1=10;
#10;
    write_enable_image1 = 0;
#22;
59 |     write_enable_image1 = 1;
    data_1=210;
#10;
    write_enable_image1 = 0;
#22;
64 |     write_enable_image1 = 1;
    data_1=201;
#10;
    write_enable_image1 = 0;
#22;
69 |     write_enable_image1 = 1;
    data_1=136;
#10;
    write_enable_image1 = 0;
#22;
74 |     write_enable_image1 = 1;
    data_1=178;
#10;
    write_enable_image1 = 0;
#5;
79 | #22;
    result_image1=1;
    write_enable_image2 = 1;
    data_2=150;
#10;
    write_enable_image2 = 0;
#22;
    write_enable_image2 = 1;
    data_2=111;
#10;
    write_enable_image2 = 0;
#22;
    write_enable_image2 = 1;
    data_2=215;
#10

```

```
94      write_enable_image2 = 0;
#22;
      write_enable_image2 = 1;
      data_2=101;
#10
99      write_enable_image2 = 0;
#22;
      write_enable_image2 = 1;
      data_2=43;
#10
104     write_enable_image2 = 0;
#22;
      write_enable_image2 = 1;
      data_2=78;
#10
109     write_enable_image2 = 0;
#22;
      write_enable_image2 = 1;
      data_2=172;
#10
114     write_enable_image2 = 0;
#22;
      write_enable_image2 = 1;
      data_2=205;
#10
119     write_enable_image2 = 0;
      result_image1 = 0;
#100;
end
//Clock generator
always #5
  clk = ~clk;
endmodule
```

B | Scripts

Listing B.1: Small 32x8 Image to Array Conversion Python Script

```
import cv2
import numpy as np

3 # Load the image
image = cv2.imread('32x8.jpg', cv2.IMREAD_GRAYSCALE)

arr = np.asarray(image)

8 # print(arr)

string = '['

13 for line in arr:
    string = string + '['
    for pixel in line:
        string = string + str(pixel) + ', '
    string = string[0:len(string)-2] + '],\n'

18 string = string[0:len(string)-2]
print(string)
```

Listing B.2: Real Image Snippet to Array Conversion Python Script

```
import cv2
import numpy as np

4 image = cv2.imread('images/simba-the-lion-king-2019-movie.jpg')
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

arr = np.array(gray)
flat_array = arr.flatten()

9 hex_array = [hex(x) for x in flat_array]

print("/----- ARR -----\\")
print(arr)
print("\\\-----\\n\n")

14 string = '['

for line in arr[0:8]:
    string = string + '['
    for pixel in line[0:32]:
        string = string + str(pixel) + ', '
    string = string[0:len(string)-2] + '],\n'

24 string = string[0:len(string)-2] + ']'

print("/----- String -----\\")
print(string)
print("\\\-----\\n\n")

29 string = '['

for line in arr[0:8]:
    string = string + '['
    for pixel in line[33:64]:
```

```
        string = string + str(pixel) + ', '
        string = string[0:len(string)-2] + '],\n'

    string = string[0:len(string)-2] + ']'
39   print("/----- String -----\\\"")
print(string)
print("\\\-----/\\n\\n")

44 string = '['

for line in arr[159:167]:
    string = string + '['
    for pixel in line[140:172]:
        string = string + str(pixel) + ', '
        string = string[0:len(string)-2] + '],\n'

    string = string[0:len(string)-2] + ']'

54   print("/----- String -----\\\"")
print(string)
print("\\\-----/\\n\\n")

cv2.imshow('Original image',image)
59 cv2.imshow('Gray image', gray)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

C | Project Planning

Rotation Day																	
Module x Week		08-Mar	Sprint 0	29-Mar	Sprint 1	05-Apr	Sprint 2	19-Apr	Sprint 3	03-May	Sprint 4	17-May	Sprint 5	31-May	Sprint 6	14-Jun	Sprint 7
Group	Colour																
I	Yellow	Hugo Carvalho	João Faria	Francisco Dias	Daniel Cunha	Rui Costa	Gonçalo Freitas	João Carvalho	André Lopes	Samuel Carvalho							
II	Green	Rui Duro	Pedro Matias	José Gomes	Dinis Fernandes	Alexandre Mano	João Relvas	António Capela	Jorge Pinto	João Bento							
III	Blue	Heitor Silva	Vítor Ribeiro	Carlos Ribeiro	João Borlido	Simão Leite	Rui Esteves	Mariana Duarte	João Costa	Alexandre Gonçalves							
IV	Purple	Tiago Aston	Tiago Costa	José Mendes	Pedro Lages	Pedro Duarte	Diogo Matias	Sara Pereira	Francisco Rocha	João Silva							

Members - Sprint 0													
Group	Colour	Tiago Aston	Daniel Cunha	Pedro Matias	José Gomes	Dinis Fernandes	Carlos Ribeiro	João Borlido	Pedro Lages	X	X	X	
I	Yellow	Tiago Aston	Daniel Cunha	Pedro Matias	José Gomes	Dinis Fernandes	Carlos Ribeiro	João Borlido	Pedro Lages	X	X	X	
II	Green	Hugo Carvalho	Francisco Dias	Samuel Carvalho	João Relvas	Alexandre Mano	António Capela	Jorge Pinto	José Mendes	Vítor Ribeiro	João Costa	Francisco Rocha	
III	Blue	Rui Duro	Gonçalo Freitas	Alexandre Mano	João Silva	Diogo Matias	Pedro Duarte	Rui Esteves	Tiago Costa	X	X	X	
IV	Purple	Heitor Silva	João Faria	Rui Costa	João Carvalho	André Lopes	Mariana Duarte	Sara Pereira	Simão Leite	X	X	X	

Members - Remaining Sprints													
Group	Colour	Tiago Aston	Daniel Cunha	Pedro Matias	José Gomes	Dinis Fernandes	Carlos Ribeiro	João Borlido	Pedro Lages	X	X	X	
I	Yellow	Hugo Carvalho	Francisco Dias	Samuel Carvalho	João Relvas	Alexandre Mano	António Capela	Jorge Pinto	José Mendes	Vítor Ribeiro	João Costa	Francisco Rocha	
II	Green	Rui Duro	Gonçalo Freitas	Alexandre Mano	João Silva	Diogo Matias	Pedro Duarte	Rui Esteves	Tiago Costa	X	X	X	
III	Blue	Heitor Silva	João Faria	Rui Costa	João Carvalho	André Lopes	Mariana Duarte	Sara Pereira	Simão Leite	X	X	X	
IV	Purple												

Figure C.1: ESRG Project Plan

D | Schematics

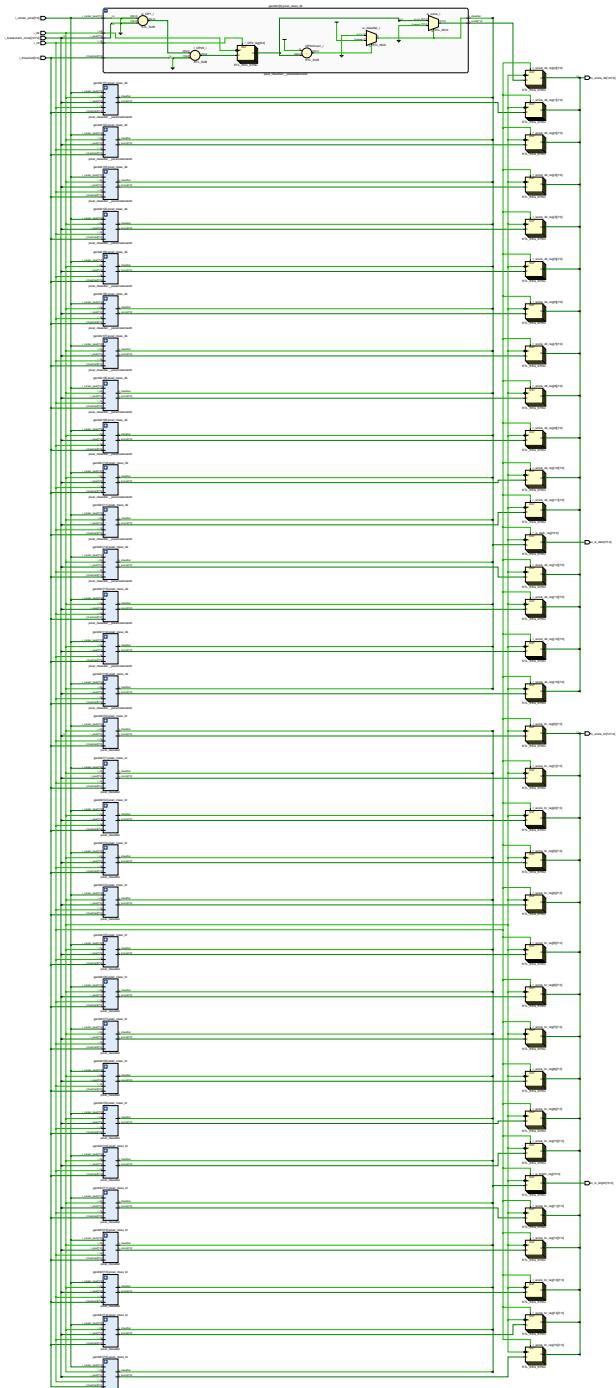


Figure D.1: Classifier Schematic (FAST-12)

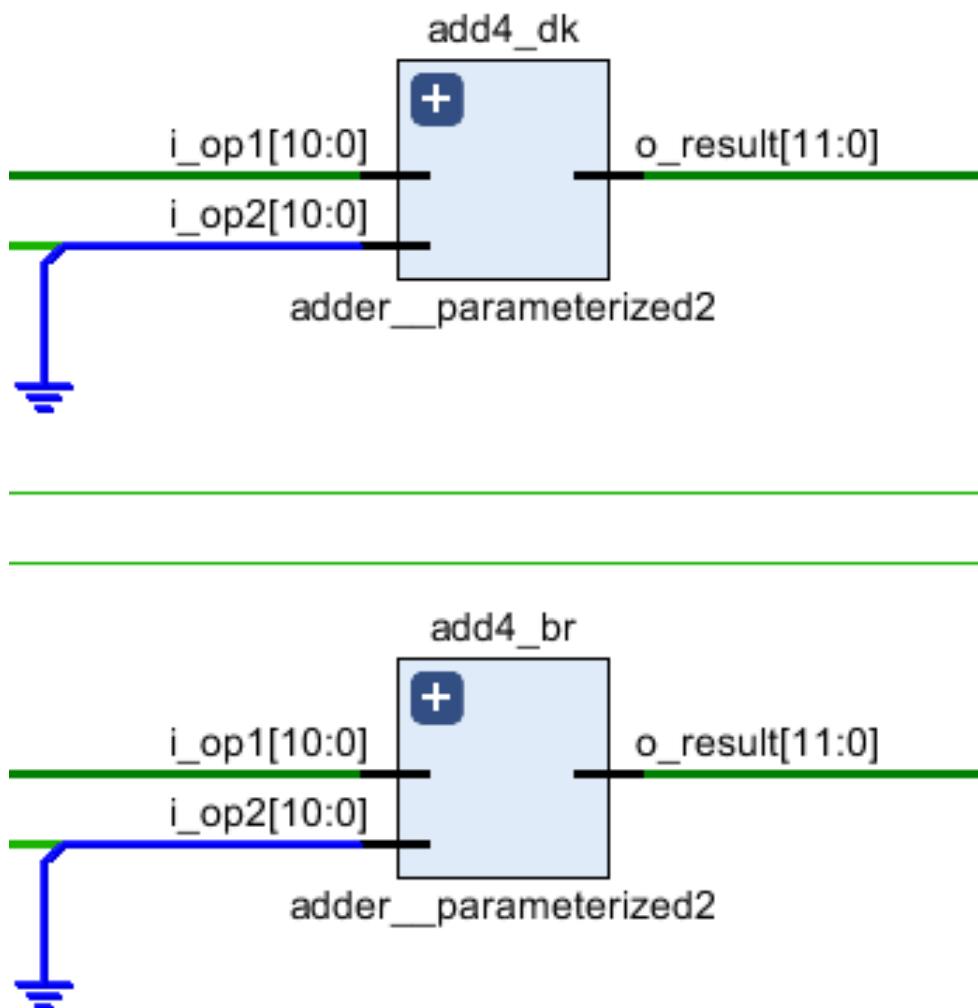


Figure D.2: Schematic: Bit Concatenation

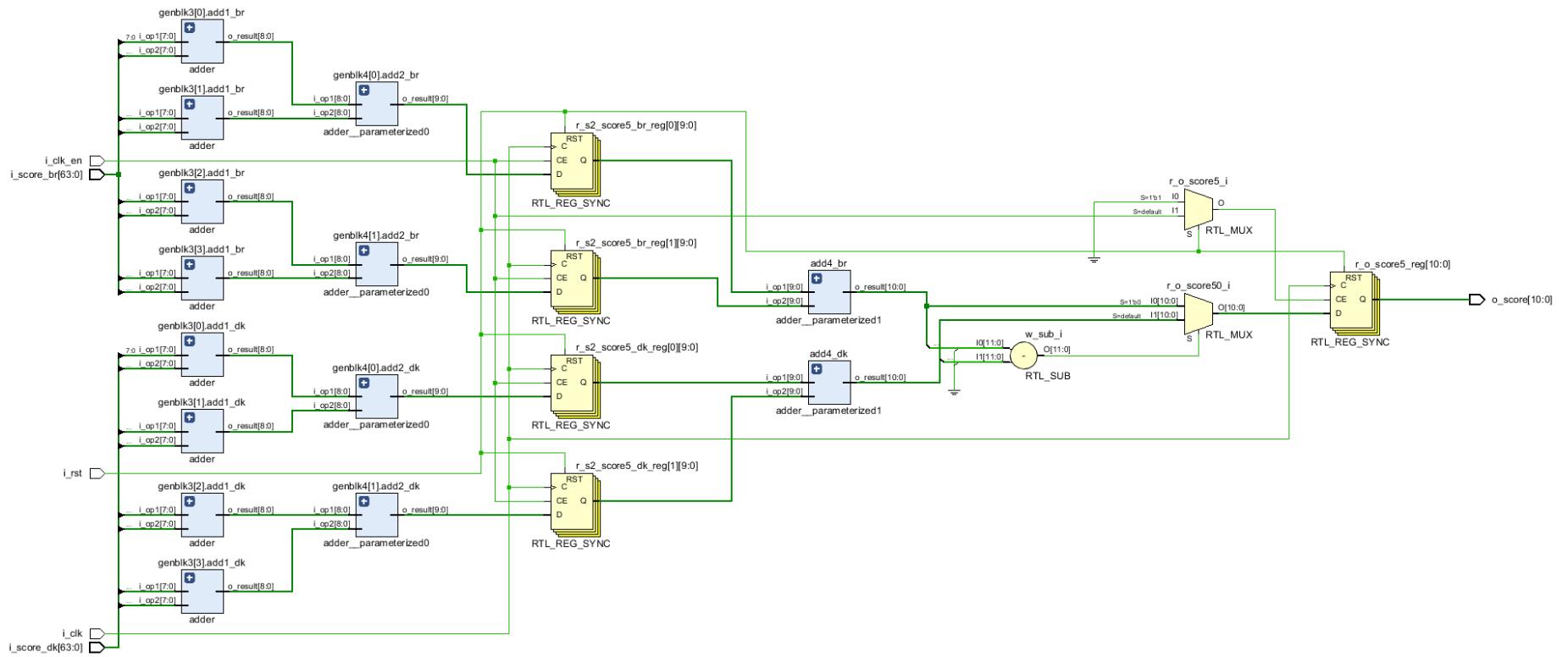


Figure D.3: Schematic: Score FAST-5

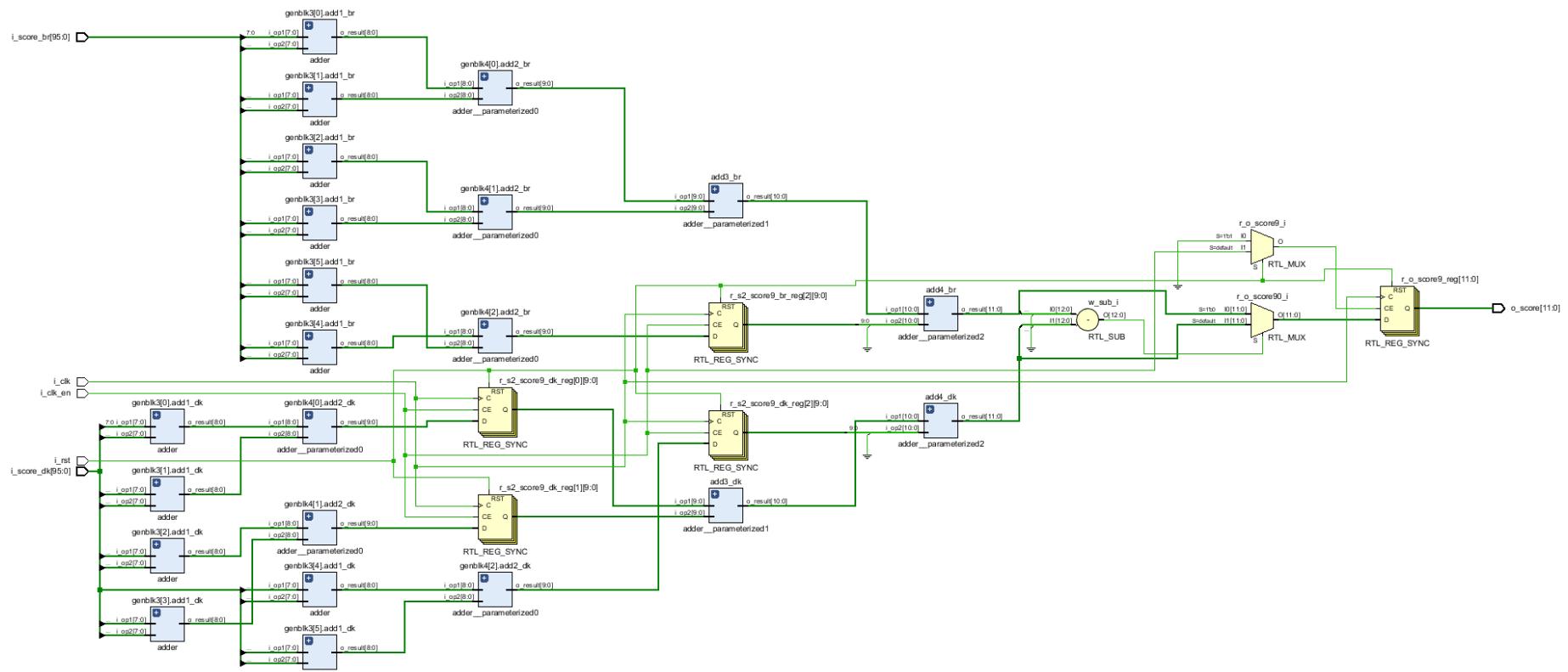


Figure D.4: Schematic: Score FAST-9

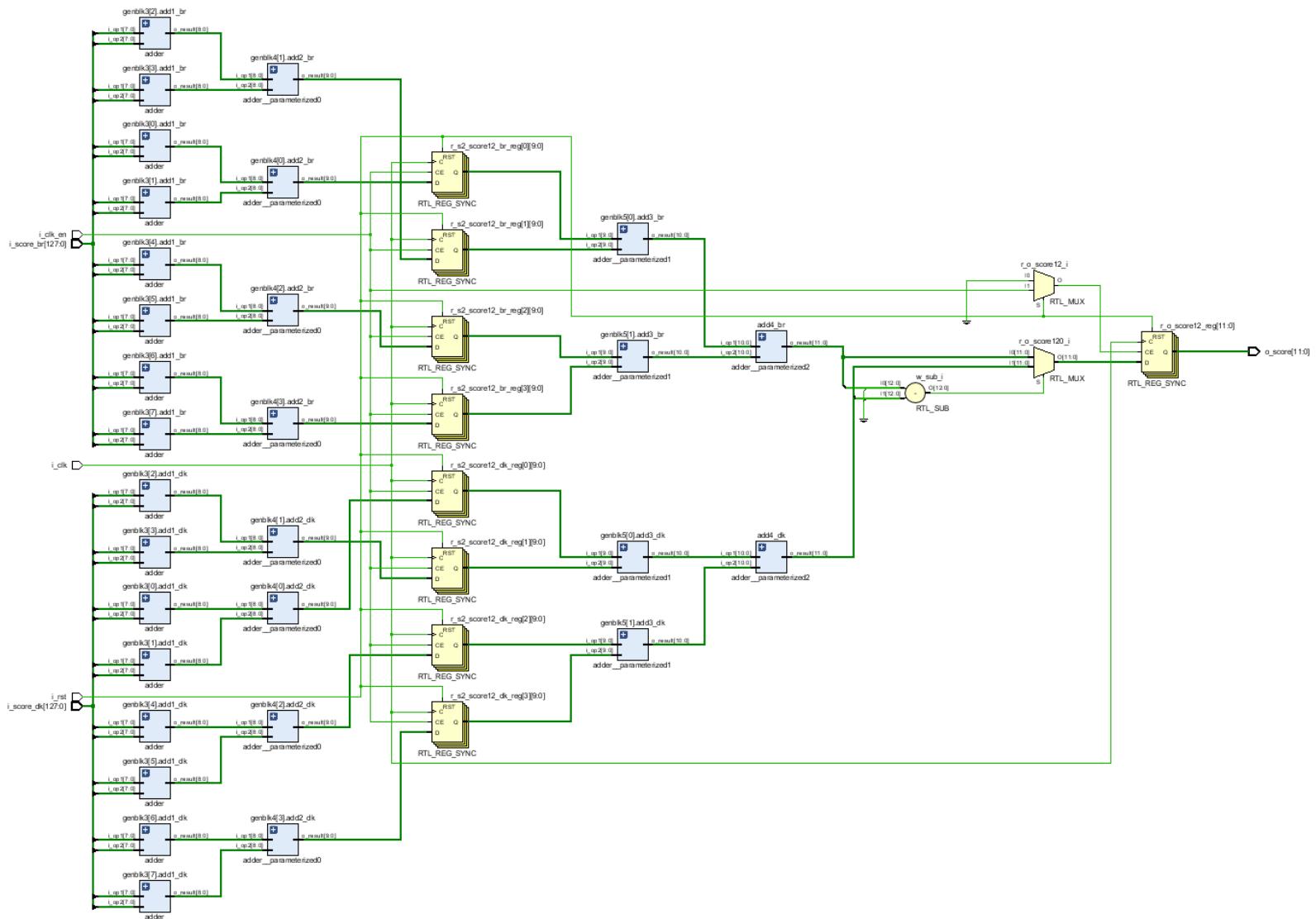


Figure D.5: Schematic: Score FAST-12

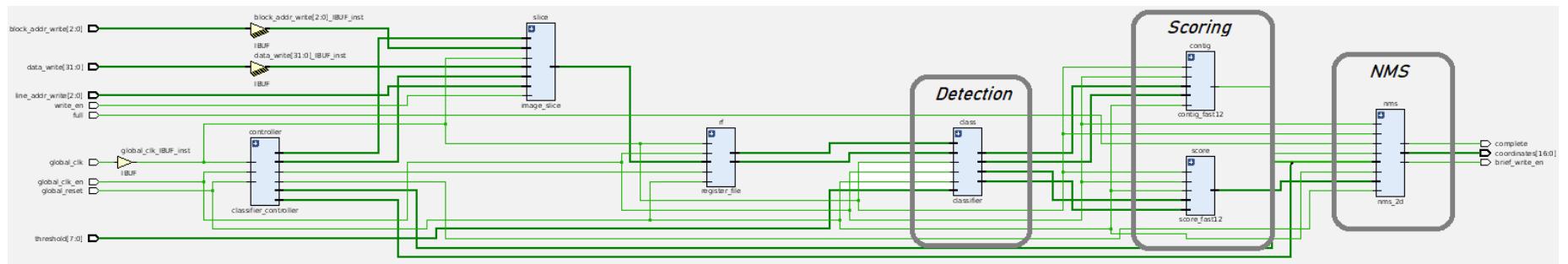


Figure D.6: Schematic: Integration of BRISK detection stages