

FPGA implementatie van het HOG algoritme

Stef VAN WOLPUTTE

Promotor: ing. Van Beeck Kristof Masterproef ingediend tot het behalen van de graad van master of Science in de industriële wetenschappen: Elektronica-ICT Afstudeerrichting Elektronica

©Copyright KU Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor(en) als de auteur(s) is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot KU Leuven campus De Nayer, Jan De Nayerlaan 5, B-2860 Sint-Katelijne-Waver, +32-15-316944 of via e-mail iiw.thomasmore.denayer@kuleuven.be.

Voorafgaande schriftelijke toestemming van de promotor(en) is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Dankwoord

Gedurende mijn thesis heb ik de steun en hulp gekregen van verschillende mensen. Via deze weg zou ik hen graag bedanken.

Als eerste zou ik graag mijn promotor, Van Beeck Kristof, willen bedanken. Bij hem kon ik steeds met al mijn vragen terecht en zijn raad en advies hebben mij doorheen dit project geholpen en me veel bijgeleerd.

Mijn dank gaat ook uit naar de onderzoeks groep EAVISE om mij deze kans te bieden. Ik kon steeds op steun rekenen om zo optimaal mogelijk te kunnen werken.

Als laatste zou ik graag mijn gezin, familie, vriendin en kotgenoten willen danken. Zij steunden mij doorheen dit jaar en bij het nalezen van deze scriptie.

Stef Van Wolputte
Herenthout, mei 2014

Abstract

English abstract

The HOG algorithm is a good and reliable approach to perform person detection. This algorithm has one disadvantage: to achieve reasonable speeds, we need to use fast hardware. Using such hardware in real life is often unfeasable. Therefore, we aim to speed up this algorithm with this case study.

We started our project with a comprehensive study of the HOG algorithm. Next, we investigated which hardware platform would meet our requirements. We chose the Xilinx Zedboard as our hardware platform since it includes both a processor and an FPGA fabric, this would allow for a software-hardware combination. Then, we implemented a hardware block which describes the gradient vector calculating part. Our profiler indicated this is a slow part of the HOG algorithm. We have theoretically determined that our implementation would accelerate the gradient calculation part by 50%. Then, we also wrote test programs which included a hardware-software combination. Furthermore, we succeeded in booting Linux on the Zedboard. Unfortunately, we do not have a complete implementation of the algorithm. However, all parts are tested and implemented separately and we also noticed speed improvements. Finally, we provided some possibilities for further speedups.

Nederlandstalig abstract

Het HOG algoritme is een goed en betrouwbaar systeem om persoonsdetectie te verwezenlijken. Dit algoritme heeft echter één nadeel: zijn snelheid. Om voldoende snelle uitvoeringstijd te bekomen is er nood aan relatief krachtige hardware. Deze hardware is helaas niet altijd toepasbaar in de praktijk, daarom is het doel van deze thesis om het algoritme te versnellen. We zijn ons project gestart met een uitgebreide studie rond het algoritme. Daarna hebben we onderzocht op welk hardware platform we onze implementatie konden verwezenlijken. We kozen voor het Xilinx Zedboard als hardware platform omdat deze zowel een processor als een FPGA fabric bevat. Zo konden we het gradiëntberekenend deel versnellen door middel van een hardware implementatie op de FPGA en de andere delen van het algoritme uitvoeren op de processor. Onze profiler toonde immers aan dat het gradiëntbepalend deel een traag onderdeel was van het HOG algoritme. We hebben daarna theoretisch kunnen aantonen dat onze implementatie het gradiëntberekenend deel met 50% versnelt. Daarna hebben we ook testprogramma's gemaakt die een hardware-software combinatie realiseerden. Ook hebben we een Linux kunnen booten op het Zedboard.

Ondanks het feit dat we geen totale implementatie konden realiseren, hebben we al de nodige onderdelen geïmplementeerd, getest en hebben we snelheidsverbeteringen vastgesteld. Als laatste hebben we suggesties beschreven om nog verdere snelheidsverbeteringen te verkrijgen.

Short summary

Introduction

There are various algorithms to perform person detection. A good and reliable system for person detection was demonstrated by Dalal and Triggs. They suggested the Histograms of Oriented Gradients, or HOG algorithm. This algorithm produces good results, but also forms the basis for other object detection algorithms. A disadvantage of this algorithm is its speed. For the detection to be performed in real time there, is a need for relatively fast hardware. This hardware is often not very practical. Therefore, we want to accelerate the process, so we can implement the HOG algorithm on an embedded platform. We want to achieve a speedup through implementation on an FPGA. Given the complexity of the algorithm, the first thing to do is to examine which parts are suitable for a hardware implementation. We also had to examine whether it is possible to realize a software-hardware architecture which only implemented the parts that lend themselves to implementation on an FPGA and execute the remaining parts on a processor.

HOG algorithm

HOG, or Histograms of Oriented Gradients, is an algorithm that will detect objects based on their contours or gradients. This method makes it possible to detect objects that can occur in many forms. Detection of persons is an example of this. The first step in the HOG algorithm is color normalization; this will improve the contrast of the input image. The next step is to compute the gradient. The angles of the gradient vectors are directed based on the contrast changes and the magnitude of the vectors is proportional to the contrast difference. Then, the gradients are taken together in cells of 8x8 pixels. Subsequently, the data in this cell are combined into a histogram. In order to be resistant to fluctuations in contrast, the cells are combined into blocks of 2x2 cells. The histograms of these blocks are then sent to an SVM. An SVM is a supervised learning model that analyzes data and recognizes patterns. After the SVM is trained, it will be able to decide whether or not a person appears on the image.

Hardware platform

After the study about the HOG algorithm we investigated which hardware platform is suited for implementation. After comparing the Altera and Xilinx board development platforms, we have decided to perform our implementation on the Xilinx Zedboard. This Zedboard is a development kit based on the Zynq SoC. The board holds all the necessary I/O, even should we ever decide to process streaming video. It also contains an SD-slot so we can boot a Linux

on the SoC. The Zynq SoC consists of both an ARM Cortex-A9 processor and a Xilinx Artix-7 FPGA. The communication between software and hardware is provided by the AXI bus.

Hardware implementation

After we had decided to use the Zedboard for our implementation, we studied which part of the algorithm we were going implement first. We decided to convert the gradient calculating part first. With the profiler Valgrind we noticed that the gradient vector calculating part consumed 30% of the total computation time. This part of the algorithm also lent itself to hardware implementation: the sobel filter can be executed quickly and this part can be split up so it will be able to work in parallel. This implementation consists of two main parts. The first part is the gradient calculating part itself. This section begins with two line buffers which ensure that the input pixels arrive to the next block in the correct way. The next part is the sobel filter. This filter takes the derivative of the image. As a result, a large value is obtained at presence of edges and a very low value at low contrast fluctuations. The output of the sobel filter will then go to a block where the data is made ready for the calculation of the square root and arctangent. These calculations are implemented in the CORDIC algorithm, and their input needs to be adjusted. After the CORDIC cores, both the magnitude and the angle of the gradient vector are determined. To calculate a frame of 640x480 pixels our implementation would take approximately 3.072 milliseconds to complete the calculation.

The second hardware block will connect the first part to the AXI bus so it can communicate with the processor. The ARM will store the image to be processed in RAM memory. This data must be transferred, through the AXI bus, to the gradient calculation. These results should then be re-saved in RAM. To accomplish this, the gradient calculation must be embedded between two FIFO registers. This assembly is then monitored by means of a master controller. This controller starts when it receives a go-signal from the ARM through the common register. The controller then determines, on the basis of the status of the FIFO, whether data needs to be read or written via the AXI bus. For communication control with the AXI bus, we have built another FSM. To transfer an image of 640x480 pixels between the RAM and the FIFOs would take approximately 14.5 milliseconds using an example configuration from a Xilinx datasheet. This would make the total calculation time 17.5 milliseconds. We also measured that a calculation in software would take about 40 milliseconds, bringing our speed up by 50%.

Software environment

To read images from the SD card and run existing software, we have made for a boot file to boot in Linaro Linux from the SD card. However, this is not a simple procedure. Our own hardware block had to be re-inserted and the device tree had to be adjusted manually. After creating our own boot file, we have run some tests with a software-hardware implementation. We have created a program where the ARM sends a number to the FPGA, this number was then made visible through the LEDs. We have also sent data in the reverse direction, from the FPGA back to the ARM. Furthermore, we have also created a test that simulated the data transaction between the FIFOs and RAM using the AXI bus.

Further work

To speed up our implementation, several additional adjustments can be made. First of all, the input image can be split up into several parts so the gradient calculation part can work in parallel. Next, our gradient calculating part can be used multiple times between the existing FIFO structure. Of course, the total processing time can be reduced by implementing more parts of the algorithm into the FPGA. Ideally, the next part to be implemented in hardware, should be an operation before or after the gradient calculating part.

Conclusion

The aim of this thesis was to design a hardware implementation for the HOG algorithm to in order to speed up the processing time. At the end of the project we managed to decide which hardware board is most suitable for our purpose and which part should be implemented first. We chose to implement the gradient calculating part first and use the Xilinx Zedboard to create the software-hardware implementation. Although we could not make a complete implementation of the algorithm, we succeeded in implementing, testing all the necessary components and calculated that we speed up the gradient calculating part by 50%. We also created a bootable Linux file on the SD card and came up with ideas for further speed-ups.

Inhoudsopgave

1 Inleiding	1
1.1 Situering en doelstelling	1
2 Het HOG algoritme	3
2.1 Situering	3
2.2 Inleiding	5
2.3 Kleurnormalisatie	6
2.4 Gradiënt vectoren berekenen	7
2.5 Gradiënt histogrammen	9
2.6 Bloknormalisatie	11
2.7 Lineaire SVM	11
2.8 Onderzoeken rond het HOG algoritme	12
3 Hardwareplatform	13
3.1 Zedboard	13
3.2 Zynq	14
3.3 AXI bus	15
3.4 Technische uitleg over de FPGA	17
3.4.1 Configurable logic blocks (CLB)	18
3.4.2 Routering	18
3.4.3 I/O blokken	19
3.5 ARM processor	19
3.6 Ontwerpprocedure met Xilinx ISE WebPACK	20
4 Implementatie van de gradiëntberekening in FPGA	21
4.1 Blokschema	21
4.1.1 Lijnbuffers	22
4.1.2 Sobelfilter	22
4.1.3 Berekeningsblok	24
4.1.4 Aanpassingsblok	24
4.1.5 Voorbeeld	25
5 CORDIC algoritme	28

5.1	Werking van CORDIC	28
5.2	Voorbeeld van het CORDIC algoritme	29
5.3	Fouten die optreden bij CORDIC	30
5.4	Gebruik van CORDIC in VHDL	31
6	Gradiëntberekening aansluiten op de AXI bus	32
6.1	Het blokschema	32
6.2	Mastermodule	33
6.3	Inhoud van de User logic file	33
6.3.1	Data pad	36
6.3.2	Controller	39
7	Booten van Linux OS in combinatie met eigen hardware	43
7.1	Importeren van eigen hardware blok	43
7.2	Aanpassen van device tree	44
7.3	Genereren van de boot file	45
7.4	Compile Linux Kernel	45
7.5	Klaarmaken van de SD-kaart	46
8	Hardware-software communicatie	47
8.1	Communicatie via het gemeenschappelijk register	47
8.2	Inlezen van data via de AXI bus	48
9	Resultaten	50
9.1	Mogelijke verbeteringen	50
9.1.1	Wijzigingen aan huidige implementatie	51
9.1.2	Mogelijke uitbreidingen	51
10	Conclusie	52

Lijst van figuren

2.1	Kindertekening van een wagen	4
2.2	De gradiënten van een wagen	4
2.3	Gradiënten van een persoon	5
2.4	Resultaat van een detectie	5
2.5	Blokschema HOG	5
2.6	Blokschema van HOG descriptor [9]	6
2.7	Het effect van normalisatie [9]	7
2.8	Blokschema om gradiënt vectoren te berekenen	7
2.9	Het effect van het sobel filter [9]	8
2.10	Illustratie van de blocks [9]	8
2.11	De berekende vector gradiënt op de voorbeeldafbeelding [9]	9
2.12	Voorbeeld van de grootte van een 8x8 cel [9]	10
2.13	Voorbeeld van een 9-bin histogram [9]	10
2.14	Illustratie van de blocks [9]	11
2.15	Voorbeeld van een getrainde SVM	12
3.1	Opstelling Zedboard	14
3.2	Blokschema van de Zynq	14
3.3	Opbouw van Zynq met AXI bus tussen PS en PL	15
3.4	Handshake principe AXI	16
3.5	Lezen en schrijven op de AXI bus	16
3.6	Timing van een 4 byte write actie over AXI bus	17
3.7	De architectuur van een FPGA	18
3.8	De architectuur van een CLB	18
3.9	Opbouw van een schakelblok	19
3.10	Blokschema van de ARM processor	19
3.11	Design flow voor Zynq	20
4.1	Blokschema van de VHDL code voor de gradiëntberekening	21
4.2	Testbench van de lijnbuffers	22
4.3	Resultaten van gebruik van verschillende sobel filters	23
4.4	Blokschema van sobel filter met voorbeeldwaarden	23

4.5	Testbench van het sobel masker	24
4.6	Testbench van het berekeningsblok	24
4.7	Voorbeeld van input testbench	25
4.8	Berekende uitkomst van testbench	26
4.9	Blokschema met voorbeeldwaardes	26
5.1	Planaire draaiing bij CORDIC	29
5.2	Rotatie mode	29
5.3	Vector mode	29
5.4	Werking CORDIC	30
5.5	Vector mode	31
5.6	Unrolled architectuur bij CORDIC	31
6.1	Blokschema voor implementatie en aansturing van het gradiëntberekenend deel	33
6.2	De hiërarchie bij de files van een mastermodule	33
6.3	Een overzicht van de wijze waarop de gemeenschappelijke registers gebruikt worden	34
6.4	Blokschema van een digitaal systeem	35
6.5	Het digitaal systeem van de User logic file	35
6.6	FSM van de converter	36
6.7	Testbench van de converter	37
6.8	Testbench totaal blok deel 1: vullen van de input FIFO	37
6.9	Testbench totaal blok deel 1: lezen van de input FIFO	38
6.10	Testbench totaal blok deel 1: vullen van de output FIFO	38
6.11	Testbench totaal blok deel 1: lezen van de output FIFO	38
6.12	FSM van de master controller	39
6.13	Testbench FSM master command deel 1: vullen van FIFO's	40
6.14	Testbench FSM master command deel 2: vullen en lezen van de output FIFO	40
6.15	Testbench FSM master command deel 3: vullen van input FIFO	41
6.16	Testbench FSM master command deel 4: Einde van een frame	41
6.17	Het FSM om data te lezen van de AXI bus	42
6.18	Het FSM om data te schrijven naar de AXI bus	42
7.1	Voorbeeld van een gebruikers bibliotheek in XPS	44
7.2	Voorbeeld van de ISE hiërarchy	44
8.1	FSM voor het testprogramma voor het vullen en lezen van FIFO's via de AXI bus	48

Lijst van afkortingen

ARM	Acorn RISC Machine
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
CLB	Configurable Logic Blocks
CORDIC	COordinate Rotation for DIgital Computer
DDR SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
ELF	Executable and Linking Format
FPGA	Field Programmable Logic Controller
FSM	Finite State Machine
FIFO	First In - First Out
FSBL	First Stage Boot Loader
GPU	Graphics Processing Unit
HOG	Histograms of Oriented Gradients
IO	Input/Output
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
MIO	Multiplexed Input Output
MUX	MULTipleXer
NGO	NG Linker Object
OS	Operating System
PS	Processing System
PL	Programmable Logic
RISC	Reduced Instruction Set Computing
SURF	Speeded Up Robust Features
SQRT	Square Root
SVM	Support Vector Machine
SoC	System on Chip
UCF	User Constraint File
VHDL	VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
ZEDBOARD	Zynq Evaluation and Development Board

1

Inleiding

1.1 Situering en doelstelling

Er bestaan verschillende algoritmes om objectdetectie te verwezenlijken. Een goed en betrouwbaar systeem voor persoonsdetectie werd aangetoond door Dalal en Triggs [3]. Zij stelden het Histograms of Oriented Gradients, of HOG algoritme voor. Dit algoritme levert goede resultaten op maar vormt deze dagen ook de basis voor andere, HOG gebaseerde, persoonsdetecties zoals onder andere [5]. Een nadeel van dit algoritme is zijn snelheid. Om de detectie real-time te kunnen uitvoeren is er nood aan relatief krachtige hardware. Het gebruik van zulke hardware is vaak moeilijk praktisch toe te passen. Daarom kijken wij naar een implementatie op een embedded platform waardoor een real-time applicatie moeilijk wordt door de rekenintensiteit van het algoritme. Om het proces te versnellen zijn er een paar mogelijkheden: Het zoekalgoritme zou efficiënter gemaakt kunnen worden, de zoekruimte kan verkleind worden, of er kan een parallelle implementatie op GPU of een implementatie op een FPGA gerealiseerd worden. Het doel van deze thesis is om het HOG objectdetectie algoritme te versnellen op basis van deze laatste optie.

Vertrekkende van een geschreven C- en Matlab code moet er gekeken worden of het algoritme, al dan niet gedeeltelijk, kan omgezet worden naar een FPGA implementatie, om zodoende een versnelling te realiseren.

Gezien de complexiteit van het algoritme moet eerst onderzocht worden welke delen zich lenen tot een hardware implementatie. Daarnaast moet ook een geschikt platform gevonden worden waarop we het algoritme gaan implementeren. Ook moet onderzocht worden of het mogelijk is om een software-hardware architectuur te realiseren waarbij enkel de delen die eenvoudig paralleliseerbaar zijn te implementeren in de FPGA fabric. De overige delen kunnen dan uitgevoerd worden door een processor.

Deze masterproef is uitgewerkt in de onderzoeksgrond EAVISE (Embedded Artificially intelligent VISion Engineering). Dit is een onderzoeksgrond die gevestigd is in campus De Nayer. Het doel van de EAVISE onderzoeksgrond is om state-of-the-art visiesystemen te ontwikkelen voor industriële problemen.

Deze scriptie is zodanig opgebouwd dat er eerst een algemene beschrijving van het algoritme gegeven wordt in hoofdstuk 2. Daarna wordt het hardwareplatform en de ontwerpprocedure besproken in 3. In hoofdstuk 4 wordt vervolgens dieper ingegaan op de implementatie van het gradiëntberekenend deel. Verder bespreek ik in hoofdstuk 5 de werking van het COR-DIC algoritme, omdat dit gebruikt wordt tijdens 4. Vervolgens wordt besproken hoe we ons hardwareblok aansluiten op de AXI bus, deze bus zorgt voor de koppeling tussen de processor en de FPGA. Daarna volgt er een uiteenzetting over hoe een hardware-software combinatie is verwezenlijkt in een Linux omgeving in hoofdstukken 7 en 8.

Als laatste hebben we onze resultaten samengevat en nog enkele manieren voor mogelijke verbeteringen voorgesteld. Op basis hiervan vormden we onze conclusie over deze thesis.

2

Het HOG algoritme

In dit hoofdstuk wordt het HOG algoritme, dat we willen implementeren, verder uitgediept. Eerst wordt door middel van een situering duidelijk gemaakt waarom er nood is aan dit algoritme. Vervolgens zullen we in sectie 2.2 een inleiding geven over het algoritme aan de hand van een blokschema. Daarna zal elk blok van dit schema in een aparte sectie worden uitgediept.

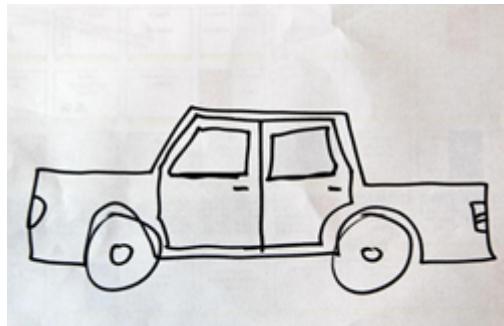
2.1 Situering

In de laatste jaren is de automatisatie met behulp van beelden sterk vooruit gegaan. In de beginjaren van de beeldverwerking waren er algoritmes die randen konden detecteren en voorwerpen konden segmenteren. Vervolgens zijn er algoritmes ontstaan om bepaalde voorwerpen te gaan herkennen op basis van een referentiebeeld. SURF [1] is bijvoorbeeld zo een algoritme dat op een inkomend beeld *local features* gaat proberen te herkennen van een object dat men wil gaan herkennen. Deze technieken heten objectdetectie. Men gaat aan de hand van een referentie een object gaan detecteren. Zo kan men bijvoorbeeld een stopbord ingeven als object dat men wil herkennen en vervolgens videobeelden nemen op straat. Men kan dan op deze inkomende beelden op zoek gaan naar dat specifiek verkeersbord. Omdat de vorm en het uitzicht van een stopbord vastliggen, kan men dit doen aan de hand van een voorbeeldafbeelding. Op de inkomende beelden kan een eventueel stopbord voorkomen in verschillende schalen, rotaties of vervormingen, maar het object blijft steeds herkenbaar.

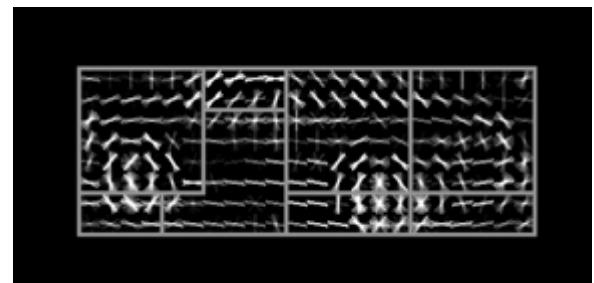
De laatste jaren gaat men echter nog verder, men wil nu objecten gaan herkennen in plaats van detecteren. Niet elk object kan immers vergeleken worden met een referentiebeeld. Zo kan men bijvoorbeeld niet aan de hand van één bepaalde auto al de voorbijrijdende auto's op een snelweg herkennen. Elke auto is echter van een ander merk, een andere categorie of uitvoering. Hetzelfde probleem heeft men wanneer men personen gaat detecteren. Elke persoon is immers anders en ook de kleding en houding van deze personen zijn nooit hetzelfde. Hierdoor kan men niet aan de hand van enkele voorbeeldfoto's elke persoon in het straatbeeld gaan herkennen en daarom zijn voorgaande technieken uitgesloten.

Dit probleem kan worden opgelost door middel van algoritmes die een object herkennen op basis van een model in plaats van een referentiebeeld. Deze algoritmes gaan een object niet detecteren zoals hierboven beschreven, maar herkennen een object aan de hand van zijn specifieke kenmerken. Zo is niet elke auto hetzelfde, maar elke auto kan wel beschreven worden aan de hand van zijn contouren of *gradiënten*.

Wanneer een kind een wagen tekent zal dit er typisch uitzien als figuur 2.1. Zo een tekening maakt niet duidelijk van welk merk de wagen is, of welke soort wagen, maar iedere persoon over heel de wereld weet dat dit een wagen voorstelt. Een soortgelijke methodologie wordt gebruikt bij objectherkenning. We gaan de contouren van de inkomende beelden vergelijken met die van een model. Dit model is opgesteld door zeer veel foto's uit een databank. Zo geeft figuur 2.2 de gradiënten van een auto weer. Deze zijn zeer gelijkaardig aan de kindertekening van figuur 2.1.



Figuur 2.1: Kindertekening van een wagen



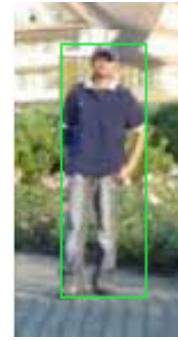
Figuur 2.2: De gradiënten van een wagen

Deze techniek kunnen we nu ook toepassen bij persoonsdetectie. Elke persoon is anders en zelfs dezelfde persoon zal steeds een ander voorkomen hebben door zijn kleding. Ook de momentele houding van een persoon varieert constant. Toch is het mogelijk om door een databank aan te leggen met foto's van verschillende personen een model te maken. Daarna kunnen we met dit model personen gaan herkennen op basis van gradiënten, zoals hierboven beschreven met het voorbeeld van de auto's. Een persoon is immers, net zoals een auto, te beschrijven op basis van algemene kenmerken. Als we vervolgens een zelflerend algoritme een grote verzameling foto's geven, met en zonder personen erop, dan kunnen we dit trainen om op basis van deze gradiënten een beslissing te maken of er zich al dan niet een persoon op de afbeelding bevindt. De gradiënten die bepaald zijn op figuur 2.3 geven duidelijk deze algemene kenmerken van een persoon weer, aan de hand van deze gradiënten ziet uw in afbeelding 2.4 het resultaat van een detectie.

Figuur 2.3: Gradiënten van een persoon



Figuur 2.4: Resultaat van een detectie

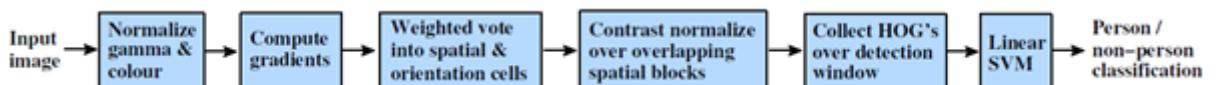


2.2 Inleiding

HOG is de afkorting voor Histograms of Oriented Gradients. Het algoritme herkent een voorwerp, in dit geval een persoon, door het voorwerp te generaliseren door middel van zijn contouren. Deze vorm wordt dan vergeleken met gelijkaardige voorwerpen. Om mensen te herkennen aan de hand van zo een generalisatie moet een SVM (Support Vector Machine) getraind worden. Dit gebeurt aan de hand van vele voorbeelden met foto's met en zonder personen.

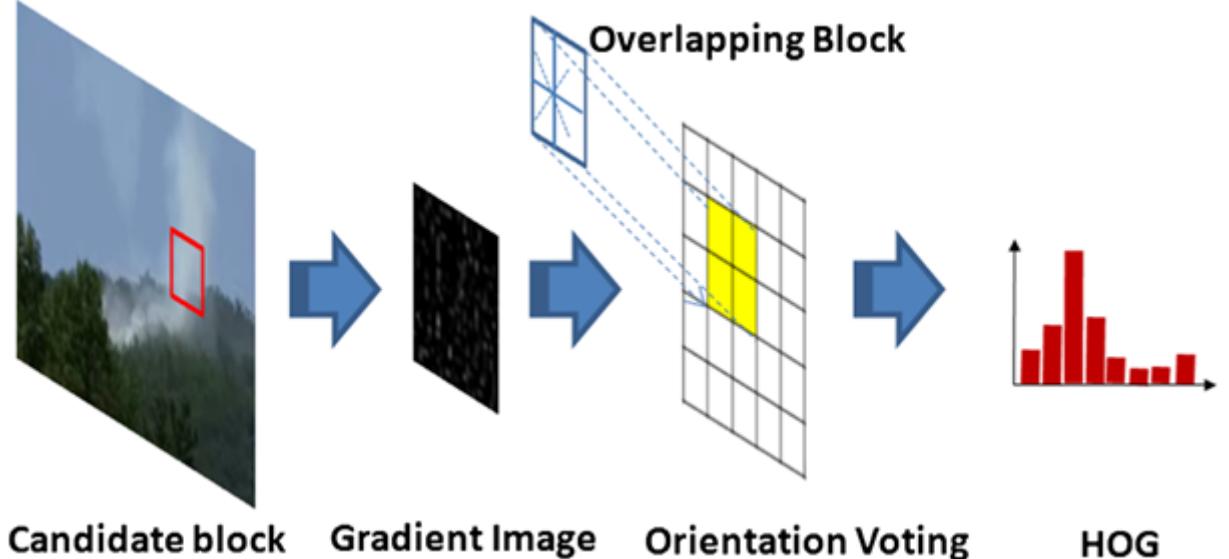
De HOG detectie werkt aan de hand van een sliding window dat over de afbeelding beweegt. Op elke positie van dit window wordt de HOG gradiënt berekend. In een cel nemen we een aantal gradiënten samen om van deze de resultante te berekenen. Deze cellen worden vervolgens in blokken ingedeeld om het geheel robuuster te maken tegen contrastschommelingen. Al deze resulterende vectoren worden daarna naar de SVM gestuurd, deze bepaalt dan of er een persoon in de afbeelding staat of niet.

Figuur 2.5: Blokschema HOG



Het blokschema dat Dalal en Triggs in hun paper [3] gebruiken is weergegeven in figuur 2.5. Om figuur 2.5 nog te verduidelijken maken we gebruik van figuur 2.6. Deze geeft op een afbeelding de stappen visueel weer. In de figuur ziet u links dat de positie van het sliding window is weergegeven door een rode kader. Van dit window ziet u in de volgende figuur de gradiëntbepaling. De volgende stap is om de figuur op te delen in verschillende blokken, zoals u kunt zien in figuur 2.6 onder *overlapping block*. Hierop is ook al te zien dat we voor de contrastnormalisatie vier blokken samen nemen. Van deze vier blokken wordt dan in de rechtse afbeelding het histogram berekend. Dit histogram beschrijft dan onder welke hoek de gradiënten het meest georiënteerd liggen. Aan de hand van dit histogram kan vervolgens de SVM een bepaling doen of er al dan niet een persoon op de afbeelding aanwezig is. Het blok van de SVM is echter niet weergegeven in figuur 2.6.

Figuur 2.6: Blokschema van HOG descriptor [9]



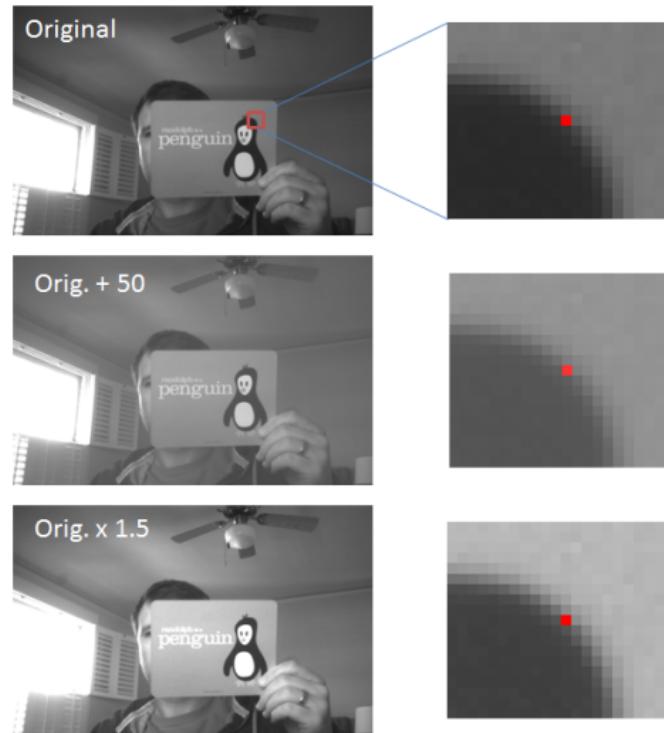
Dalal en Triggs gaven aan dat het beste resultaat wordt bekomen bij maximale kwaliteit van de afbeelding. Na *blurren* van het inputbeeld zullen randen vervagen waardoor de kracht van het algoritme zal afnemen. In de paper van Dalal en Triggs werd er maar op één schaal gewerkt. Dit zorgde ervoor dat het cel formaat 8x8 pixels overeenstemde met de ledematen van de personen op de testbeelden en daarom voor de beste resultaten zorgde. In [14] wordt een manier besproken die wel met variable celformaten werkt.

2.3 Kleurnormalisatie

Hierboven hebben we de werking van het algoritme algemeen besproken. Vanaf nu gaan we elk deel van het algoritme apart uitdiepen. De eerste stap in het algoritme is de kleurnormalisatie van de afbeelding. In deze stap zullen we een vaste waarde met de pixelwaarden vermenigvuldigen. Deze vermenigvuldiging zal voor een contrast verbetering zorgen waardoor de gradiënt vectoren aan de rand van een object groter worden.

Het effect van normalisatie wordt duidelijk aan de hand van figuur 2.7. De eerste foto is de originele afbeelding. Er is, in de rode kader, één pixel aangeduid in het close-up beeld. De tweede figuur geeft de situatie weer na een sommatie van de pixelwaarden met 50. Hierdoor wordt de afbeelding lichter. Dit zal niet helpen in ons algoritme omdat er geen absolute verandering is met de originele afbeelding. Op de laatste afbeelding zijn al de pixelwaarden vermenigvuldigd met 1,5. Op deze laatste zien we dat vermenigvuldigen wel een groot effect heeft op de helderheid van een afbeelding. De lichte kleuren worden veel helderder, de donkere kleuren wijzigen slechts weinig. Hierdoor vergroot het contrast van de afbeelding. Dit grotere verschil tussen donker en licht zal het nadien eenvoudiger maken om de gradiënt te beoordelen.

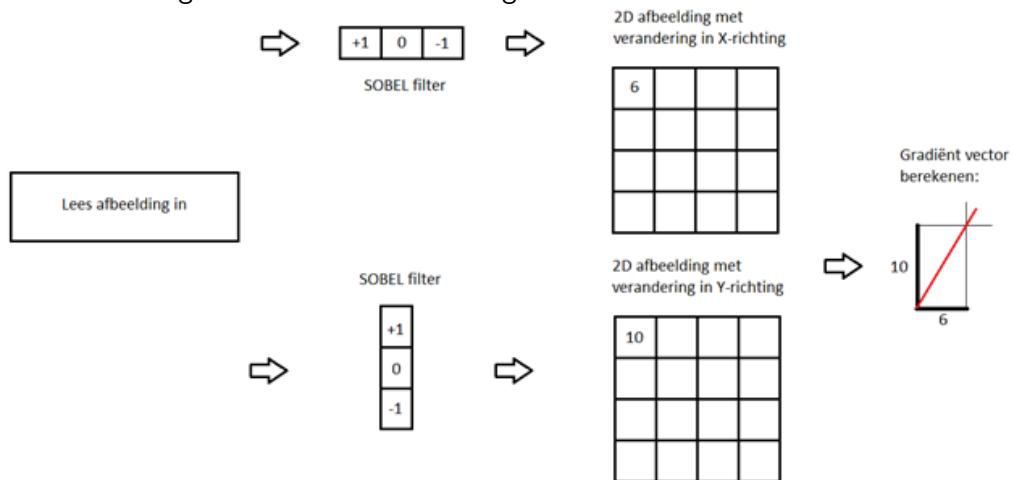
Figuur 2.7: Het effect van normalisatie [9]



2.4 Gradiënt vectoren berekenen

Het berekenen van de gradiënt vector zal uitgelegd worden aan de hand van figuur 2.8.

Figuur 2.8: Blokschema om gradiënt vectoren te berekenen

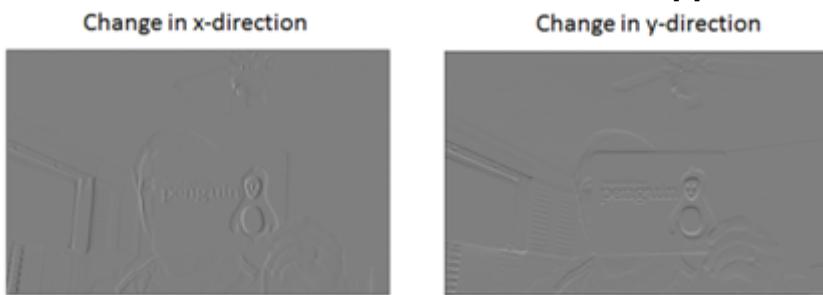


Als input wordt de, al dan niet genormaliseerde, afbeelding ingeladen. Op deze afbeelding wordt dan een sobel filter toegepast zowel in x- als in y-richting. Deze filter geeft als resultaat de afgeleiden in de x- en de y-richting. Het effect van deze filtering wordt weergeven door

figuur 2.9.

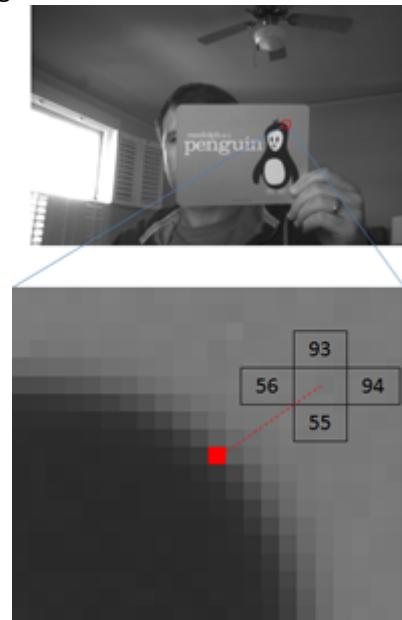
Beide resultaten van het filter worden in afbeelding 2.8 voorgesteld als een matrix met een bepaalde pixelwaarde. Van elke overeenkomstige pixel van beide figuren wordt vervolgens de gradiënt vector bepaald. Zowel de magnitude als de hoek bepalen de vector. Dit gebeurt door middel van de stelling van Pythagoras toe te passen op de beide waarden en de boogtangens van de waarden te berekenen. Vervolgens nemen we 8x8 pixels samen in één cel. Al de gradiënt vectoren in deze cel worden opgeteld tot één resulterende vector.

Figuur 2.9: Het effect van het sobel filter [9]



Om een beter beeld te krijgen van hoe een gradiënt vector wordt berekend gebruiken we een voorbeeld. Het berekenen van de vector gradiënt is eenvoudig. Een vector gradiënt is een vector die de verandering van de pixelwaarde in de x- en y-richting rondom de pixel weergeeft. In dit voorbeeld berekenen we de vector gradiënt van de pixel die rood omkaderd is in figuur 2.10.

Figuur 2.10: Illustratie van de blocks [9]



De foto bestaat uit grijswaarde. De pixelwaarden gaan van 0 (zwart) tot 255 (wit). De pixels links en rechts van de rode pixel zijn 56 en 94. Om de vector component in de x-richting

te vinden nemen we eenvoudig het verschil van deze twee. We bekomen dus een waarde van 38 positief in de x-richting. Dit herhalen we voor de y-richting zodat we, $93 - 55$, 38 in de y-richting bekomen. Deze twee waardes geven ons de gradiënt vector.

$$\begin{bmatrix} 38 \\ 38 \end{bmatrix}$$

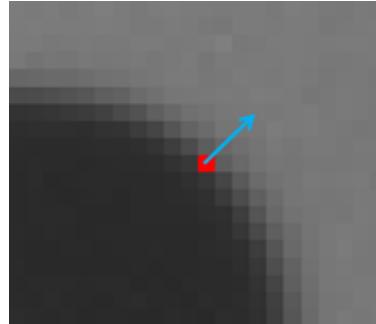
Deze gradiënt vector kunnen we vervolgens ook als volgt voorstellen:

$$\text{Magnitude} = \sqrt{38^2 + 38^2}$$

$$\text{Hoek} = \arctan\left(\frac{38}{38}\right) = 45^\circ$$

Als we deze vector opnieuw op de afbeelding plaatsen, merken we op dat de vector de omlijning van het hoofd van de pinguïn aangeeft in figuur 2.11.

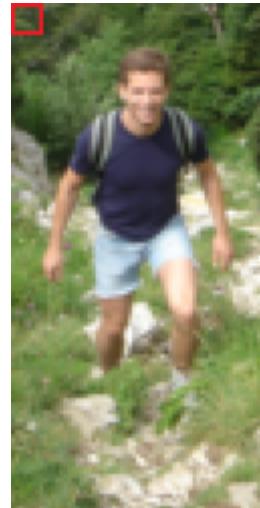
Figuur 2.11: De berekende vector gradiënt op de voorbeeldafbeelding [9]



Deze bewerking gaan we nu uitvoeren op de totale afbeelding. Merk op dat de pixelwaarde nu kan variëren van -255 tot 255. Omdat dit niet in op een afbeelding kan weergegeven worden wordt dit omgevormd naar een bereik van 0-255. Hierdoor worden pixels met een grote negatieve verandering zwart en met een grote positieve verandering wit. Alle pixels zonder grote intensiteitovergangen worden grijs.

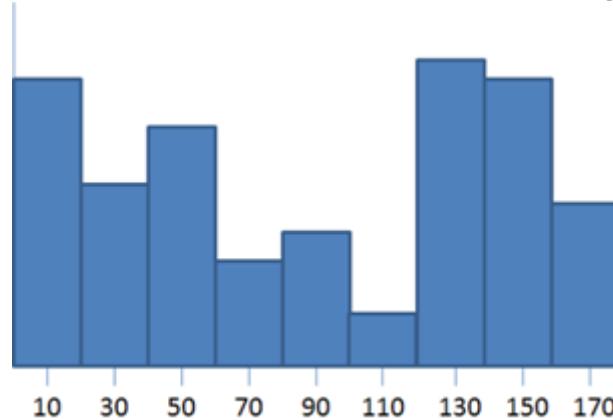
2.5 Gradiënt histogrammen

Nadat we de vector gradiënten hebben bepaald, gaan we het gradiënt histogram invullen. Dit gebeurt aan de hand van een cel van 8x8 pixels. Deze cel bevindt zich in een blok van meerdere cellen. De totale afbeelding is ingedeeld in blocks waarbij deze elkaar 50% overlappen.

Figuur 2.12: Voorbeeld van de grootte van een 8×8 cel [9]

Binnen in deze (rode) cel berekenen we eerst de gradiënt vectoren. In hoofdstuk 2.4 werd beschreven hoe zo een gradiënt berekend wordt. We nemen onze 64 gradiënt vectoren (8×8) en plaatsen ze in een histogram met 9-bins verdelingen tussen de 0° en de 180° , met een tussenverdeling van 20° .

Figuur 2.13: Voorbeeld van een 9-bin histogram [9]



De verdeling van de gradiëntwaarden in het histogram wordt bepaald door zowel de grootte als de hoek van de vector. De waarde van de magnitude wordt als een gewogen gemiddelde verdeeld over de bins. Hoe dichter de hoek aanleunt bij een waarde op de x-as, hoe groter de waarde die we optellen bij deze bin. Als we bijvoorbeeld een vector hebben met waarde 1 onder een hoek van 85° dan voegen we $1/4$ toe aan de balk van de 70° en $3/4$ aan de balk van de 90° .

2.6 Bloknormalisatie

De blokken die gebruikt worden door Dalal en Triggs bestaan uit 2x2 cellen. De blokken overlappen elkaar 50%. De opbouw wordt duidelijk door onderstaande figuur 2.14.

Figuur 2.14: Illustratie van de blocks [9]

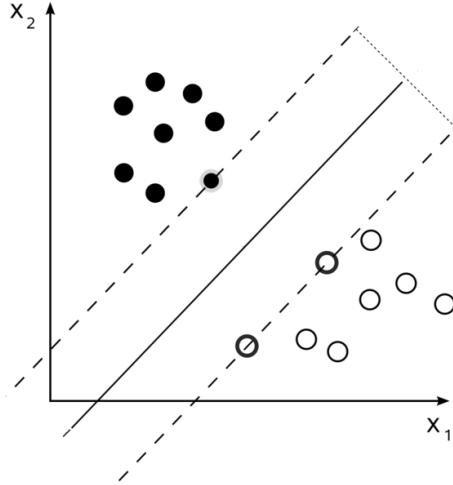


De bloknormalisatie gebeurt door de histogrammen van de vier cellen te concateneren in een vector met 36 componenten (4 histogrammen x 9 bins/histogram). Deze vector wordt dan gedeeld door zijn magnitude om hem te normaliseren.

2.7 Lineaire SVM

De lineaire SVM (support vector machine) is een lerend algoritme dat patronen tracht te herkennen in een ingevoerde dataset. In dit geval zullen we de SVM een aantal afbeeldingen geven met en zonder personen erop. Bij elke afbeelding geven we ook mee of er zich al dan niet een persoon op de afbeelding bevindt. Aan de hand van deze voorbeelden zal het algoritme zich dan een beeld proberen te vormen over hoe hij afbeeldingen met en zonder personen kan onderscheiden, door een optimale scheiding te vinden tussen beide klassen. Omdat we gebruik maken van een Lineaire SVM zal deze scheidingslijn een lineair vlak zijn. Het gewenste resultaat is dat we een figuur bekomen zoals in figuur 2.15, waarbij één puntenwolk de afbeeldingen met personen en de andere wolk de afbeeldingen zonder personen aangeeft. Hiertussen kan dan een scheidingslijn getrokken worden. Wanneer er nu een nieuwe afbeelding binnenkomt zal het algoritme kijken bij welke puntenwolk deze het dichtste aanleunt. Op basis hiervan wordt dan beslist of er al dan niet een persoon op de afbeelding staat. Merk ook op dat er een grijze zone is in deze beslissing. Wanneer er zich een punt tussen de stippellijnen bevindt dan is de keuze niet voor de hand liggend. Daarom is het de bedoeling dat er voldoende en goede voorbeeldafbeeldingen worden gebruikt om het algoritme te trainen. Hoe beter de voorbeelden, hoe beter de SVM een onderscheid zal kunnen maken, hoe minder foute beslissingen hij zal nemen.

Figuur 2.15: Voorbeeld van een getrainde SVM



2.8 Onderzoeken rond het HOG algoritme

We zijn ons onderzoek begonnen met het bestuderen van de paper van Dalal en Triggs [3]. Op deze paper wordt immers mijn thesis gebaseerd. Nadien zijn we op zoek gegaan of er sindsdien ook nog andere studies zijn gebeurd rond dit algoritme. Zo vonden we dat J. Brookshire et al. [7] ook al een Hardware-software implementatie hadden verwezenlijkt. Zij hebben enkel het *window processing* deel in hardware geïmplementeerd. C. Bourrasset et al. [2] hebben onderzocht hoe je het gradiënt berekenend deel, het histogram berekenend deel en de normalisatie kunt implementeren. Zij losten de moeilijke berekening van de boogtangens en de vierkantswortel op door middel van LUTs. Ook splitste hij deze berekeningen op, om ze zo parallel te kunnen uitvoeren. LUO hai-bo [6] ging op zoek naar een manier om de berekening van de boogtangens en de vierkantswortel te gaan omzeilen. Deze zijn immers zeer rekenintensief voor een FPGA. Hij stelt functies voor die sneller uitvoerbaar zijn en beide berekeningen benaderen. V. Prisacariu heeft [10] een versnelling realiseerde door een implementatie op een GPU. De paper van Q. Zhu [14] stelde een aanpassing van het algoritme voor om de snelheid te verbeteren. Hij verkoos om de parameters van het algoritme te variëren om zo eerst personen die groot afgebeeld zijn te vinden en daarna pas kleinere personen te gaan detecteren.

3

Hardwareplatform

Na een bespreking over het HOG algoritme gaan we in dit hoofdstuk het gebruikte hardware-platform bespreken. Het ontwikkelbord dat we gebruikten voor dit project is het Zedboard van Xilinx. Ook zullen we de gebruikelijke ontwerpprocedure voor dit ontwikkelbord toelichten.

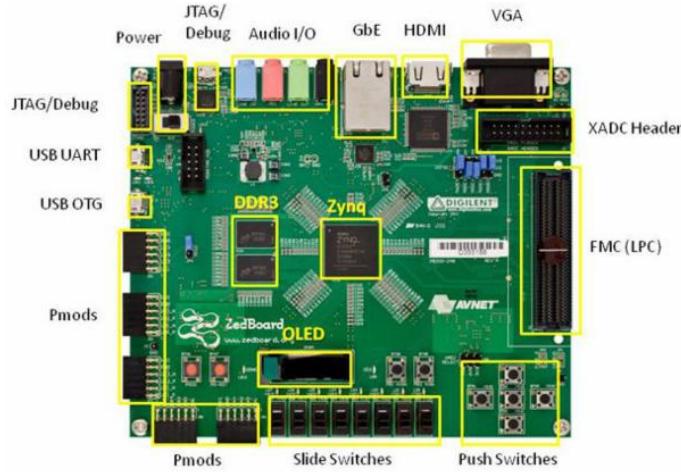
3.1 Zedboard

Na de studie over het algoritme, zijn we op zoek gegaan naar een evaluatiebord waarop we onze implementatie konden verwezenlijken. Omdat er software van het HOG algoritme beschikbaar is, hebben we besloten om niet voor een traditioneel FPGA bord te kiezen maar voor het Zedboard (Zynq Evaluation and Development Board). Het Zedboard is een nieuw evaluatiebord rond de Xilinx Zynq SoC (system on chip). Dit maakt het mogelijk om een gedeelte van het algoritme softwarematig te gaan implementeren op de ARM processor en een ander deel op de FPGA die de Zynq bevat. Hierdoor is het mogelijk om een software-hardware implementatie te verwezenlijken met één component. De code die beschikbaar is kunnen we nu laten runnen op de processor om ze vervolgens stap per stap om te gaan vormen naar VHDL code.

Een andere mogelijkheid was de SoCKIT. Dit is een ontwikkelbord rond de Cyclone V SoC van Altera. Dit bord bevat gelijkaardige *features* als het Zedboard voor een vergelijkbare prijs. Een voordeel van de SoCKIT is dat hij meer RAM geheugen bevat. Het argument dat de doorslag gaf om toch voor het Zedboard te kiezen was de uitgebreide hoeveelheid informatie en handleidingen die beschikbaar zijn voor het Zedboard.

Het Zedboard beschikt over de nodige interfaces die we kunnen gebruiken. Als eerste heeft het een SD-slot. Via een SD-kaart kunnen we afbeeldingen inlezen, om zo onze testen uit te voeren op gekende afbeeldingen. Ook kunnen we via deze een OS (operating system) booten. Verder beschikt het bord met een USB-input, HDMI- en VGA-output over voldoende I/O om ook streaming video mogelijk te maken. Het bord beschikt tevens over voldoende RAM geheugen (512MB) voor onze toepassing. De opbouw van het bord staat weergegeven in figuur 3.1.

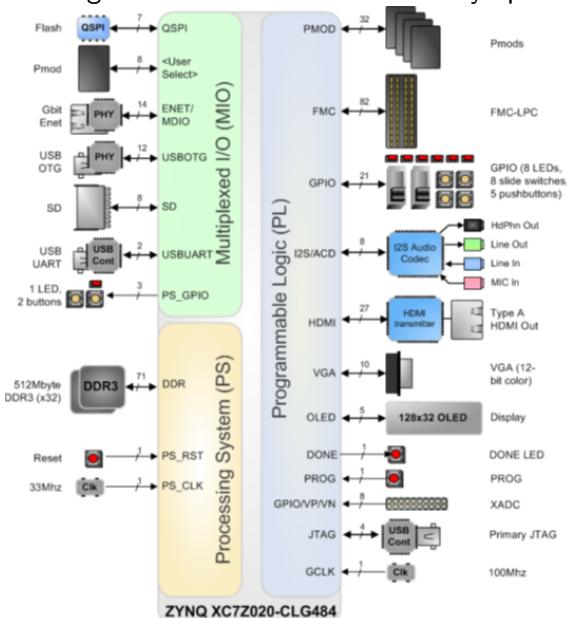
Figuur 3.1: Opstelling Zedboard



3.2 Zynq

Zoals hierboven vermeld, is het Zedboard een evaluatiebord rond de Zynq-7000. De Zynq-7000 is een All Programmable SoC. De Zynq bestaat zowel uit een processor (PS-processing system) als uit een FPGA (PL-programmable logic). De I/O is aanspreekbaar vanuit één van deze twee delen ofwel via de MIO (multiplexed input/output), figuur 3.2 geeft hier meer duidelijkheid over.

Figuur 3.2: Blokschema van de Zynq



Vanuit de PS is het RAM geheugen meteen bereikbaar. Deze is ook verbonden met een resetknop en een klok van 33MHz. De PL is rechtstreeks verbonden met de schakelaars, LEDs en drukknoppen. Ook zijn de extensieconnectoren meteen aanspreekbaar. Verder zijn

van hieruit de HDMI, VGA en audioconnectoren aanspreekbaar. Als laatste is de PL ook nog verbonden met extensieconnectoren en signalisatie-LEDs. De derde mogelijkheid is dat I/O verbonden is met de MIO. Via deze multiplexer bepaalt de gebruiker zelf welke I/O hij wenst te gebruiken. Zo zijn onder andere de ethernetpoort, het SD-slot en de USB-poort verbonden via deze MIO.

3.3 AXI bus

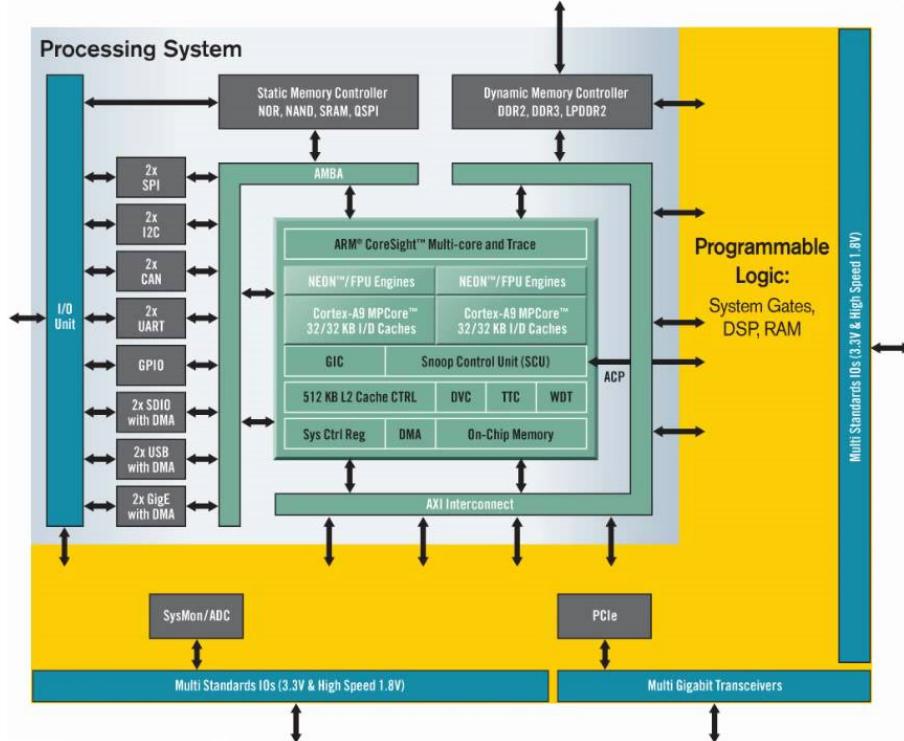
Hierboven werd besproken dat de Zynq zowel uit een PS als PL bestaat en hoe deze verbonden zijn met de I/O. Uiteraard moet de ARM controller ook in contact staan met de FPGA. Voor het datatransport tussen beiden zorgt de AXI bus.

AXI staat voor Advanced eXtensible Interface, het protocol van Xilinx voor IP cores is de opvolger van AMBA. Er bestaan drie types van AXI interfaces:

- **AXI:** De standaardvorm. Deze wordt gebruikt tijdens dit project
- **AXI-Lite:** Een simpele, tragere versie
- **AXI-Stream:** Een versie speciaal voor streaming data tegen hoge snelheden

Figuur 3.3 geeft de interne connectie tussen PS en PL duidelijk weer.

Figuur 3.3: Opbouw van Zynq met AXI bus tussen PS en PL

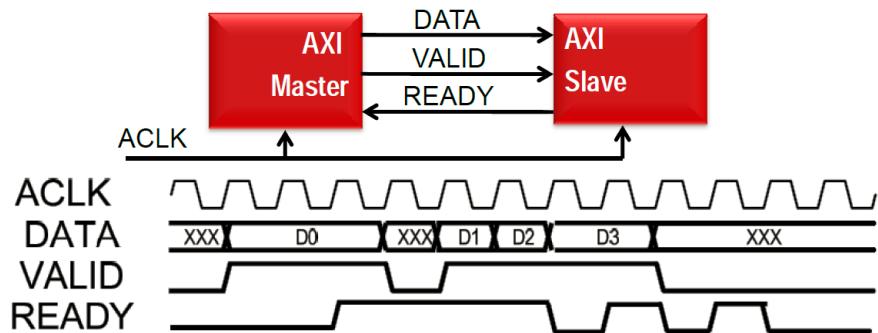


AXI is een *memory mapped interface*, dit wil zeggen dat elke transactie gebaseerd is op een bestemmingsadres binnen het systeemgeheugen. AXI kan *burst* lengtes bevatten tot 256 beats

met een instelbare databreedte van 32 tot 1024 bits. Een burst is een transactie die bestaat uit meer dan één transfer.

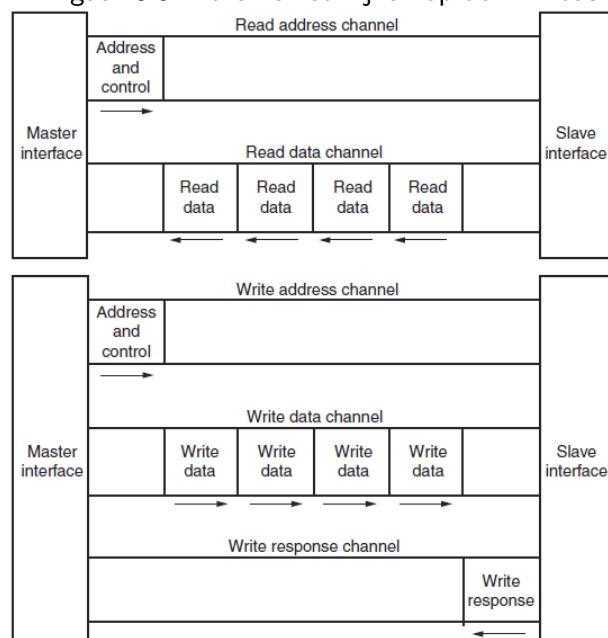
Het *handshake* principe van AXI staat weergegeven in figuur 3.4. De master zal 'VALID' hoog maken en houden zolang er data beschikbaar is. De slave zal 'READY' hoog maken wanneer hij klaar is om data te ontvangen. Zolang zowel 'VALID' als 'READY' beiden hoog zijn, zal er data verzonden worden. Dit handshake principe is geldig voor de drie AXI types.

Figuur 3.4: Handshake principe AXI



De AXI specificatie beschrijft de communicatie tussen een master en een slave. De AXI interface bestaat uit vijf kanalen: *Read- and write address*, *read- and write data channel* en als vijfde een *write response channel*. Het is een full-duplex interface, dus data kan gelijktijdig in beide richtingen getransporteerd worden. Figuur 3.5 geeft weer hoe er gelezen of geschreven kan worden op de AXI bus. Let hierbij op dat er aparte data- en adresverbindingen zijn waardoor full-duplex mogelijk wordt.

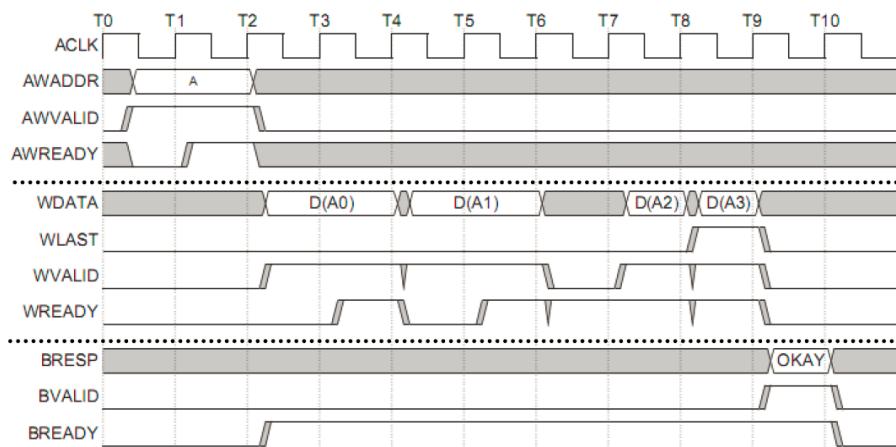
Figuur 3.5: Lezen en schrijven op de AXI bus



Om dit deel over de AXI bus samen te vatten geven we nog een voorbeeld van de timing bij

een schrijfopdracht van 4 bytes. Dit *timing diagram* staat op figuur 3.6. Via AWADDR geven we het adres aan waarnaar we data willen sturen, ook geven we via AWVALID aan dat we klaar zijn voor een transactie. Wanneer ook de slave aangeeft dat hij klaar is met AWREADY kan de transactie beginnen. De data wordt gestuurd over signaal WDATA, bij het versturen van de laatste byte wordt WLAST hoog gemaakt. Zoals hierboven reeds aangehaald kan de transactie pas doorgaan als zowel WVALID als WREADY hoog zijn. Na het versturen van de data wordt de connectie gesloten na het 'OKAY' commando van slave naar master via signaal BRESP.

Figuur 3.6: Timing van een 4 byte write actie over AXI bus



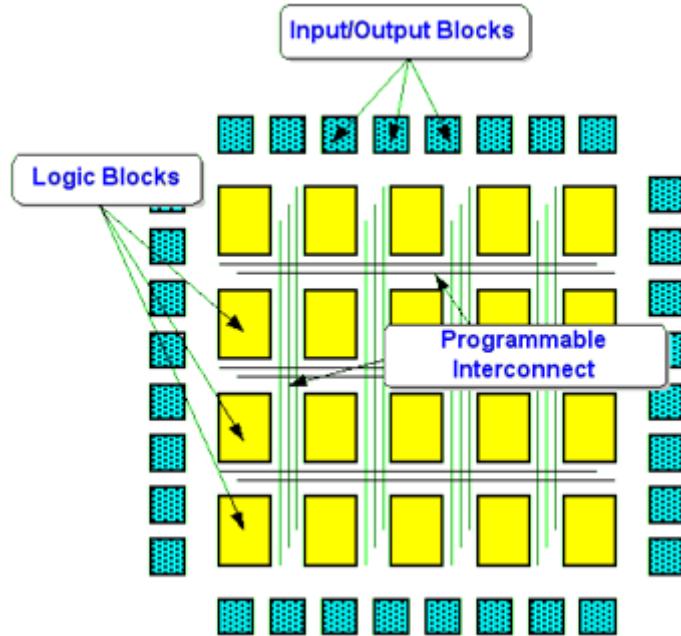
3.4 Technische uitleg over de FPGA

De hardware implementatie van het HOG algoritme gebeurt op de programmeerbare logica van de Zynq SoC. Daarom wordt er in dit hoofdstuk een kleine toelichting gegeven over hoe een FPGA is opgebouwd. Een FPGA is een digitaal component waarvan de gebruiker de werking kan bepalen. Deze werking beschrijft de gebruiker in een hardware beschrijvende taal zoals VHDL of Verilog en kan vervolgens in de FPGA gedownload worden. Een FPGA bestaat uit drie grote delen: Logische blokken, I/O blokken en routering. Deze drie blokken zijn ook duidelijk zichtbaar op figuur 3.7. De gebruiker kan in deze logische blokken zijn functies beschrijven, deze verschillende delen met elkaar verbinden via de routering en vervolgens laten communiceren met de buitenwereld via de I/O blokken. In dit hoofdstuk worden deze drie grote delen verder besproken.

De FPGA die aanwezig is in de Zynq SoC is de Artix-7 van Xilinx. De onderstaande tabel geeft een opsomming van enkele belangrijke kenmerken van de Artix-7.

Programmable Logic Cells	85K
Look-Up Tables	53.200
Flip-Flops	106.400
Extensible Block RAM	560KB
Programmable DSP Slices	220

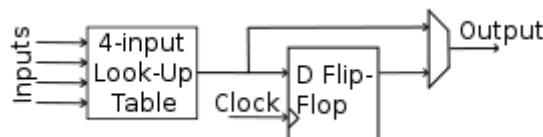
Figuur 3.7: De architectuur van een FPGA



3.4.1 Configurable logic blocks (CLB)

Een FPGA is opgebouwd uit een grote matrix met CLB's. De gebruiker kan iedere CLB naar wens programmeren. Een CLB bestaat uit drie delen: een LUT (Look-up table) met vier inputs, een D flip-flop en een multiplexer. Figuur 3.8 geeft de opbouw van een CLB weer.

Figuur 3.8: De architectuur van een CLB



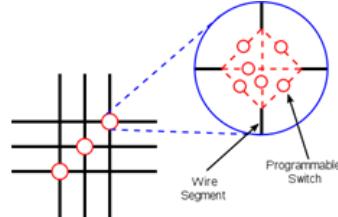
De 4-input LUT's is een geheugen dat kan ingelezen of uitgelezen worden met behulp van deze vier ingangen. De LUT bevat dus zestien geheugenplaatsen. De gebruiker implementeert nu zijn gewenste functies door de overeenstemmende waardes in deze geheugenplaatsen te stockeren. De uitkomst kan via de data flip flop gedurende één klokpuls opgeslagen worden of in één keer naar buiten worden gebracht. Dit wordt bepaald door de MUX.

3.4.2 Routering

De verschillende CLB's kunnen met elkaar verbonden worden via de routeringslijnen. Het bijzondere van de routering van een FPGA is dat ze niet statisch is. Elke route is langs één kant verbonden met een CLB en aan de andere met een schakelblok. Dit houdt in dat de gebruiker via deze schakelblokken CLB's met elkaar moet verbinden. Zo kunnen, door de aaneenschakeling van verschillende CLB's, complexere functies beschreven worden. Figuur 3.9

toont de opbouw van een schakelblok.

Figuur 3.9: Opbouw van een schakelblok



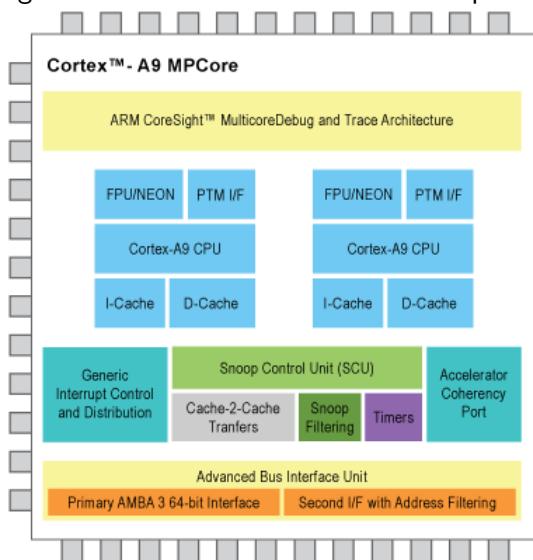
3.4.3 I/O blokken

Een FPGA staat in contact met de buitenwereld via I/O blokken. Als de gebruiker wil gebruik maken van bepaalde I/O blokken zal hij dit moeten instellen in de UCF file (User Constraint File). Met deze file bepaald hij de aansluitpinnen intern in de FPGA.

3.5 ARM processor

Buiten de PL bevindt de Zynq ook nog een processor, de Dual ARM Cortex-A9. Dit is een 32-bit dual core processor met een kloksnelheid tot 1GHz. De ARM is opgebouwd rond de ARMv7-A architectuur, dit is RISC gebaseerde architectuur met twee geïmplementeerde datasets, de 32-bit ARM instruction set en de 16-bit Thumb-2 instruction set. Verder bevat de ARM 37 registers van 32-bits. De processor bevat zowel een AMBA als AXI interface. Het blokschema van de processor vindt u op figuur 3.10.

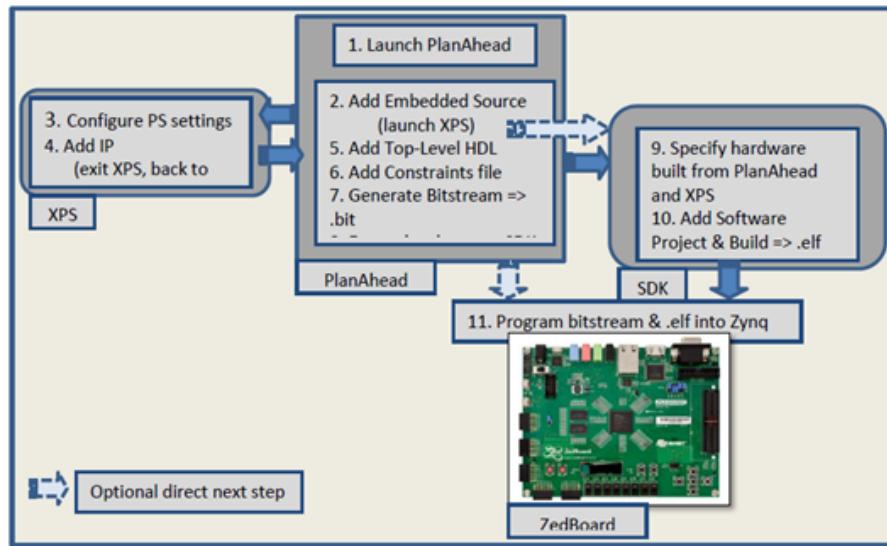
Figuur 3.10: Blokschema van de ARM processor



3.6 Ontwerpprocedure met Xilinx ISE WebPACK

Na de besprekking van het hardwareplatform, leggen we in dit deel de designflow uit die we tijdens ons project moesten volgen om de implementatie te verwezenlijken. Deze ontwerpprocedure staat grafisch weergegeven in figuur 3.11.

Figuur 3.11: Design flow voor Zynq



Eerst beschrijven we een deel van het algoritme in VHDL. Een pakket als *ISE Project Navigator* kan gebruikt worden om de code te schrijven en deze te testen in een testbench. Daarna starten we in *PlanAhead* een nieuw project op. Planahead kan gezien worden als de cockpit van het software pakket. Van hieruit kan de gebruiker de andere software pakketten starten.

Als eerste starten we *XPS* op. XPS (Xilinx Platform Studio) is een ontwikkelruimte waarin de gebruiker zijn hardware kan instellen. Zo kan hij de processor instellen, randapparatuur kiezen en de connecties maken tussen de verschillende hardware blokken. Ook voegen we hier ons zelfgeschreven hardware blok aan ons project toe.

Na de hardware beschrijving, in het XPS pakket, zal er in Planahead hiervan een top HDL kunnen gegenereerd worden. Ook kan hier de bitstream gegenereerd worden. De bitstream is de data die de FPGA zal configureren zoals de gebruiker dat wenst. Deze bitstream kan je vervolgens eenvoudig doorgeven aan *SDK*.

SDK(Software Development Kit) is een programmeeromgeving, gebaseerd op Eclipse. Hierin kan de gebruiker C/C++ software schrijven en vervolgens de .elf (Executable and Linking Format) file genereren, dit is een binair formaat voor uitvoerbare programma's. Deze file kunnen we in de FPGA programmeren. Ook kunnen we in *SDK* onze boot file genereren voor ons OS. Wij maken gebruik van Linaro linux. *SDK* kan door middel van de .bit, .elf en een standaard bootfile van Linaro een eigen bootfile maken. Deze file bevat de eigen geschreven hardware. Deze bootfile kan vervolgens op een SD-kaart geplaatst worden.

De Zynq kan nu, bij juiste positionering van de jumpers, automatisch de eigen bootfile starten bij het aanschakelen van het Zedboard.

4

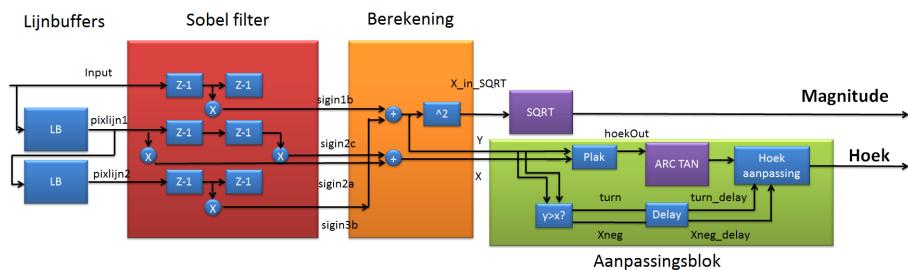
Implementatie van de gradiëntberekening in FPGA

Na de studie over het HOG algoritme in hoofdstuk 2 en de keuze over het hardwareplatform in 3 moesten we beslissen welk deel van het HOG algoritme we eerst gingen implementeren op FPGA. We hebben beslist om het deel dat de gradiënt vectoren berekent eerst te gaan implementeren. Via de profiler *Valgrind* zagen we immers dat 30% van de totale rekentijd aan deze functie werd besteed. Het deel bevat namelijk wiskundige functies die traag verwerkt worden bij softwarematige uitvoering. Ook leent dit deel zich tot implementatie: zo kan de sobelfiltering op een snelle manier gebeuren en het input frame kan opgesplitst worden, om zo delen van de afbeelding gelijktijdig te behandelen. De uitleg over hoe deze vectoren berekend worden, is beschreven in hoofdstuk 2.4. In dit hoofdstuk zullen we onze implementatie die gradiënt vectoren berekent bespreken.

4.1 Blokschema

Om dit probleem aan te pakken hebben we het opgedeeld in verschillende delen. Dit is duidelijk te zien op het blokschema 4.1. In dit hoofdstuk bespreken we onze VHDL code aan de hand van blokschema 4.1 en afbeeldingen uit de testbench. Eerst wordt elk blok van het schema apart toegelicht, hierbij wordt ook gebruik gemaakt van de signaalnamen die op het blokschema beschreven zijn. Op het einde van het hoofdstuk is er een rekenvoorbeeld uitgewerkt.

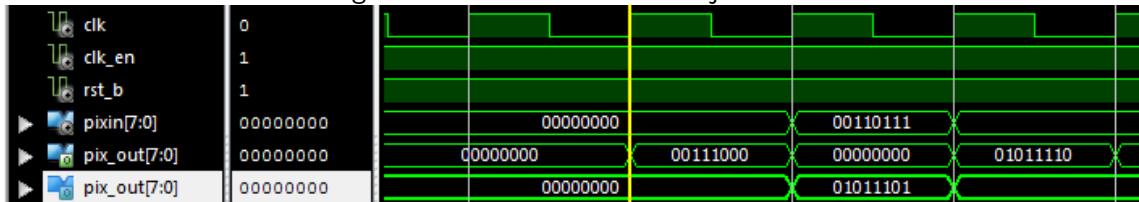
Figuur 4.1: Blokschema van de VHDL code voor de gradiëntberekening



4.1.1 Lijnbuffers

Het eerste deel van deze VHDL code zijn twee lijnbuffers. Dit zijn twee shiftregisters die er voor zorgen dat drie onderliggende pixels gelijktijdig zullen toekomen aan het volgende blok, het sobel filter. Dit is ook te zien op de output van de testbench in figuur 4.2. De twee lijnbuffers hebben een instelbare lengte. Zo kan de gebruiker de lengte van het inkomende frame eenvoudig instellen. Of indien hij het binnengkomende frame zou opsplitsen, kan hij hier de lengte van het deel instellen. In mijn implementatie heb ik gekozen voor een vaste framelengte van 640x480.

Figuur 4.2: Testbench van de lijnbuffers



4.1.2 Sobelfilter

Na de lijnbuffers komen de drie inputpixels toe op het sobelblok. De sobel filter is een beeldverwerkingsalgoritme dat voornamelijk gebruikt wordt bij randdetectie. Het filter neemt de afgeleiden van de pixelwaarde van een afbeelding. Bij weinig kleurverandering zal, net zoals bij de afgeleiden van een constante, het resultaat klein of zelfs nul zijn. Maar bij een grote verandering, zoals bij randen, zal de afgeleide een grote waarde zijn. Omdat de afgeleiden nemen van een afbeelding niet eenvoudig is, wordt deze bewerking gerealiseerd door een convolutie van de totale afbeelding met een klein masker. Hierdoor wordt de rekentijd beperkt. De convolutie wordt tweemaal uitgevoerd, één keer voor de horizontale en één keer voor de verticale afgeleiden. Het eigenlijke sobel masker is van de vorm:

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

In de paper van Dalal en Triggs werd voorgesteld om de convolutie te bepalen met het volgende masker:

$$\begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} +1 \\ 0 \\ -1 \end{bmatrix}$$

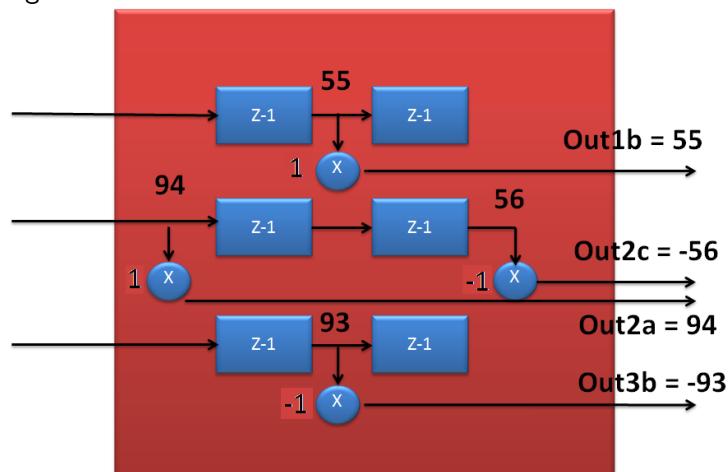
We hebben deze stap van het algoritme ook in Matlab geprogrammeerd om de tussenstappen grafisch te kunnen weergeven. Zo kan je op figuur 4.3 duidelijk het verschil zien tussen het resultaat van de horizontale en de verticale filtering.



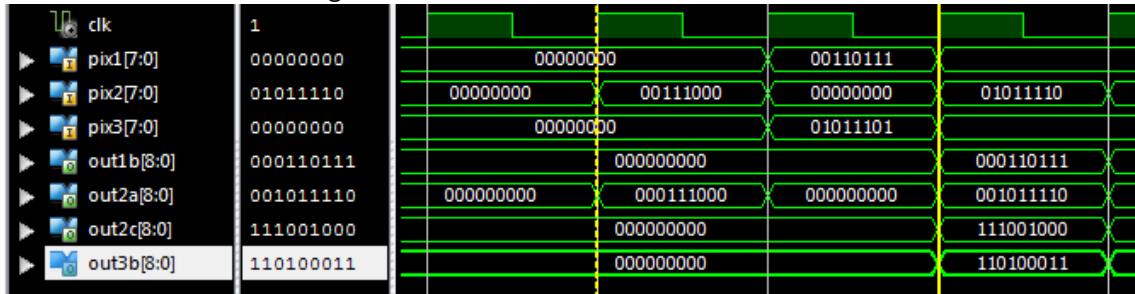
De hardware implementatie van het sobel filter is gebaseerd volgens [8]. Hierin staat een blokschema voor het implementeren van een convolutiefilter in hardware. Het blokschema dat wij gebruiken op basis hiervan staat in figuur 4.4.

Door twee delays per pixel in te voegen wordt een 3×3 masker gecreëerd. Van dit masker wordt er op vier plaatsen afgetakt, twee van deze aftakkingen worden rechtstreeks naar buiten gebracht, de andere worden vermenigvuldigd met -1. Deze vermenigvuldiging met -1 verkrijgen we door het *two's compliment* van de waarde te nemen. Om nog in het bereik van -255 en +255 te blijven zullen de uitgangssignalen van dit blok 9-bit signalen zijn. Dit is ook te zien op de testbench in figuur 4.5. In het voorbeeld van figuur 4.4 worden de waarden '55' en '94' zonder conversie naar outputs *out1b* en *out2a* gekoppeld. De waarden '56' en '93' worden omgevormd naar *two's compliment* en naar *out2c* en *out3b* gebracht.

Figuur 4.4: Blokschema van sobel filter met voorbeeldwaarden



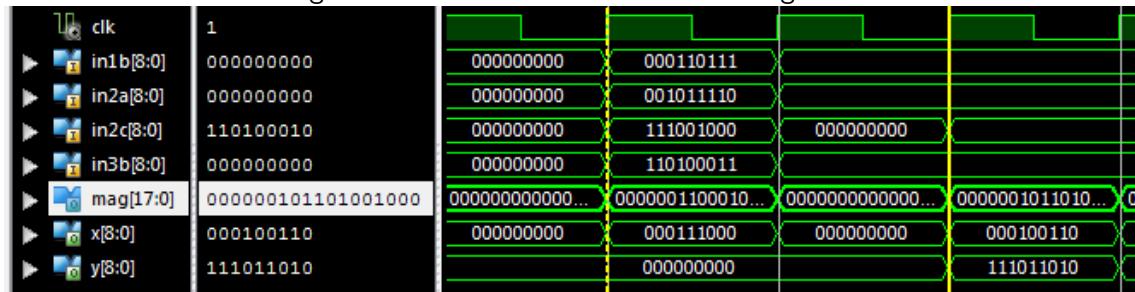
Figuur 4.5: Testbench van het sobel masker



4.1.3 Berekeningsblok

De uitgangen van het sobel masker worden vervolgens omgevormd naar x- en y-vectoren. Dit gebeurt in het berekeningsblok door een simpele optelling. Door de som van de horizontale pixels out2a en out2c bekomen we de x-vector, met out1b en out3b bepalen we de y-vector. Van deze vectoren wordt ook nog de som van hun tweede macht genomen. Deze berekening is de voorbereiding om met de CORDIC core de vierkantswortel te berekenen. De resultaten van deze berekeningen zijn te zien op figuur 4.6.

Figuur 4.6: Testbench van het berekeningsblok



4.1.4 Aanpassingsblok

Wij maken gebruik van twee CORDIC cores in onze VHDL code. Deze cores zullen aan de hand van het CORDIC algoritme, zie hoofdstuk 5, de vierkantswortel en boogtangens bepalen. Het SQRT blok is rechtstreeks aangesloten aan het berekeningsblok, de core voor de boogtangens heeft een aangepaste input nodig. Voor deze aanpassing dient het aanpassingsblok, dit bestaat uit verschillende delen die apart besproken zullen worden.

Grootste bepalen

Het eerste blok in het aanpassingsblok bepaalt welk van de beide vectoren, x of y, het grootste is. Dit is belangrijk omdat het CORDIC core enkel kan werken in het domein tussen $\pm 45^\circ$. Wanneer de absolute waarde van y groter is dan x zou de hoek buiten dit bereik vallen, daarom draaien we de vectoren om. Indien de vectoren omgedraaid werden moet dit onthouden worden,

dit wordt met output 'turn' weergegeven. Ook geeft de output van dit blok weer of de x-vector negatief is. Dit moet ook mee in rekening worden gebracht na de berekening van de CORDIC core.

Concatenate

In dit blok, "plak" genaamd, worden de twee vectoren aan elkaar geplakt. De CORDIC core kan echter enkel werken met één lange inputvector. De core verwacht dat de y-vector vooraan staat (MSB). Ook zal dit blok de voorstelling van de inkomende signalen veranderen. Er komen waarden toe tussen ± 255 . Het CORDIC blok kan immers enkel werken met inputs met het 1QN formaat tussen ± 1 . Bij deze voorstelling staat er een komma na de tweede bit. Hierdoor is waarde "01.000000" gelijk aan +1 en waarde "11.000000" gelijk aan -1. Deze omvorming hebben we op een efficiënte wijze toegepast. Eerst gaan we na of een cijfer positief of negatief is, vervolgens sommeren we het getal met +1 om daarna te delen door vier. Deze deling voeren we uit door tweemaal naar rechts te shiften. Op deze wijze vallen de inputwaarden steeds binnen de gewenste grenzen.

Hoekbepaling

Nadat de CORDIC core zijn berekening gedaan heeft moeten nog enkele wijzigingen gebeuren. Er moet rekening worden gehouden met welke vector de grootste is en of de x-vector negatief is. Afhankelijk hiervan moeten er correcties worden toegepast.

- **turn = '1' en x = negatief**

Wanneer dit geval optreedt moet de uitkomst van het CORDIC blok, als volgt worden aangepast: $uit = -90^\circ - \text{hoekIn}$

- **turn = '1' en x = positief**

In dit geval is de aanpassing als volgt: $uit = 90^\circ - \text{hoekIn}$

- **andere gevallen**

In alle andere gevallen wordt de berekening van het CORDIC blok gewoon doorgelust naar de output zonder aanpassing.

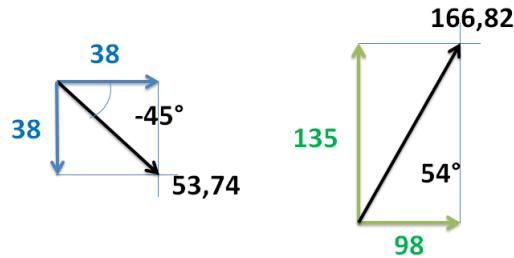
4.1.5 Voorbeeld

Om dit hoofdstuk samen te vatten werken we een voorbeeld uit om het geheel duidelijk te maken. Het signaal dat we aanleggen in het voorbeeld is weergegeven in figuur 4.7. In figuur 4.8 staat dan de gewenste uitkomst van zowel de blauwe als de groene vectoren.

Figuur 4.7: Voorbeeld van input testbench

		55	190		
	94	30	56	128	
		93	55		

Figuur 4.8: Berekende uitkomst van testbench

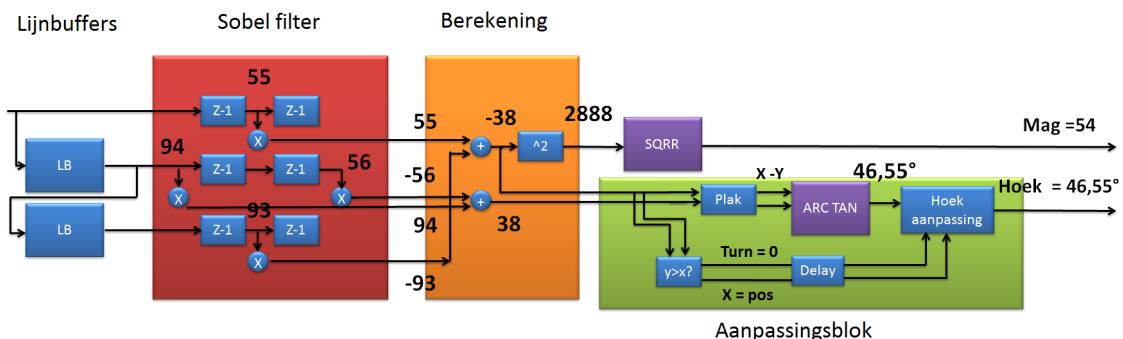


De pixels die eerst binnengaan zijn waarden '93' en '55'. Deze worden dan vertraagd in de lijnbuffers zodat de onderliggende pixels, op de inputafbeelding, ook op deze manier toekomen op het sobel filter. Op figuur 4.9 wordt de situatie weergegeven wanneer de blauwe pixelwaarde in het sobel filter toekomen.

Na het toepassen van het filter kunnen de x-en y-vector bepaald worden. In dit geval zijn de twee vectoren: $x = 38$ en $y = -38$. Ook wordt de som van de kwadraten bepaald als voorbereiding op de SQRT core.

In het aanpassingsblok gebeuren bewerkingen om de CORDIC core correct te laten werken. Omdat de absolute waarde van y niet groter is dan die van x en x positief is moeten er geen aanpassingen gebeuren.

Figuur 4.9: Blokschema met voorbeeldwaarden



Als eindresultaat bekomen we een magnitude van 54 (binair = "00110110"), deze waarde ligt zeer dicht bij de uitgerekende waarde van 53,74. Als hoek verkrijgen we de binaire code "111.00101", wat gelijk is aan een hoek van 46.55 graden. Dit verschil van 1.55 graden met de theoretisch correcte waarde valt binnen de theoretische grenzen die aangegeven worden in hoofdstuk 5.3.

Dit schema levert pas nuttige uitkomsten wanneer er drie rijen pixels zijn ingeladen. Tijdens het inlezen van de eerste twee rijen moet het resultaat niet worden onthouden. Ook kan het 3x3 masker de bovenste rand niet meer bepalen. De vertraging van het totale schema, of latency, bedraagt 15 klokpulsen. Als laatste hebben we de totale rekentijd voor een 640x480 frame bepaald: Eerst is er een opstartprocedure van $640 * 2$ klokpulsen. Daarna volgt de gradiëntberekening gedurende $(480 - 2) * 640$ klokpulsen. In totaal zorgt dit voor 307200 klokpulsen, bij een kloksnelheid van bijvoorbeeld 100MHz levert dit een snelheid op

van $307200 * 10\text{ns} = 3.072\text{ms}$. Uiteraard neemt het datatransfer tussen het RAM geheugen en de FIFO's ook nog tijd in beslag. Aan de hand van een voorbeeldtransactie uit [13] en bijlage 1, hebben we deze tijd berekend. Hier nam een 32-bit transactie van 16 bursts 600ns in beslag, de totale transfertijd bij deze businstelling wordt dan:

om FIFO 1 keer te vullen zijn er 32 bursts nodig
 $600 * 32 = 22400$

De input FIFO moet 150 keer gevuld worden en de output ongeveer 600 keer
 $(150 * 22400) + (600 * 22400) = 14.800.000\text{ns}$
 $= 14.4 \text{ ms}$

Tel hierbij nog de rekentijd van de berekening zelf, dan bekomen we een totale *processing time* van 17.5ms.

Daarna hebben we de C++ code van OpenCV 50 keer uitgevoerd, op een *Dell Latitude E6500 met Dual core @2.4GHz met een Ubuntu Linux OS*, om de totale rekentijd te monitoren. Na het elimineren van de hoogste rekentijden hebben we de gemiddelde rekentijd genomen, deze bedroeg 409ms. Omdat de profiler *Valgrid* aangaf dat de gradiëntberekening 30% van de totale tijd in beslag nam, komt dit neer op een tijd van 120ms. Het verwerkte frame was echter een kleurbeeld, dus deze waarde dient nog gedeeld te worden door 3. Zo kunnen we besluiten dat de softwarematige uitvoering, op de gebruikte PC, 40ms in beslag neemt.

Hieruit kunnen we besluiten dat we ten opzichte van een softwarematige uitvoering op een dual core PC, een snelhedsverbetering van ongeveer 50% kunnen verkrijgen voor het bepalen van de gradiënt vectoren.

5

CORDIC algoritme

In het vorige hoofdstuk werd uitgelegd hoe de VHDL code die de gradiënt vector bepaalt opgebouwd is. Omdat in deze code gebruik werd gemaakt van het CORDIC algoritme zal er in dit hoofdstuk dieper ingegaan worden op de werking van dit algoritme. CORDIC staat voor COordinate Rotation for DIgital Computer. Dit algoritme werd ontwikkeld door Volder [11] en biedt oplossingen voor de volgende problemen:

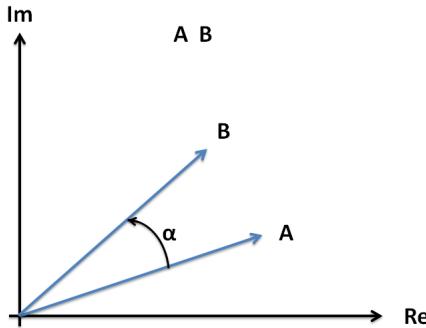
- trigonometrische functies
- vectorrotaties (cathesis $< - >$ polair)
- conversie tussen vector voorstellingen

Dit doet het ook met een minimale hoeveelheid aan hardware en met oog op een snelle implementatie.

5.1 Werking van CORDIC

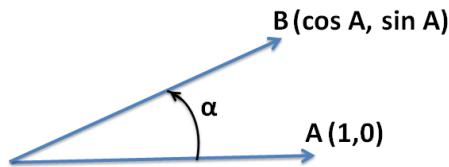
De werking van het algoritme zullen we uitleggen aan de hand van figuur 5.1. Het CORDIC algoritme voert een planaire rotatie uit. Dit betekent dat vector A zijn coëfficiënten zal aanpassen door hem over een hoek α te verdraaien en zo vector B te bekomen. Het algoritme heeft twee verschillende werkmethodes: rotatie mode en vector mode.

Figuur 5.1: Planaire draaiing bij CORDIC



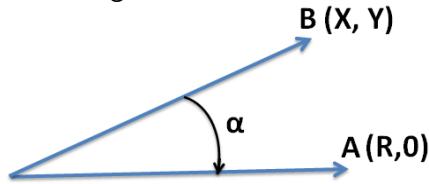
Bij de rotatie mode is vector A een eenheidsvector, dit wil zeggen dat hij op de x-as ligt. De verdraaiing richting vector B wordt nu benaderd door een opeenvolging van steeds kleiner wordende vaste hoeken. Eenmaal vector B bereikt is kan hij gebruikt worden om de cosinus en de sinus van de hoek α te vinden. Dit is weergeven op figuur 5.2.

Figuur 5.2: Rotatie mode



De tweede methode is de vector mode. Deze zal voor een gegeven vector A de hoek α en de magnitude van deze vector bepalen. Dit wordt bepaald door een vectorrotatie richting de x-as met een opeenvolging van steeds kleiner wordende vaste hoeken. Het basisprincipe wordt duidelijk met figuur 5.3.

Figuur 5.3: Vector mode

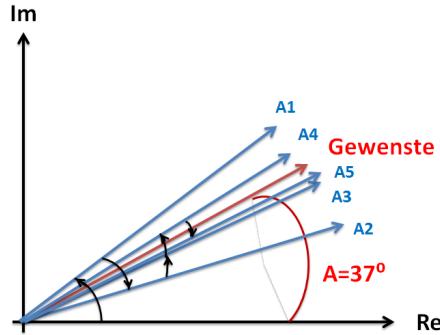


Dit algoritme werd later nog uitgebreid door Walther [12] zodat ook een vierkantwortel en boogtangens bepaald kunnen worden. Deze uitbreiding wordt gebruikt tijdens dit project.

5.2 Voorbeeld van het CORDIC algoritme

Een verdere verklaring van het algoritme geven we aan de hand van een voorbeeld. Op figuur 5.4 is een uitwerking gegeven van de manier waarop CORDIC naar een hoek toewerkt. De werking is vergelijkbaar met die van een binaire zoekopdracht.

Figuur 5.4: Werking CORDIC



In dit voorbeeld gaan we toewerken naar een hoek van 37° . De eerste rotatie die het algoritme uitvoert is een draaiing over 45° tegenwijzerzin. Vervolgens wordt de resterende hoek bepaald, wat in dit geval ($37-45$) gelijk is aan -8° . Aan de hand van het teken van deze resterende hoek ziet het algoritme dat het de volgende verdraaiing in tegengestelde richting (in wijzerzin) moet doen. Deze verdraaiing is er één met een grootte van 26.565° . Dit zorgt voor een resterende hoek van $(26.565-8) + 18.565^\circ$. Bemerkt dat nu het teken van de resterende hoek positief is zodat de volgende verdraaiing in tegenwijzerzin moet gebeuren. Met steeds kleiner wordende vaste hoeken werkt het algoritme toe naar de gewenste hoek van 37° . De hoeken waarmee we telkens verdraaien zijn vast bepaald en worden gegeven in figuur 5.5.

Merk de grote gelijkenis op tussen CORDIC en een binaire zoekopdracht waarbij elke stap een stap dichter is bij de eindoplossing. Het is dus ook begrijpelijk dat de nauwkeurigheid van het CORDIC algoritme afhangt van het aantal iteraties. In het ideale geval zal de gewenste hoek exact bereikt worden, praktisch zal dit nooit haalbaar zijn.

De methode uitgewerkt in figuur 5.4 is de rotatie methode. De vector methode werkt op dezelfde manier, enkel wordt hierbij naar de x-as toegewerkt vertrekende vanaf een bepaalde vector. Elke iteratie richting de x-as is een niet orthogonale rotatie, hierdoor wordt de vector telkens iets groter met een factor $\sqrt{1 + \delta^2}$. Deze factor wordt de *stretch factor* genoemd. Na de uitwerking van de vector methode kan de magnitude van de vector bepaald worden door de verkregen x-waarde te delen door de cummulatieve stretch factor. De boogtangens wordt verkregen door de som van de verdraaiingen.

5.3 Fouten die optreden bij CORDIC

Omdat CORDIC een iteratief algoritme is, is het vanzelfsprekend dat het aantal iteraties belang heeft voor de nauwkeurigheid van het resultaat. Hoe meer iteraties er worden toegepast, hoe meer steeds kleinere wordende stappen het gewenste resultaat kunnen benaderen. In tabel 5.5 wordt de fout in graden uitgedrukt in functie van het aantal iteraties. In onze VHDL code hebben we een output van één byte waarvan één tekenbit en vijf cijfers na de komma. Hierdoor blijkt dat we een nauwkeurigheid van 1.7899° kunnen bereiken zoals aangegeven in figuur 5.5. Dit blijkt ook uit het testprogramma uit hoofdstuk 4.1.5. In dit voorbeeld was de gewenste hoek 45° . Deze hoek van 45° is bewust gekozen, het is immers de eerste van de vaste hoeken in het algoritme. Tijdens de eerste iteratie wordt al meteen geroteerd over 45° zodat de resterende hoek nul wordt, toch doet het algoritme verder. Bij de volgende stap zal hij ofwel plus of min 26.565° doen, de volgende iteraties zal hij steeds roteren in dezelfde

richting om zo dicht mogelijk bij 45° aan te sluiten. Er blijft echter nog een fout van 1.55° over, dit leunt dicht aan bij de voorspelde fout van 1.7899° .

Figuur 5.5: Vector mode

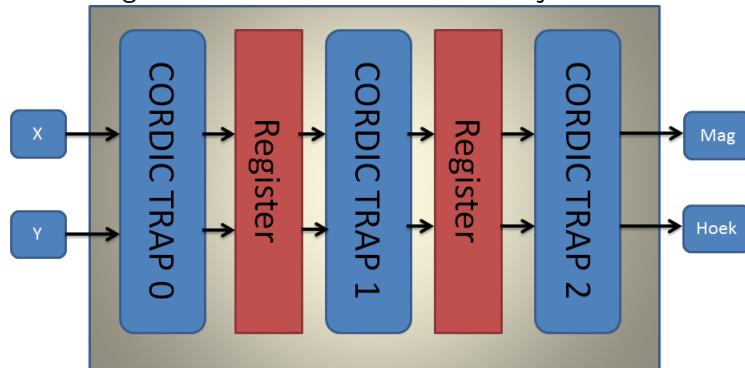
n	$1/2^n$	$C(n)[^\circ] = (360^\circ/2\pi)C_n$
0	1	45.0000°
1	$1/2$	26.5650°
2	$1/4$	14.0362°
3	$1/8$	7.1250°
4	$1/16$	3.5763°
5	$1/32$	1.7899°
6	$1/64$	0.8952°
7	$1/128$	0.4476°

5.4 Gebruik van CORDIC in VHDL

Om het CORDIC algoritme in ons ISE project in te voegen hebben we gebruik gemaakt van IP Core Generator van Xilinx. Op deze manier kunnen moeilijke berekeningen, zoals boogtangens, toch hardwarematig geïmplementeerd worden. Deze cores hebben echter soms aanpassings-blokken nodig. Deze dienen om de data om te vormen naar de gewenste input voor het CORDIC blok. In ons project was er één nodig voor het ARCTAN blok.

We hebben voor onze CORDIC cores gekozen voor *maximum pipeline* architectuur. Deze keuze zorgt ervoor dat we een pipeline of unrolled architectuur voor het CORDIC algoritme opteren. Deze architectuur bestaat uit een cascade van CORDIC trappen die elk een deel van het totale algoritme uitvoeren. Zo zal de eerste trap steeds de verdraaiing doen over 45° , de tweede over 26.565° en zo verder. Tussen deze verschillende trappen staan telkens registers. Al de trappen werken simultaan zodat er elke klokpuls een nieuwe waarde bepaald wordt indien de pipeline volledig gevuld is. De opbouw van deze architectuur, met drie trappen, wordt weergegeven door afbeelding 5.6

Figuur 5.6: Unrolled architectuur bij CORDIC



6

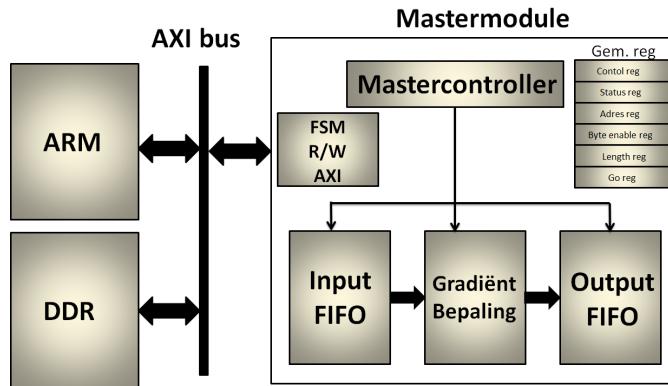
Gradiëntberekening aansluiten op de AXI bus

In hoofdstuk 4 hebben we de werking van onze VHDL code besproken om een gradiënt vector te bepalen. Uiteraard kan dit blok niet in deze vorm gebruikt worden. Er is nood aan hardware dat dit blok kan aansluiten op de AXI bus om zo in contact te staan met de PS. Hoe we dit gerealiseerd hebben wordt in dit hoofdstuk besproken.

6.1 Het blokschema

De wijze waarop we het gradiëntbepalend deel gaan koppelen aan de AXI bus staat principieel weergegeven op figuur 6.1. De binnenkomende beelden worden gestockeerd in het DDR geheugen door de ARM core. De ARM processor zal enkel de plaats van het start- en eind adres doorgeven aan het hardware gedeelte via de gemeenschappelijke registers. De input FIFO zal dan na een Go-commando vanuit de software de data tussen deze twee geheugenplaatsen, in bursts van 256 bytes, inlezen via de AXI bus. Het hardwaredeel zal deze data inlezen, verwerken en vervolgens opnieuw in een FIFO geheugen stockeren. Deze verwerking wordt gestuurd door de mastercontroller. De data van de output FIFO wordt opnieuw via de AXI bus naar het DDR geheugen gestuurd om opgeslagen te worden. De communicatie via de AXI bus wordt verzorgd door middel van een *FSM R/W* controller.

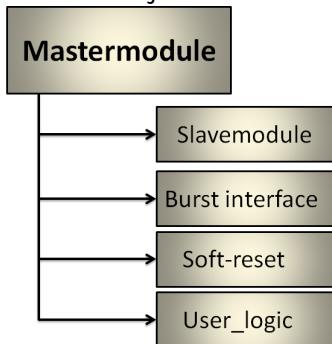
Figuur 6.1: Blokschema voor implementatie en aansturing van het gradiëntberekenend deel



6.2 Mastermodule

Er is gekozen om onze module master te maken op de AXI bus. Zo kan het hardware gedeelte zelf bepalen wanneer het zijn verwerkte data via de AXI bus naar het geheugen verzendt. Na het aanmaken van een nieuw hardware blok zal het ISE pakket al een aantal files genereren. De hiërarchie van deze files is te zien in figuur 6.2.

Figuur 6.2: De hiërarchie bij de files van een mastermodule



De top module bevat de port mapping van de onderliggende files. De eerste van deze onderliggende files is een slave interface op de AXI bus, deze file hebben we niet gebruikt. De tweede file is een interface voor datatransactie over de AXI bus. Deze roepen we op in de User logic file. De derde file maakt het mogelijk om een softwarematige reset te geven. De laatste file tenslotte is de User logic file. In deze file hebben we onze eigen hardware geïntegreerd. In het verdere verloop van dit hoofdstuk zal het dus steeds gaan over hoe we de implementatie, in deze file, hebben verwezenlijkt.

6.3 Inhoud van de User logic file

De User logic file bevat onze code om het gradiëntberekenend deel aan te sluiten op de AXI bus. Voor een snelle communicatie tussen processor en FPGA hebben ze beide toegang tot gemeenschappelijke registers. In de User logic file lezen we deze registers uit, in hoofdstuk 8

wordt het gebruik van deze registers verder besproken. Wij gebruiken vier gemeenschappelijke registers van 4 bytes zodat de gebruiker, via de software, de hardwaremodule kan initialiseren. Vervolgens kan de software de hardware laten starten door een "Go-commando". We beschikken met onze module over de volgende registers:

- **Control register:** geven van read/write request en instellen van burst operatie
- **Adres register:** instellen van het target adres
- **Byte enable register:** byte enable bus
- **Length register:** instellen hoeveel bytes er per burst gegeven worden
- **Go register:** op basis van dit commando start de hardware met het berekenen van de gradiënt
- **Status register:** een register dat de hardware gebruikt om zijn status door te geven aan de software

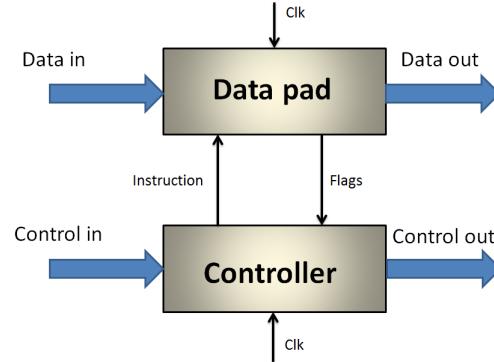
Figuur 6.3: Een overzicht van de wijze waarop de gemeenschappelijke registers gebruikt worden

		Status reg. C_BASE + 0x1	Controle reg. C_BASE + 0x0
	Adres reg. C_BASE + 0x4		
		Byte enable reg. C_BASE + 0x8	
Go reg. C_BASE + 0xF		length reg. C_BASE + 0xC	

Verder wordt er in de User logic file alle nodige hardware beschreven die nodig is om de gradiëntberekening correct aan te sluiten tussen beide FIFO's. Ook bevat de User logic file de nodige FSM's om de dataverwerking correct aan te sturen en de communicatie via de AXI bus te verzorgen. De User logic file bestaat dus uit een deel dat de data zal verwerken en een deel dat dit alles zal controleren, we spreken over een digitaal systeem.

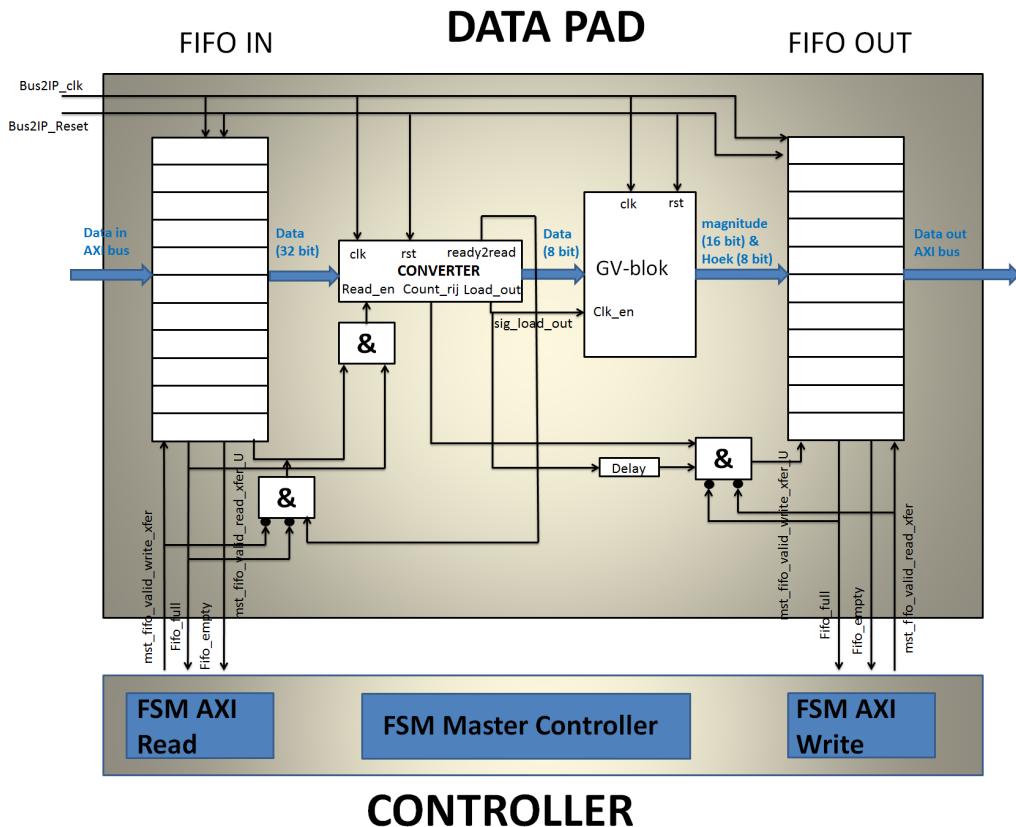
Een digitaal systeem bestaat uit twee delen: een data pad en een controller. Het data pad is in ons geval het gradiëntbepalend deel. In het data pad zal de inkomende data, in ons geval de input pixels, verwerkt worden volgens een bepaalde functie. Deze functie is in ons geval het bepalen van de gradiënt vector. Naast het data pad is er ook nog de controller. De controller zal aan de hand van instructies de bewerking die de data ondergaat in het data pad bepalen. De instructies kunnen gewijzigd worden door middel van flags die het data pad weergeeft aan de controller. Het principe van data pad en controller is weergegeven door figuur 6.4.

Figuur 6.4: Blokschema van een digitaal systeem



In schema 6.5 geven we een detailbeeld van ons digitaal systeem. In het datapad bevindt zich de gradiëntberekening (GV-blok) tussen de twee FIFO's. De extra hardware wordt verder in dit hoofdstuk verklaard. De controller bestaat uit een FSM dat aan de hand van de status van de FIFO's alles controleert. Ook bevinden er zich FSM's in de controller voor de communicatie over de AXI bus.

Figuur 6.5: Het digitaal systeem van de User logic file



In dit hoofdstuk worden nu zowel de controller als het data pad verder in detail besproken aan de hand van dit schema, FSM's en resultaten uit de testbench.

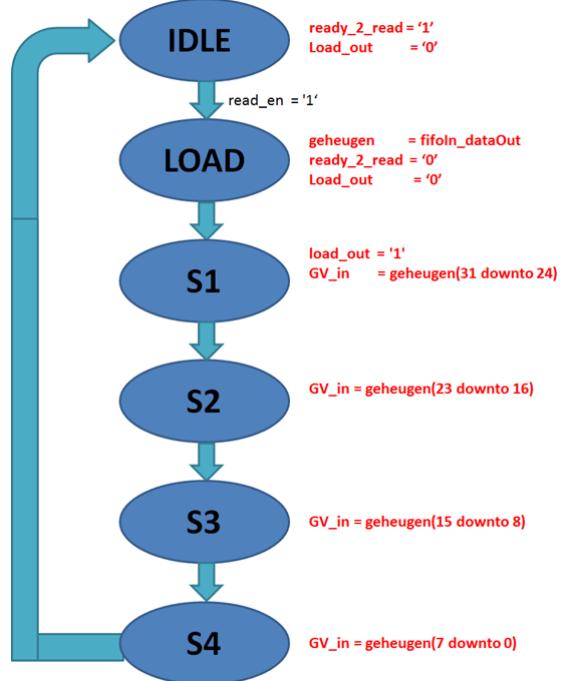
6.3.1 Data pad

De verwerking in het data pad is het bepalen van de gradiënt vectoren. Op figuur 6.5 is te zien dat het gradiëntbepalend deel, GV-blok, centraal geplaatst is tussen de twee FIFO's. De beide FIFO's hebben een diepte van 512 woorden van 32 bit breed. Omdat de input grijssignalen zijn (8-bit) kunnen er dus per woord vier input pixels gestockeerd worden. Ook bevatten de FIFO's controlesignalen om aan te geven wanneer ze vol of leeg zijn. Via de signalen "fifo-read" en "fifo-write" starten de FIFO's het lezen of schrijven.

Converter

Omdat er in het input FIFO per woord vier input pixels gestockeerd zitten hebben we een converterblok moeten maken om steeds maar 1 pixel per klok te laten toekomen op het GV-blok. Het converterblok werkt volgens het FSM in figuur 6.6.

Figuur 6.6: FSM van de converter

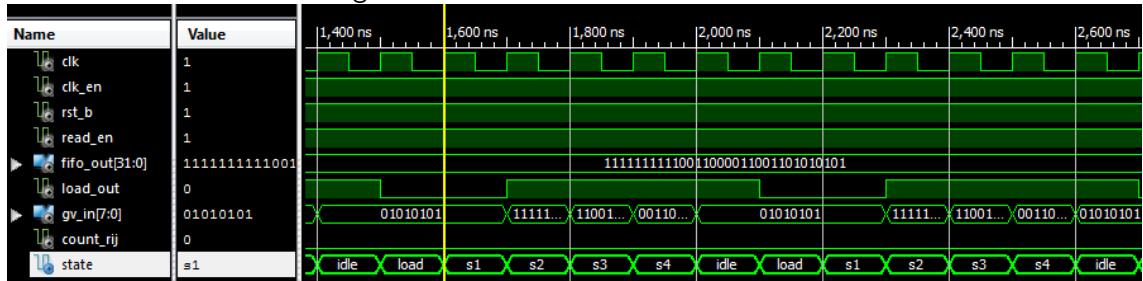


In de IDLE state geeft de converter aan de FIFO aan dat hij klaar is om data in te lezen. Wanneer ook de FIFO hiervoor klaar is zal de converter een woord inladen. Dit laden gebeurt in de LOAD state. Vervolgens wordt in de states S1 tot S4 telkens 1 byte doorgegeven aan het GV-blok. Tijdens de IDLE en LOAD state zal de converter het GV-blok ook uitschakelen, door middel van een enable bit, zodat het geen foutieve data verwerkt. Ook onthoudt de converter hoeveel pixels er reeds verwerkt zijn. Er is echter pas correcte data nadat er 3 rijen pixels ingeladen zijn. Dit geeft de converter aan zodat er pas hierna data in de output FIFO wordt opgeslagen.

De testbench van de converter is gegeven in figuur 6.7. Hierop is te zien dat de converter de input van 32 bit steeds per state als 1 byte naar buiten brengt en dat de "loadout" enkel dan hoog is. Dit zorgt ervoor dat enkel dan het GV-blok data verwerkt. Het signaal "State" is ook

gegeven, dit signaal geeft de volgende state aan, het signaal loopt dus een klokperiode voor. De uitgang van het GV-blok, een 16-bit lange magnitude en hoek van 8-bit, word dan geconcateneerd en gestockeerd in FIFO out.

Figuur 6.7: Testbench van de converter

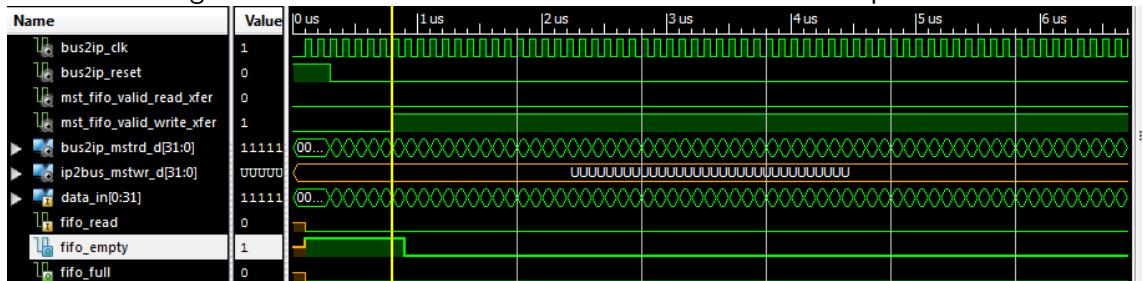


Simulatie van het data pad

Uiteraard hebben we ook het totale data pad getest in een testbench. Omdat dit een grote simulatie is splitsen we ze op en wordt telkens een deel verklaard.

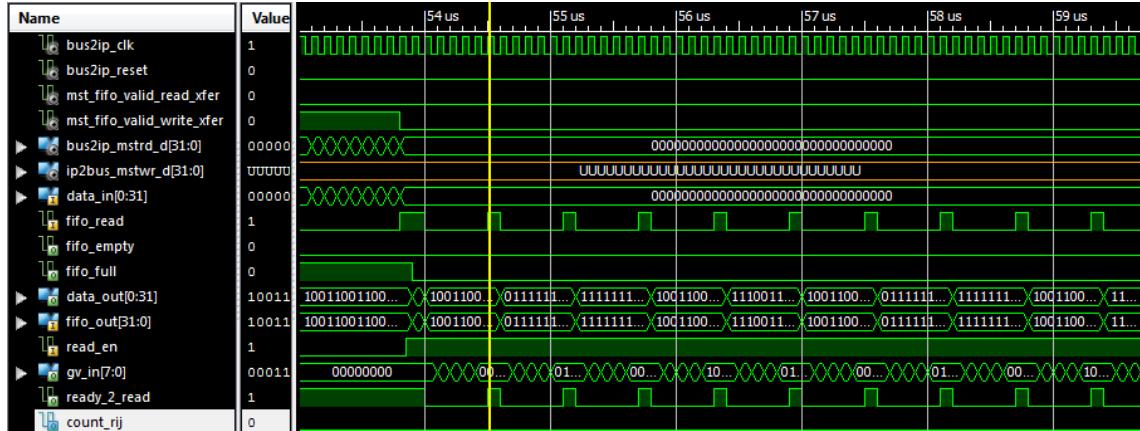
Figuur 6.8 geeft de beginsituatie weer. Nu zit er nog geen data in de input FIFO zodat het signaal "fifo-empty" hoog is. Het FSM master zal vervolgens het signaal "mst-fifo-valid-write-xfer" hoog maken zodat de FIFO gevuld wordt met de input pixels afkomstig van de AXI bus.

Figuur 6.8: Testbench totaal blok deel 1: vullen van de input FIFO



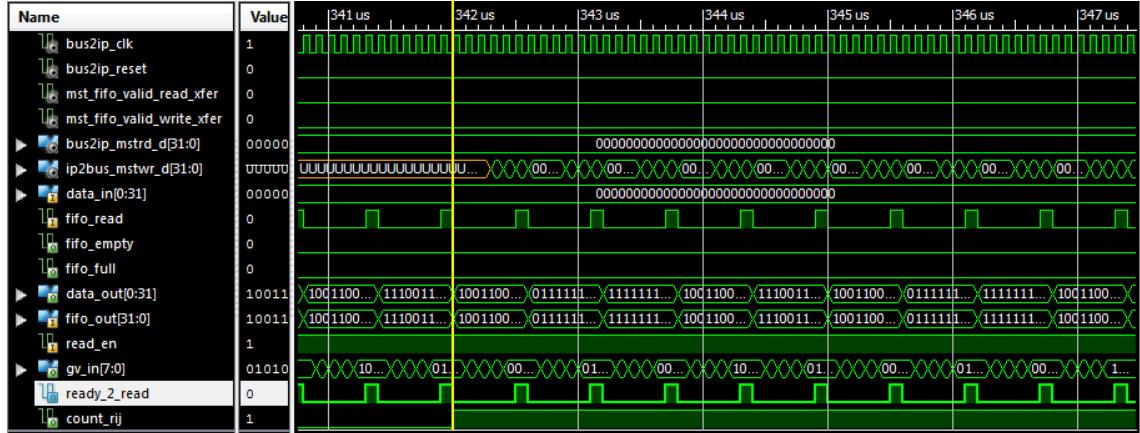
Wanneer de input FIFO gevuld is treedt de situatie op zoals in figuur 6.9. Het signaal "fifo-full" wordt hoog zodat de FIFO stopt met het lezen van de AXI bus, "mst-fifo-valid-write-xfer" wordt laag. Dit is het signaal om de converter te laten starten de input FIFO uit te lezen. Het "ready-to-read" signaal stond reeds lang klaar en de converter neemt het eerste woordt binnen. Zoals reeds besproken stuurt hij nu de data byte per byte naar het GV-blok via het signaal "gv-in"

Figuur 6.9: Testbench totaal blok deel 1: lezen van de input FIFO



Iets later treedt er de volgende actie op. Er zijn nu drie rijen pixels ingelezen waardoor pas vanaf nu de uitkomst van het GV-blok nuttige data is. Vanaf dit moment wordt het signaal "count-rij" hoog. Dit is te zien op figuur 6.10. Nu wordt de output FIFO gevuld tot hij helemaal vol is.

Figuur 6.10: Testbench totaal blok deel 1: vullen van de output FIFO



Wanneer de uitgangs FIFO helemaal vol is moet deze uitgelezen worden via de AXI bus. Dit is weergegeven in figuur 6.11. De controller geeft via het signaal "mst-fifo-valid-read-xfer" het bevel dat het lezen mag starten, De data komt op signaal "data-out" en het "fifo-full" signaal is niet langer hoog.

Figuur 6.11: Testbench totaal blok deel 1: lezen van de output FIFO



Het in- en uitlezen van zowel het input- als output FIFO register zijn de vier mogelijke situaties die in het data pad kunnen voorkomen. De volgorde van de situaties wordt bepaald door de master controller.

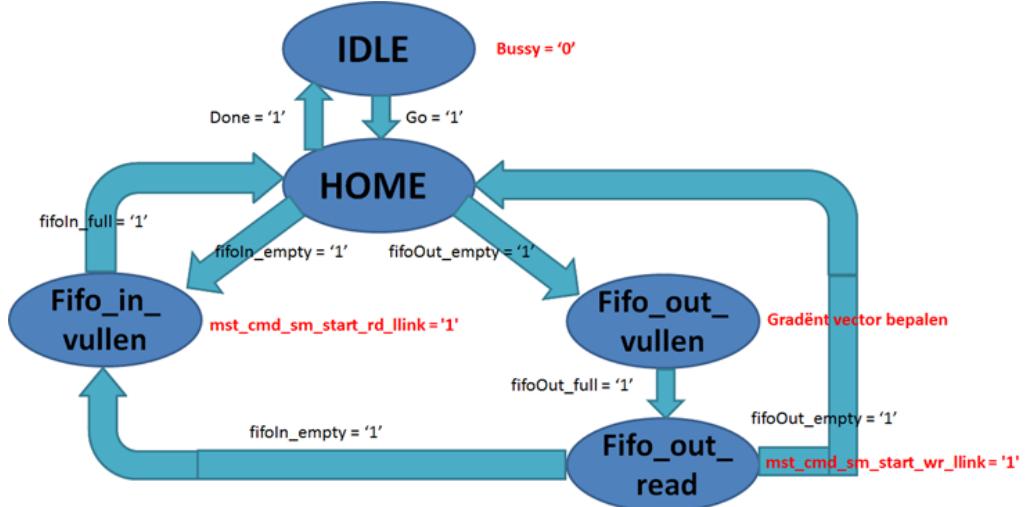
6.3.2 Controller

Het data pad van figuur 6.5 moet aan de hand van zijn status, die hij weergeeft via zijn *flags*, bestuurd worden, hiervoor zorgt de controller. De *flags* die de controller ontvangt van het data pad geven de status van de FIFO's weer. Ook bevat de controller FSM's die de communicatie via de AXI bus besturen.

FSM van de mastercontroller

Het FSM van onze toepassing staat in figuur 6.12. Het zal op basis van de status van de FIFO's en de inhoud van de gemeenschappelijke registers de werking van het data pad bepalen.

Figuur 6.12: FSM van de master controller



Het FSM start bij state IDLE. Hierin wacht hij op het "Go-commando" van de software. Dit commando is het signaal om één frame te verwerken. Hierdoor komt het FSM in de HOME state.

In de HOME state worden er drie controles gedaan. De eerste (hoogste prioriteit) is het controleren of er reeds een frame verwerkt is. Dit wordt bereikt door te tellen hoeveel maal de input FIFO werd geladen en het aantal keer de output FIFO werd uitgelezen tijdens de laatste keer dat de input FIFO gevuld werd. Stel dat het inputframe een beeld is van 680x480 pixels. Dan moet de input FIFO (van 512 woorden van 4 bytes) 150 keer ingeladen worden om een frame te verwerken. Omdat per woord dat in de input FIFO zit we 4 woorden nodig hebben in ons output FIFO (een byte wordt namelijk een magnitude van 16 en hoek van 8 bit) lijkt het logisch dat we onze output FIFO 4 keer meer moeten uitlezen dan we ons input FIFO schrijven. Maar niet al de verwerkte pixels worden opgeslagen. Er moeten eerst drie rijen pixels worden ingelezen vooraleer er nuttige data is. Het resultaat van de eerste twee rijen wordt daarom niet opgeslagen. Daarom is de verhouding, tussen hoeveel keer het input FIFO moet gevuld worden en het output FIFO gelezen wordt, niet exact vier. De laatste keer dat we de input FIFO vullen zal de inhoud van de output FIFO vijf keer moeten weggeschreven worden. Hierop

testen we dan ook of het hele frame verwerkt is.

De tweede controle kijkt na of het input FIFO leeg is, indien dit het geval is springt het FSM naar de "fifoIN-vullen STATE".

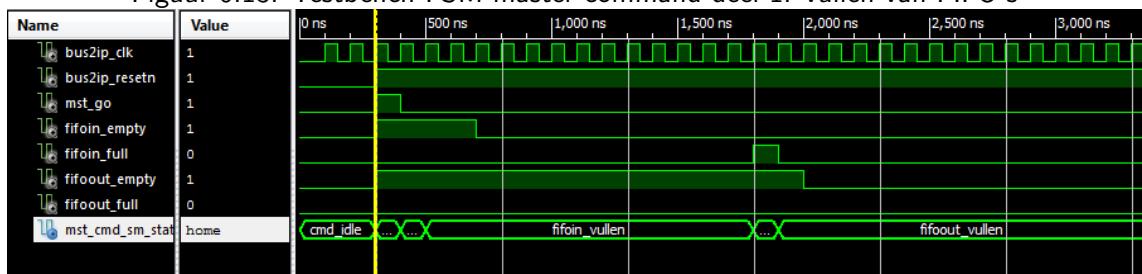
De laatste test gaat na of het output FIFO leeg is. Als dit het geval is dan springen we naar de "fifoOUT-vullen STATE".

In de state "fifoOUT-vullen" gaan we pixels uit het input FIFO lezen met de converter, het GV-blok zijn bewerking laten doen en het resultaat stockeren in het output FIFO. Tijdens deze state wordt er steeds gecontroleerd of niet één van de FIFO's leeg is. Indien zo wordt naar state gesprongen waar de lege FIFO gevuld wordt.

Het FSM 6.12 is een vereenvoudigd FSM. Tijdens de states "fifoIN-vullen" en "fifoOUT-lezen" worden nog stappen gebruikt die controleren of de data transfer over de AXI bus goed zijn verlopen en bij eventuele foutvlaggen errorbits in het controle register zetten. Maar deze zijn voor de duidelijkheid weggelaten.

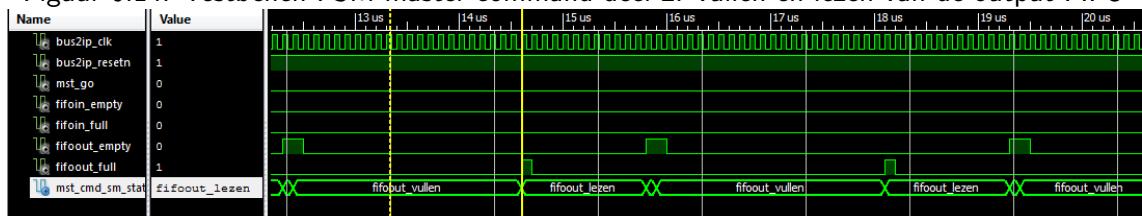
Dit FSM hebben we ook getest via een testbench. Na het genereren van een "mst-go" signaal gaven we signalen dat de FIFO's vol of leeg waren. In figuur 6.13 is de beginsituatie weergegeven. Hierbij bevindt het FSM zich nog in de IDLE state. Bij het krijgen van een GO-puls springt hij naar de HOME state. Daarna zal hij eerst de input FIFO vullen en vervolgens de output FIFO.

Figuur 6.13: Testbench FSM master command deel 1: vullen van FIFO's



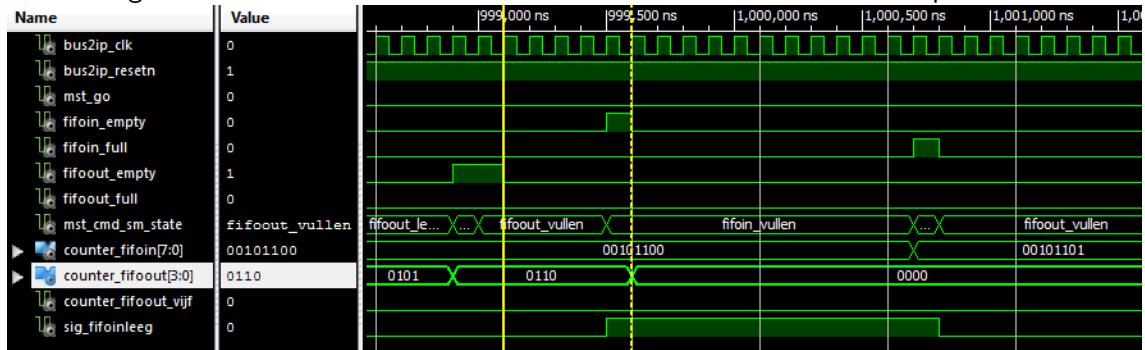
Omdat we met één vol input FIFO de output FIFO meerdere keren kunnen vullen, zien we dat op figuur 6.14 de output FIFO verschillende keren leeg loopt door zijn data over de AXI bus te versturen en gevuld wordt met resultaten van het GV-blok.

Figuur 6.14: Testbench FSM master command deel 2: vullen en lezen van de output FIFO



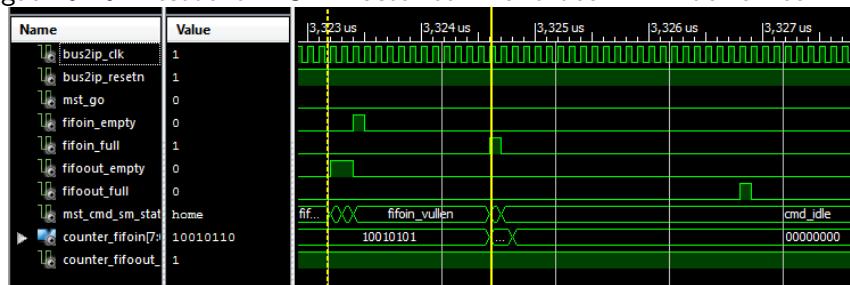
Ook de situatie waarbij de input FIFO leeg geraakt is gesimuleerd. Dit staat in figuur 6.15. Hierop is te zien dat we, zolang er nog waarden beschikbaar zijn in de input FIFO, gradiënt vectoren bepalen (state fifoOut-vullen). Vanaf het moment dat de input FIFO leeg is, springt het FSM naar de state waarin deze FIFO gevuld wordt. Na het terugkeren naar de home state begint het FSM opnieuw met bepalen van de gradiënt vector.

Figuur 6.15: Testbench FSM master command deel 3: vullen van input FIFO



Ook het einde van het verwerken van een frame zit in de testbench en is te zien op afbeelding 6.16. Hierbij is de input FIFO 150 keer gevuld en is al deze data verwerkt en doorgestuurd. Als deze twee voorwaarden voldaan zijn wordt er opnieuw naar de IDLE state gesprongen waar gewacht wordt op het volgende Go-signal van de software.

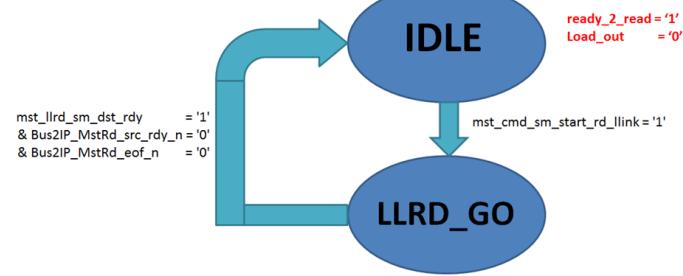
Figuur 6.16: Testbench FSM master command deel 4: Einde van een frame



FSM om te lezen van AXI bus

Om data te lezen van de AXI bus en zo de input FIFO te vullen, maken we gebruik van het FSM volgens afbeelding 6.17.

Figuur 6.17: Het FSM om data te lezen van de AXI bus

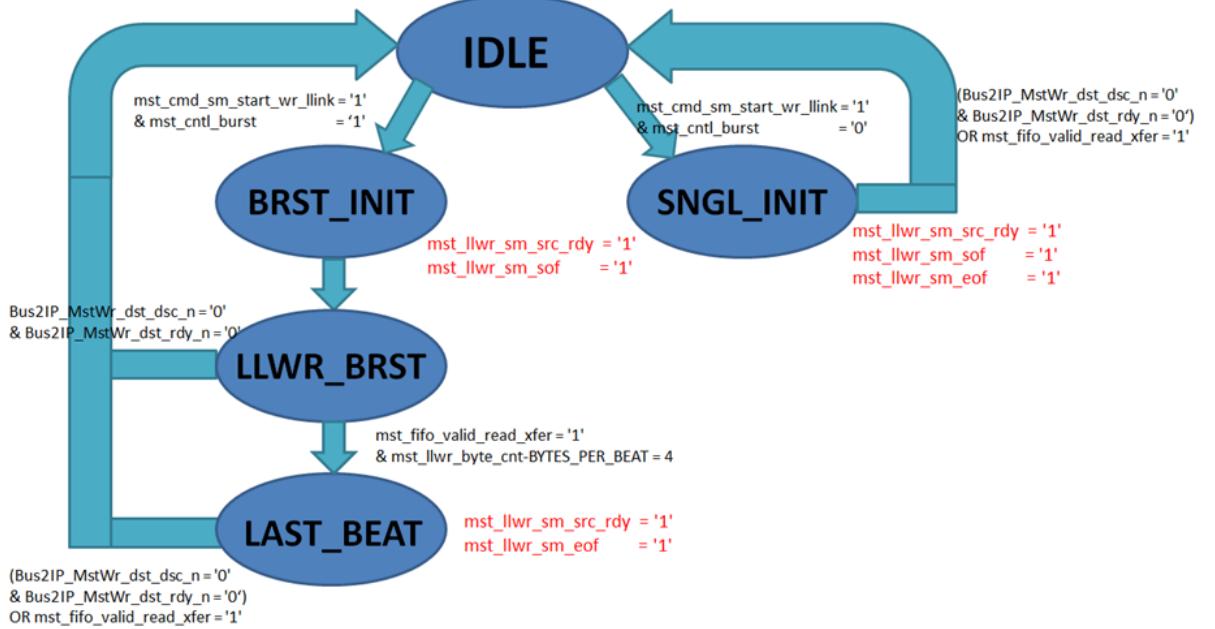


Wanneer er vanuit de mastercontroller het signaal gegeven wordt dat er data in gelezen moet worden via AXI bus springt het FSM naar de state LLDR-GO". In deze state zal er data worden ingelezen tot er een "ready-signaal" komt.

FSM om te schrijven naar AXI bus

Het schrijven van data vanuit de output FIFO wordt bereikt door middel van FSM op figuur 6.18.

Figuur 6.18: Het FSM om data te schrijven naar de AXI bus



Via het gemeenschappelijke controle register kan er geopteerd worden om al dan niet via bursttransmissie te werken. Afhankelijk van deze keuze wordt ofwel de "burst initialisatie" of de "single initialisatie" gedaan. In dit project wordt geopteerd om via burst transmissie te werken. In de state "LLDR-BRST" wordt daarna de data weggeschreven. Er is een speciale state "LAST-BEAT" voor gevallen waarbij de data niet verstuurd kan worden door een geheel aantal keer de bursts. Met andere woorden wanneer de hoeveelheid data niet deelbaar is door het aantal bytes-per-beat.

7

Booten van Linux OS in combinatie met eigen hardware

In de vorige hoofdstukken werd besproken hoe de hardware opgebouwd is. Deze hardware moet echter op de Zynq processor geïmplementeerd worden in combinatie met een Linux OS. Deze combinatie met een Linux OS is immers nodig omdat we via Linux een afbeelding kunnen inlezen vanop de SD-kaart en de software implementatie kunnen maken. In dit project hebben we gekozen voor Linaro Linux als OS.

Het booten van de Linux in combinatie met eigen hardware is een ingewikkeld proces, daarom wordt dit hoofdstuk besproken welke moeilijkheden we ondervonden en hoe we deze hebben opgelost. Meerdere reden lagen aan de basis van het probleem om Linux te booten met eigen geschreven hardware. Zo hadden we te kampen met de volgende zaken:

- Een eigen hardware blok moest opnieuw worden geïmporteerd na het invoegen van VHDL code
- Device tree moet steeds manueel aangepast worden na invoegen van nieuwe hardware blok
- Foutief maken van een boot image

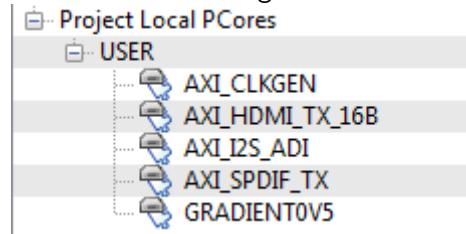
In dit hoofdstuk wordt het stappenplan besproken om een eigen hardware blok te combineren met een Linaro operatingsysteem. Bij elke stap waarbij we een probleem ondervonden zal ook telkens dieper ingegaan worden op de reden en de oplossing voor dit probleem. Voor een meer gedetailleerde uitleg verwijzen die we naar de handleiding [4].

7.1 Importeren van eigen hardware blok

De eerste stap om eigen hardware toe te voegen is het opstarten van een nieuw XPS project. De gebruiker kan nu meteen de standaard hardware, zoals LEDs en drukknoppen toevoegen.

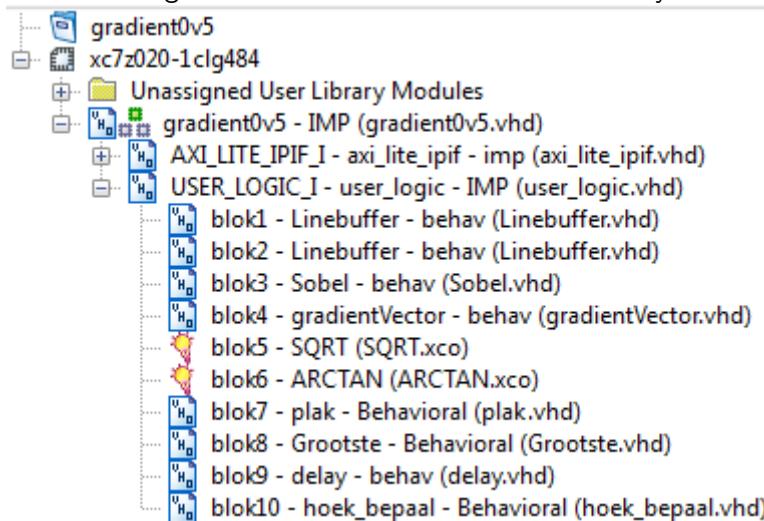
Wanneer een gebruiker een eigen hardware blok wil toevoegen moet hij de wizard "Create or import peripheral" starten in de XPS omgeving. Nadat deze doorlopen is komt het nieuwe hardware blok bij de USER bibliotheek te staan zoals in figuur 7.1.

Figuur 7.1: Voorbeeld van een gebruikers bibliotheek in XPS



Vervolgens kan de gebruiker naar het ISE project van dit hardware blok gaan om zijn eigen VHDL code te schrijven. De VHDL code van de gebruiker moet geschreven worden in de User logic file. Figuur 7.2 verduidelijkt dit.

Figuur 7.2: Voorbeeld van de ISE hiërarchie



Nadat de gebruiker zijn eigen VHDL code heeft ingevoegd in ISE, keert hij terug naar de XPS omgeving. Hier moet hij nu het hardwareblok opnieuw invoegen. Tijdens het invoegen moeten de geschreven VHDL files worden ingevoegd, ook NGO files kunnen ingevoegd worden. Om de CORDIC cores in te voegen moeten hun NGO files toegevoegd worden. Na het invoegen van het eigen hardwareblok moet er nog een netlist en bitstream gegenereerd worden.

7.2 Aanpassen van device tree

De device tree is een data structuur om de aanwezige hardware te beschrijven. Deze structuur wordt dan doorgegeven aan het operatingsysteem, in dit geval Linaro Linux. Dit zorgt ervoor dat niet alle hardware moet beschreven worden in het operatingsysteem zelf.

Na opzoekwerk hebben we gevonden dat het ISE pakket niet automatisch de device tree aanpast na het toevoegen van een hardware blok. Deze aanpassing moet de gebruiker manueel doen. Dit was één van de redenen waarom we voordien niet kon booten.

Telkens wanneer je een nieuw hardware blok aan de code toevoegt moet de gebruiker dus zelf het device toevoegen in de .dts file. Een voorbeeld hiervan is gegeven hieronder.

```
//EIGEN CORE
    master1axi_0{
        compatible = "dglnt, master1axi_0-1.00.a";
        reg = <0x6A000000 0x10000>;
    }
//EINDE EIGEN CORE
```

Vervolgens moet van deze .dts file een .dtb file worden gegenereerd. Hiervoor bestaat de tool: Device Tree Compiler (DTC). Een voorbeeld hoe dit gebruikt wordt zie je hieronder. Deze file moet vervolgens op de SD-kaart geplaatst worden waarmee je het Linaro systeem wil booten.

```
./dtc -I dts -O dtb -o devicetree.dtb /home/Desktop/digilent-zed.dts
```

7.3 Genereren van de boot file

Nadat de bitstream gegenereerd is kan deze geëxporteerd worden naar SDK. In de SDK omgeving moet er dan een nieuw project met een FSBL (first stage bootloader) worden aangemaakt. Vervolgens moet er een boot file gemaakt worden, dit gebeurt in het Xilinx Tools menu onder "create zynq boot image". Hierin kan de gebruiker drie files invoegen:

- FSBL.elf
- systeem.bit
- u-boot.elf

Het is zeer belangrijk dat de files in deze volgorde staan. Nu kan de boot file gegenereerd worden.

7.4 Compile Linux Kernel

De Linux kernel kan nu gecompileerd worden. Eerst moet de Linux kernel source code gedownload worden van de Digilent git repository. Vervolgens kan de gebruiker, indien nodig, de configuratie wijzigen. Om de kernel daarna te compileren wordt het volgende commando in een linux terminal uitgevoerd:

```
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

7.5 Klaarmaken van de SD-kaart

Tenslotte moet de SD-kaart klaar worden gemaakt om Linaro te kunnen booten. Op de kaart moeten twee partities worden aangemaakt. De handleiding hiervoor vindt u in [4]. Op de ZED_BOOT partitie plaatst de gebruiker de volgende vier files:

- BOOT.BIN
- Zimage
- device tree
- ramdisk8M.tar.gz

Op de ROOT partitie wordt het Linaro file system geplaatst. Nu is de SD-kaart klaar om Linaro met de eigen geschreven hardware op het Zedboard te booten.

8

Hardware-software communicatie

In de voorbije hoofdstukken werd besproken hoe de mastercontroller het hardwareblok bestuurt en hoe we een Linux omgeving verkrijgen. Nu moeten software en hardware met elkaar in contact kunnen staan. In dit hoofdstuk bespreken we eerst enkele tests die we hebben gemaakt voor de communicatie door middel van de gemeenschappelijke registers. Daarna beschrijven we een test die we hebben gedaan om het lezen en schrijven via de AXI bus te verduidelijken.

8.1 Communicatie via het gemeenschappelijk register

Om kennis te maken met de wijze waarop een hardware-software combinatie gevormd wordt met de Zynq processor hebben we enkele eenvoudige testvoorbeelden gemaakt. In de eerste test hebben we een eigen hardwareblok aangemaakt waarbij we gekozen hebben voor één gemeenschappelijk register. Dit register, van vier bytes, kan dus zowel in het hardware- als in het softwaregedeelte aangesproken worden. De VHDL code van dit hardwareblok kopieert de gegevens die in het gemeenschappelijke register zitten en geeft de inhoud van de laatste byte weer op de LEDs. Vervolgens hebben we voor dit hardwareblok een bootfile gegenereerd en de bijhorende files, zoals de device tree, aangepast om zo te kunnen booten met de SD-kaart. In de home directory van Linux hebben we dan een C-file toegevoegd, die vanuit het softwaregedeelte, data zal sturen naar het gemeenschappelijke register. Het compileren van deze C-file gebeurt in de Linaro Linux zelf via de terminal. Een moeilijkheid was dat de Linaro Linux de adressen van de gemeenschappelijke registers omvormde naar virtuele registers. Daarom wordt in het C-programma eerst in een lijst gecontroleerd met welk virtueel adres het adres van het gemeenschappelijke register overeenkomt. Naar dit virtueel adres werd dan een getal tussen 0 en 255 gestuurd. Dit getal werd door de hardware uit het register gelezen en op de LEDs zichtbaar gemaakt.

Vervolgens hebben we een software matige teller gemaakt zodat de status van de LEDs, de waarde van de teller weergaven.

Tenslotte hebben we het hardwareblok aangepast zodat deze ook data in het register kon

schrijven, dit keer las de software het gemeenschappelijk register uit.

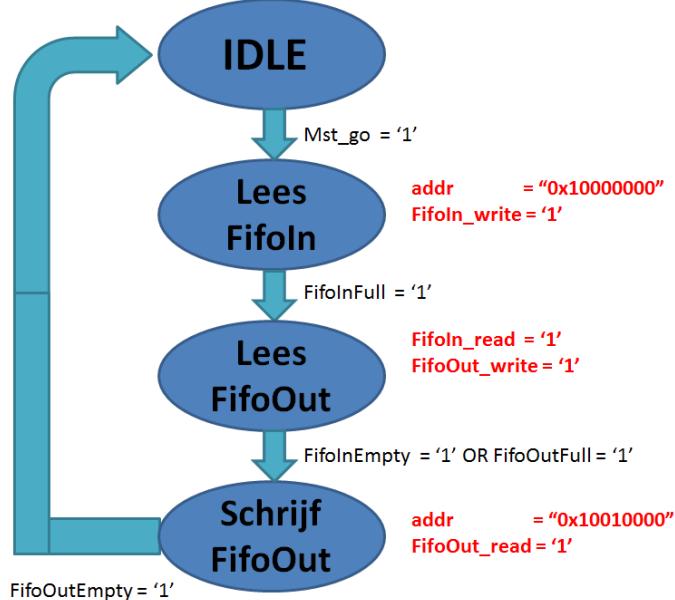
Door beide programma's hebben we ervaring opgedaan met zowel de communicatie tussen software en hardware als die tussen hardware en software. Deze wijze van werken konden we later gaan gebruiken om via software een Go-commando te geven aan het hardwareblok en de status van de hardware door te geven aan de software.

8.2 Inlezen van data via de AXI bus

In dit deel wordt besproken op welke manier de data, afkomstig van de software, naar het hardwareblok geschreven wordt. Zoals eerder aangehaald bevindt het gradiëntbepalend deel zich tussen een input- en output FIFO. Deze FIFO's worden gevuld via de AXI bus die de data ophaalt en stockeert in het RAM geheugen. Daarom hebben we een testprogramma gemaakt waarbij er in software data geschreven wordt naar het RAM geheugen, dit wordt daarna uitgelezen door het input FIFO. Vanuit het input FIFO wordt de data vervolgens naar het output FIFO gestuurd om daarna gestockeerd te worden op een andere plaats in het RAM geheugen. Op deze wijze wordt er gesimuleerd hoe er uiteindelijk moet gewerkt worden bij het verwerken van een afbeelding.

We hebben een FSM ontworpen dat de hardware stuurt tijdens deze test, deze is te zien op figuur 8.1. Dit FSM vervangt in deze test de eerder besproken mastercontroller uit hoofdstuk 6.3.2, voor deze test volstond immers een vereenvoudigde mastercontroller.

Figuur 8.1: FSM voor het testprogramma voor het vullen en lezen van FIFO's via de AXI bus



Nadat softwarematig een Go-commando is gegeven begint de input FIFO de data van het RAM in te lezen tot hij vol is. Vervolgens wordt de data van het input- naar het output FIFO getransfereerd om daarna opnieuw gestockeerd te worden in het RAM geheugen op een ander adres.

Het C-programma van deze test stockeerde vanaf geheugenplaats "0x10000000" van het RAM

geheugen oplopende waardes van 1 tot 200. Daarna werden de gemeenschappelijke registers ingesteld. Na een kleine delay, werd vervolgens op geheugenplaats "0x10010000" gecontroleerd of de FIFO's de data op een correcte manier behandelden.

In dit hoofdstuk hebben we aangetoond hoe de FIFO's data op een plaats in het RAM kunnen inlezen via de AXI bus en vervolgens, na ze al dan niet bewerkt te hebben, op een andere plaats weg te schrijven. Dit was het laatste onderdeel voor we een hardware implementatie van het gradiëntberekenend deel konden opbouwen.

9

Resultaten

In de voorbije hoofdstukken werd telkens een apart deel van onze implementatie met zijn resultaten besproken. In dit deel vatten we onze resultaten nog eens samen.

Dit project is opgestart aan het begin van dit academiejaar. We hadden, buiten het algoritme in OpenCV, nog geen informatie om onze studie op te baseren. Daarom zijn we dit project gestart met een uitgebreide studie. Deze behandelde enerzijds een studie rond het HOG algoritme zelf en anderzijds een studie over de hardware die we nodig hadden om dit project uit te voeren. Deze studie leidde tot de beslissing om het gradiëntberekenend deel eerst te implementeren gebruik makend van het Zedboard van Xilinx.

Na de studie moesten we dit hardwareplatform leren kennen, dit hebben we gedaan via tutorials van Xilinx. Vervolgens hebben we uitgezocht hoe we een Linux OS konden booten vanop de SD-kaart. Dit was een lastige procedure maar we zijn er toch in geslaagd. We hebben ook een procedurebeschrijving hiervan gemaakt om later problemen hier omtrent te vermijden.

De volgende stap was de hardwarebeschrijving van het gradiëntbepalend deel. Hiervoor hebben we een VHDL code geschreven en theoretisch bepaald dat deze implementatie een versnelling van 50% kan opleveren voor de gradiëntberekening. Ook hebben we voor deze code de nodige hardware voorzien om de connectie te maken met de AXI bus.

Als laatste hebben we programma's gemaakt die de software-hardware combinatie verzorgde. We hebben zowel programma's gemaakt die communiceerden via de gemeenschappelijke registers als een programma dat de FIFO's vulde en uitlas via de AXI bus.

Aan het einde van dit project hebben we elk deel dat nodig is voor de implementatie van het gradiëntberekenend deel gemaakt en getest. Jammer genoeg zijn we er niet toe gekomen om deze onderdelen samen te voegen.

9.1 Mogelijke verbeteringen

In dit hoofdstuk bespreken we mogelijke aanpassingen aan onze VHDL code en de totale implementatiewijze om een nog betere snelheidswinst te bekomen.

9.1.1 Wijzigingen aan huidige implementatie

Het deel dat momenteel geïmplementeerd is, het gradiëntbepalend deel, kan nog efficiënter uitgevoerd worden. Momenteel wordt het totaal binnenkomend beeld verwerkt in hetzelfde hardwareblok. Een snellere manier is de afbeelding op te splitsen in meerdere delen en deze te verwerken door meerdere hardware blokken. Zo kunnen we de afbeelding bijvoorbeeld in vier opsplitsen en elk deel gelijktijdig door vier afzonderlijke, identieke blokken laten verwerken. Dit zou ongeveer voor een snelheidswinst van factor vier zorgen. In deze methode moet echter rekening worden gehouden dat de vier delen elkaar moeten overlappen.

Naast het opsplitsen van het inkomende beeld kan er nog sneller gewerkt worden moest het huidige GV-blok meerdere malen uitgevoerd worden. Nadat de data wordt uitgelezen uit de input FIFO wordt het naar een converter gestuurd die de 4 bytes afzonderlijk naar het gradiëntbepalend blok stuurt. Een snellere uitvoering wordt verkregen indien er vier afzonderlijke GV-blokken worden geplaatst die elke byte gelijktijdig verwerken. Met deze methode moet er wel rekening worden gehouden dat de data nu in een andere volgorde wordt ingelezen en weggeschreven. Dit kan ten koste gaan van extra hardware blokken.

Verdere snelheidswinst kan geboekt worden door het verhogen van de burst lengte over de AXI bus en de diepte van de FIFO registers. Op deze wijze zijn er minder datatransacties tussen PS en PL.

9.1.2 Mogelijke uitbreidingen

Naast de besproken uitbreidingen aan het gradiëntbepalend deel kan er ook nog snelheidswinst worden geboekt op de andere delen van het algoritme.

Zo kan er nog sneller gewerkt worden moesten er nog meer delen in hardware worden omgevormd. Zo zou de eerste stap, het normaliseren van de afbeelding, in hardware kunnen uitgevoerd worden. Ook het deel nadat de gradiënten bepaald zijn, het indelen in cellen en blokken zou geïmplementeerd kunnen worden.

Indien er beslist wordt om verdere delen van het HOG om te vormen naar hardware zouden we ervoor ophouden om de delen vlak voor of vlak na het gradiëntbepalend deel om te vormen. Dit zou overbodig data transport tussen PS en PL verminderen.

10

Conclusie

Het doel van deze thesis was onderzoeken hoe het HOG algoritme kan geïmplementeerd worden in een FPGA of een software-hardware architectuur. Deze implementatie moest gebaseerd worden op een studie van het algoritme dat onderzocht welk deel zich leent tot hardware implementatie. Daarna moet een keuze gemaakt worden voor een hardwareplatform om de implementatie te verwezenlijken. Als laatste moet er dan de uiteindelijke opbouw verwezenlijkt worden.

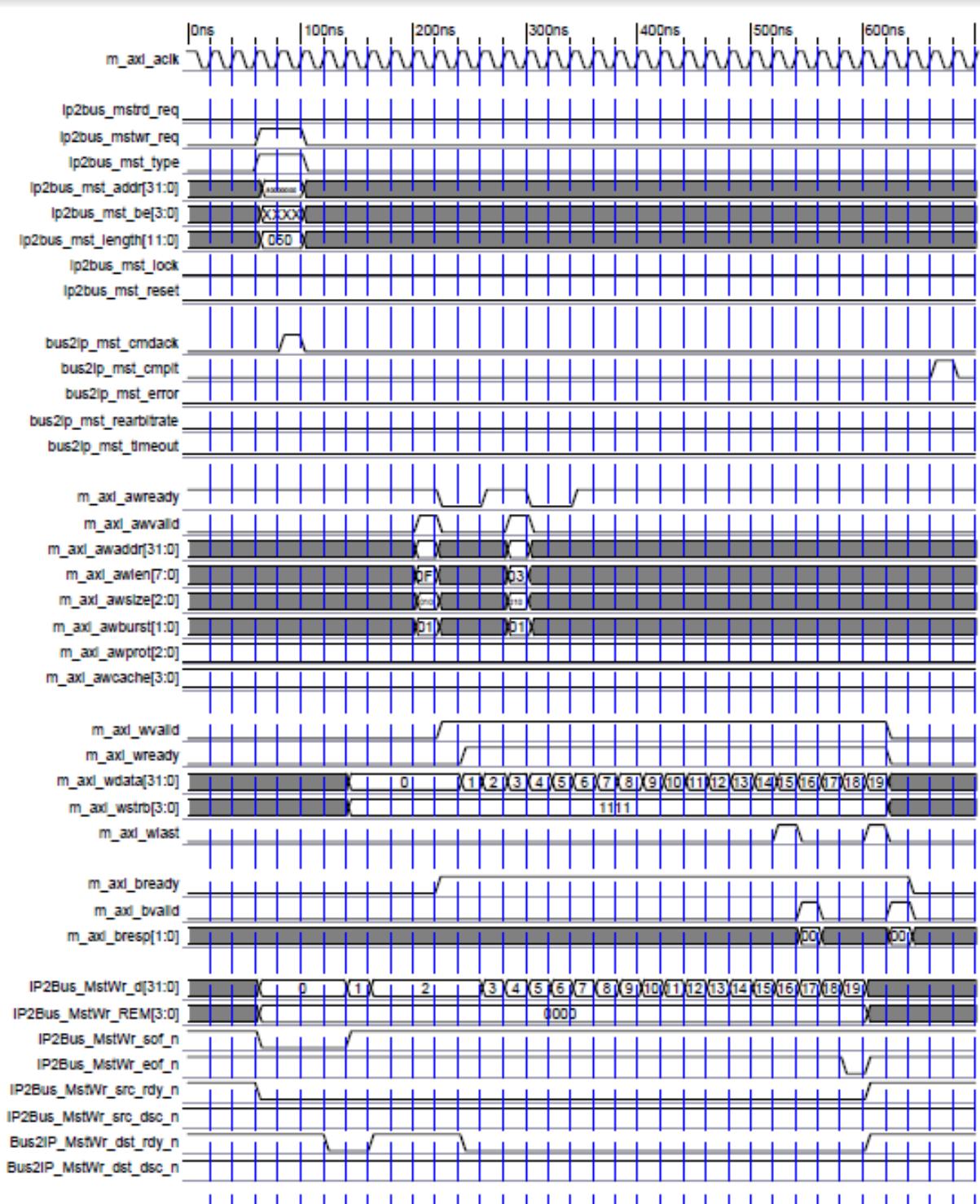
Aan het einde van ons project kunnen we besluiten dat we het merendeel van onze opdracht tot een goed einde hebben gebracht. Zo hebben we na een uitgebreide studie een keuze gemaakt over welk deel we eerst konden implementeren, dit werd het gradiëntbepalend deel. Ook hebben we op basis van deze studie een keuze gemaakt omtrent het hardwareplatform dat we konden gebruiken voor deze implementatie, namelijk het Xilinx Zedboard. Na de kennismaking met het hardwarebord zijn we erin geslaagd om via de SD-kaart een Linux OS te booten op het Zedboard. Verder hebben we ook het gradiëntberekenend deel beschreven in VHDL en berekend dat onze implementatie de snelheid van dit blok met 50% kan verbeteren. Ook hebben we de nodige hardware voorzien om deze code correct aan te sluiten op de AXI bus. Daarnaast hebben we ook testprogramma's gemaakt die data transfereerde tussen software en hardware. Als laatste maakten we suggesties voor eventuele verdere optimalisaties. We zijn er helaas niet in geslaagd om al deze delen samen te voegen om een totale implementatie te maken en zo afbeeldingen te verwerken.

Als algemeen besluit kunnen we stellen dat het grootste werk van deze opdracht is afgerond. Hoewel we geen totale implementatie hebben kunnen verwezenlijken, zijn alle onderdelen voor een totale implementatie ontworpen en getest.

Bibliografie

- [1] H. Bay, T. Tuytelaars, and L. V. Gool. Surf: Speeded up robust features. 2006.
- [2] C. Bourrasset, L. Maggiani, C. Salvadori, J. Sérot, P. Pagano, and F. Berry. Fpga implementations of histograms of oriented gradients in fpga.
- [3] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. 2005.
- [4] Digilent. Embedded linux hands-in tutorial – zedboard.pdf. Technical report, 2013.
- [5] Felzenszwalb, Girschick P., R., and D. McAllester. Cascade object detection with deformable part models. 2010.
- [6] LUO Hai-bo, YU Xin-rong, LIU Hong-mei, and DING Qing-hai. A method for real-time implementation of hog feature extraction. 2011.
- [7] Brookshire Jon, Steffensen Jorgen, and Xiao Jianxiong. Fpga-based pedestrian detection.
- [8] Van Beeck Kristof, Heylen Filip, Meel Jan, and Goedemé Toon. Comparative study of model-based hardware design tools. 2010.
- [9] Chris McCormick. hog person detector tutorial. <http://chrisjmccormick.wordpress.com/2013/05/09/hog-person-detector-tutorial/>, 2013. [Online; accessed 22-05-2014].
- [10] Victor Adrian Prisacariu and Ian Reid. fasthog - a real-time gpu implementation of hog. 2012.
- [11] Jack E Volder. The cordic trigoniometric computing technique. 1959.
- [12] J. S. Walther. A unified algorithm for elementary functions. 1971.
- [13] Xilinx. Logicore ip axi master burst.pdf. Technical report, 2011.
- [14] Qiang Zhu, Shai Avidan, Mei-Chen Yeh, and Kwang-Ting Cheng. Fast human dectection using a cascade of histograms of oriented gradients.

Bijlage 1



FACULTEIT INDUSTRIELE INGENIEURSWETENSCHAPPEN
CAMPUS GROEP T
Andreas Vesaliusstraat 13
3000 LEUVEN, België
tel. + 32 16 30 10 30
fax + 32 16 30 10 40
iiw.groep.t.leuven@kuleuven.be
www.iw.kuleuven.be



LID VAN
**ASSOCIATIE
KU LEUVEN**