KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

DSP: Kleurconversie toepassen op een real time videosignaal

Gert-Jan Andries
Xavier Dejager
Nick Steen

Master of Science in de industriële ingenieurswetenschappen: elektronica-ICT, optie Elektronica

Docent:
Dries Debouvere

Academiejaar 2016 – 2017

Inhoudsopgave

Afkortingen en symbolen

Afkortingen

ALU	Arithmetic logic unit
AVF	Active video field
cfr	Confer
CPU	Central processor unit
DAG	Data address generetor
DMA	Direct memory acces
DSP	Digital Signal Processing
EAV	End of active frame
Hz	Hertz
IDE	Integrated development environment
ITU	International Telecommunication Union
kB	Kilobyte
MAC	Multiply and accumulate
MB	Megabyte
MHz	Megahertz
NTSC	National Television System Committee
PAL	Phase Alternating Line
PPI	Peripheral Parallel Interface
RGB	Rood Groen Blauw kleurenruimte
RISC	Reduced instruction set computing
SAV	Start of active frame
SDRAM	Synchronous dynamic random access memory
SIMD	Single instruction multiple data
SRAM	Static random acces memory
TV	Televisie
YCbCr	zie YUV
YUV	Helderheid en Kleurvariatie kleurenruimte

Hoofdstuk 1

Inleiding

DSP staat voor Digital Signal Processing, ofwel digitale signaalverwerking. Deze signalen kunnen zowel beeld- als audiomateriaal zijn, in dit labo is gekozen om het videodomein verder te bestuderen. De opdracht bestond uit de real-time verwerking en filtering van een videosignaal. Dit houdt onder meer in dat een signaal binnengelezen, geanalyseerd, gefilterd of aangepast en terug uitgestuurd moet worden, en dit in een zo kort mogelijke tijdspanne om de real-time factor te respecteren.

Gedurende het laboproject werd gewerkt met een beeldfragment van een aflevering van de Smurfen. In dit fragment dienen alle Smurfen een huidskleurverandering te ondergaan zodat het uitgangssignaal een verzameling gele smurfen weergeeft. Het middel om dit te bereiken is de BlackFin EZ-Kit Lite met de BF-561 dual-core processor, met bijbehorende DSP++ IDE. Een krachtige processor in combinatie met een eenvoudige werkomgeving belooft een vlotte ervaring te geven.

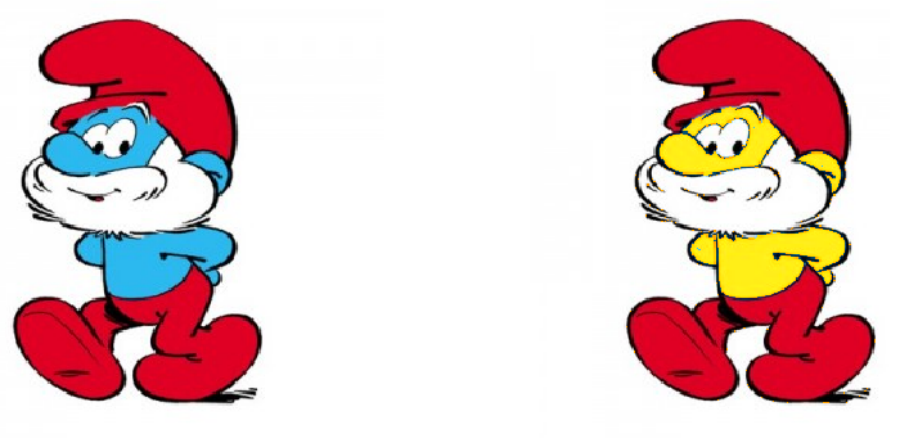
Hoofdstuk 2

Doelstelling

Het doel van dit DSP project is om met behulp van de Blackfin BF-561 DSP processor een kleurenconversie toe te passen op een realtime videosignaal. Dit videosignaal is afkomstig van een digitale camera en wordt via het NTSC protocol uitgestuurd. De kleursverandering die moet doorgevoerd worden is een transformatie van blauw (zoals bij de Smurfen) naar geel.

Als voorbereiding voor dit labo werden reeds enkele DSP-algoritmes uitgevoerd in het simulatieprogramma Matlab. Aan de hand van de opgedane kennis kan het DSP alghoritme tijdens dit labo geïmplementeerd worden op een DSP processor aan de hand van de programmeertaal C++.

Naast de praktische implementatie wordt er ook een studie gedaan naar kleurenruimtes en -converties, alsook een theoretische studie naar technieken die de performatie en de kwaliteit van het eindreslultaat kunnen vergroten door gebruik te maken van dezelfde hardware. Er werd dus theoretisch gezocht naar een zo performant mogelijk algoritme.



FIGUUR 2.1: Huidskleurverandering bij een smurf

Hoofdstuk 3

BlackFin architectuur

3.1 BlackFin processor algemeen overzicht

De BlackFin processorfamilie is een 16- of 32-bit microprocessor ontwikkeld door het bedrijf Analog Devices. De processor beschikt over een ingebouwde fixed-point DSP die wordt bijgestaan door twee 16-bit MACs. Hiernaast is er in de chip ook nog een kleine processor aanwezig. De BlackFin processorreeks is ook ontwikkeld met het oogpunt op low power functionaliteiten.

De BlackFin gebruikt een 32-bit RISC microcontroller die geïmplementeerd is op een SIMD architectuur. Een BlackFin core bevat twee 16-bit hardware MACs, twee 40-bit ALUs en een 40-bit barrel shifter. Deze opstelling zorgt ervoor dat er maximaal drie instructies per klokcyclus uitgevoerd kunnen worden. Om het aantal instructies te reduceren zijn er ook vier DAGs ingebouwd in de processor.

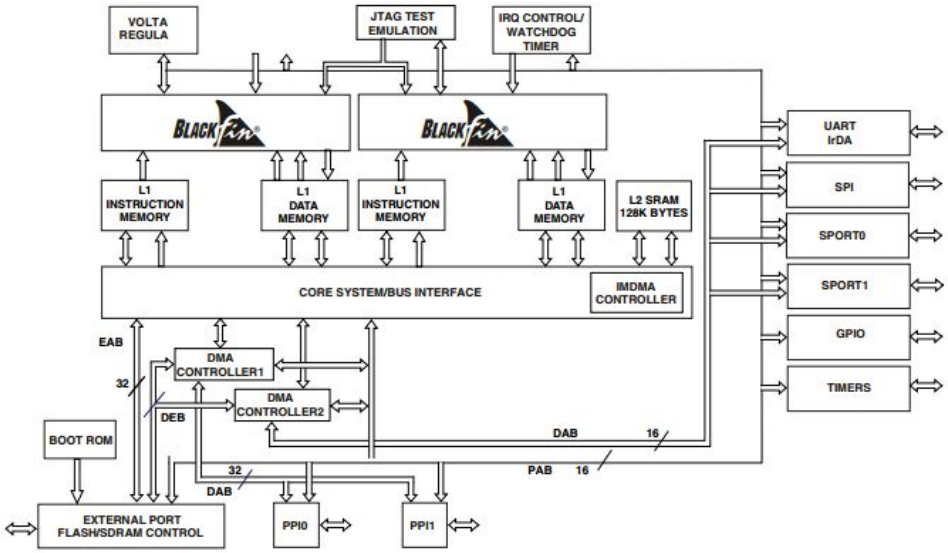
De BlackFin maakt gebruik van een byte-adresseerbare flat memory map. Het level 1 en level 2 geheugen is ingebouwd in de processor en het level 3 geheugen dient extern toegevoegd te worden. Al deze geheugens zijn gemapt op 32-bit adressen. Dit zorgt er voor dat de BlackFin processoren opgebouwd zijn volgens de Von Neumann architectuur.

Het level 1 intern SRAM geheugen, dat geklokt is op de snelheid van de processor, is gebaseerd op de Harvard architectuur. Het instructie- en datageheugen is verbonden met de processor via een gereserveerde databus. Het level 2 geheugen is trager geklokt dan de processor zelf. De codememory kan zowel in level 1 als in level 2 geheugen opgeslagen worden.

De BlackFin processor beschikt over twee DMA-controllers die kunnen opereren tussen de peripherals en de verschillende geheugens binnen de processor. Dankzij deze DMA-controllers wordt de processor niet belast met het uitvoeren van data transfers in het geheugen.

3.2 BlackFin BF-561 processor

De BlackFin BF-561 die aanwezig is op het EZ-KIT Lite beschikt over twee processorkernen. De processor kan werken op een kloksnelheid van maximaal 600 Mhz. Verder beschikt deze processor over 328 kB intern geheugen (level 1 en level 2). Hiernaast beschikt de processor over twee 12-kanaals DMA controllers om de geheugen-naar-geheugen operaties uit te voeren. In figuur 3.1 is een overzicht van de opbouw van de BlackFin BF-561 weergegeven.



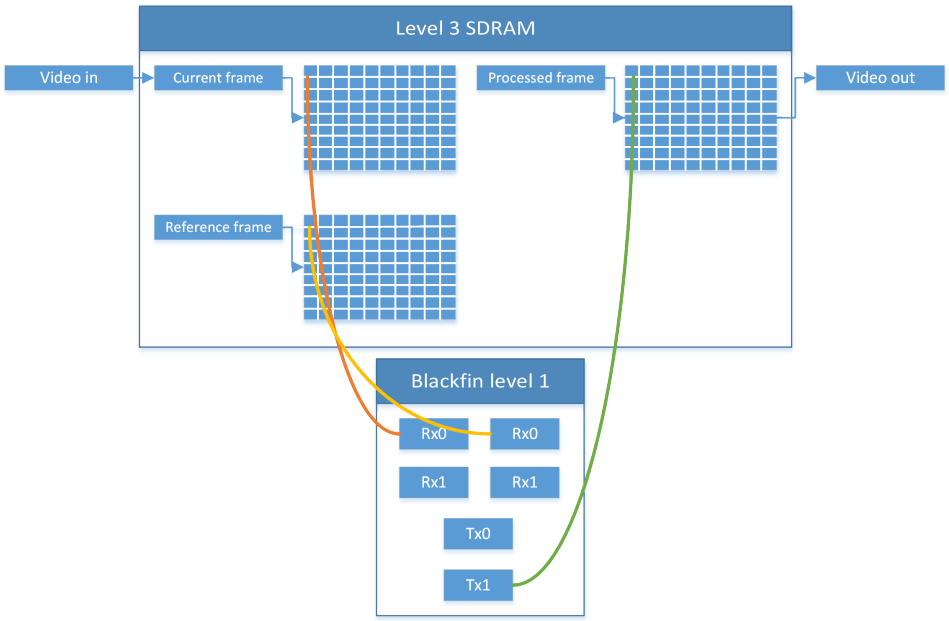
FIGUUR 3.1: BF-561 functional block diagram

Hoofdstuk 4

Video Processing

4.1 Interframe processing

Interframe processing wordt toegepast wanneer er een afhankelijk verband is tussen de verschillende frames. Dit is bijvoorbeeld het geval wanneer een inkomend frame vergeleken moet worden met een referentie frame. Het level 1 en 2 geheugen van de BlackFin processor zijn onvoldoende groot om deze frames in op te slaan. Er dient dus gebruik gemaakt te worden van het level 3 SDRAM. Het nadeel aan dit SDRAM geheugen is dat het veel trager is dan het ingebouwde level 1 en 2 geheugen van de processor zelf. Een frame wordt onderverdeeld in verschillende delen die een sub block genoemd worden. Deze sub blocks worden vervolgens getransfereerd naar het level 1 geheugen van de BlackFinprocessor om verwerkt te worden. Indien het level 1 geheugen nog te klein blijkt te zijn kan het level 2 geheugen gebruikt worden als tussenbuffer. Alle geheugenoperaties worden uitgevoerd door de DMA-controller. In figuur 4.1 wordt een voorbeeld weergegeven van de hierboven beschreven techniek. Er bestaat een afhankelijkheid tussen currentframe en referenceframe.



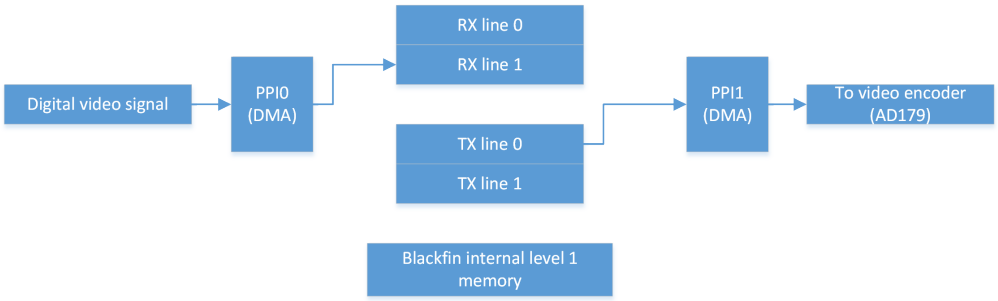
FIGUUR 4.1: Interframe processing data flow diagram

4.2 Line processing

Bij het line processing principe wordt frame per frame te werk gegaan. Het frame zal in een eerste stap opgehaald worden uit het geheugen van de peripheral. Nadat een frame opgehaald is wordt het lijn per lijn verplaatst naar het level 1 geheugen van de BlackFin. Deze actie gebeurt door middel van de DMA-controller zodat de processor hier geen hinder van ondervindt.

Een nog efficiëntere methode is om de data onmiddelijk van de peripheral lijn per lijn naar het level 1 geheugen te transfereren. Na der verwerking moet ook de data terug verstuurd worden naar een peripheral om deze weer te geven. Dit kan ook onmiddelijk lijn per lijn vanuit het level 1 geheugen, maar ook via het level 3 geheugen waar een volledig frame gebufferd kan worden.

In figuur 4.2 wordt de opbouw van een lijn-gebaseerde aanpak weergegeven. De DMA wordt gebruikt om data onmiddelijk naar het level 1 geheugen van de BlackFin te transfereren. Vervolgens wordt door de BlackFin een actie uigevoerd op de videolijn die dan weer wordt getransfereerd van het level 1 geheugen naar de video encoder. Deze transfers gebeuren door middel van de DMA-controlller. De dubbele buffering in het level 1 geheugen zorgt ervoor dat de DMA en de processing door de BlackFin processor concurrent uitgevoerd kunnen worden. Op deze manier kan video in realtime verwerkt worden.



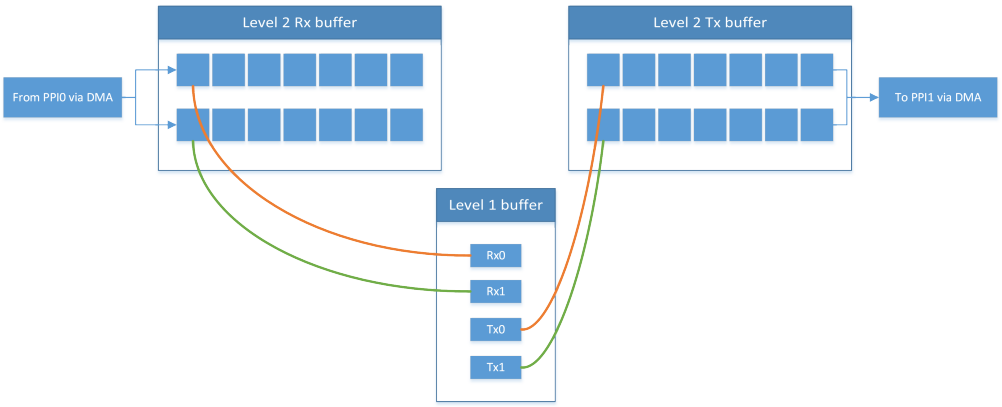
Figuur 4.2: Line processing data flow diagram

4.3 Macro block processing

Bij macro block processing wordt het frame niet in zijn geheel verwerkt, maar opgedeeld in verschillende secties. Elke sectie heeft een dimensie van n bij m. De hoogte van de macroblock wordt weergegeven door n en de breedte door m. Het level 1 geheugen van de BlackFinprocessor is te klein om een alle n-lijnen van een afbeelding op te slaan, daarom wordt gebruik gemaakt van het level 2 geheugen. Dit zorgt ervoor dat er minder bandbreedte van de databus gebruikt wordt.

Ook bij het level 2 gheugen is de opslagcapaciteit beperkt. Een volledige afbeelding past vaak niet in het level 2 geheugen. Daarom wordt slechts een aantal lijnen van de afbeelding (n lijnen) in het level 2 geheugen geplaatst. Dit gebeurt door middel van de parallel peripheral interface van de DMA. Vervolgens zullen macroblocks verplaatst worden van het level 2 geheugen naar het level 1 geheugen waar deze verwerkt kunnen worden door de BlackFin processor. Er vindt een dubbele buffering plaats in zowel het level 1 als het level 2 geheugen. Telkens wordt de DMA-controller gebruikt voor deze geheugenoverdracht zodat de processor hier geen hinder van ondervindt.

In figuur 4.3 is een voorbeeldopstelling te zien van macro block processing. In dit voorbeeld worden twee DMA-controllers gebruikt om de dataoverdrachten uit te voeren. Er wordt ook een dubbele framebuffering voorzien om de processwindow te vergroten.



FIGUUR 4.3: Macro block processing data flow diagram

4.4 Optimalisatie

Door bij het design rekening te houden met de architectuur van de BlackFin processor kan er flink wat snelheid gewonnen worden. In de paragrafen hieronder worden enkele design tips kort toegelicht.

Wanneer men via een DMA-controller data wil verplaatsen van het ene geheugen naar het andere geheugen dient men rekening de houden met de breedte van de data bus. Bij de BlackFin BF-56x reeks is er een 32-bit brede databus voorzien. Door de DMA-controller zo in te stellen dat hij per operatie telkens 32-bit of een veelvoud daarvan moet verplaatsen, wordt er optimaal gebruik gemaakt van de DMA en databus.

Om de DMA-controllers efficiënt te gebruiken dient men er rekening mee te houden bij het ontwerp dat nooit twee geheugentransferts plaatsvinden op het zelfde moment binnen eenzelfde DMA kanaal.

Bronnen: [7], [10], [11], [14].

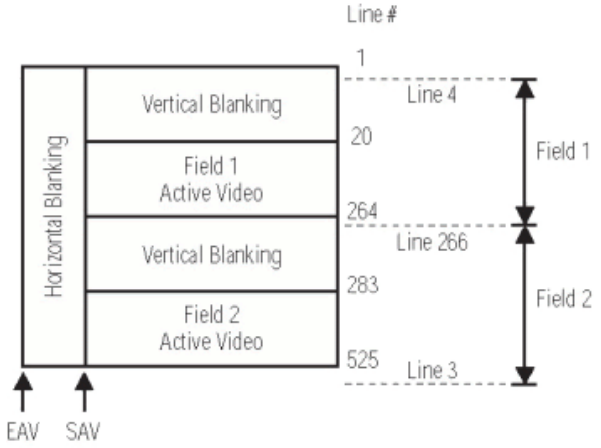
Hoofdstuk 5

Video Input

5.1 Beeldformaat

5.1.1 Video frame

De video input is een stream van frames die elk een beeld voorstellen. De NTSC standaard, gedefinieerd volgens ITU-R BT656 [5] [8], legt een framerate van 60Hz op, dus 60 frames per seconde. Elk frame bestaat uit verschillende lijnen data, zichtbare en onzichtbare. De zichtbare lijnen zijn gegroepeerd per even en oneven regels, en zijn dus in twee velden, Active Video Fields (AVF), terug te vinden in het binnekomende frame. De overige, en dus onzichtbare lijnen, zorgen voor synchronisatie tussen de bron en de weergever. Deze regels worden ook blanking lines genoemd. Hoe de lijnen precies voorkomen in een frame is te vinden in figuur 5.1.



Line Number	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0

Figuur 5.1: Opbouw ITU-R BT656 frame [2]

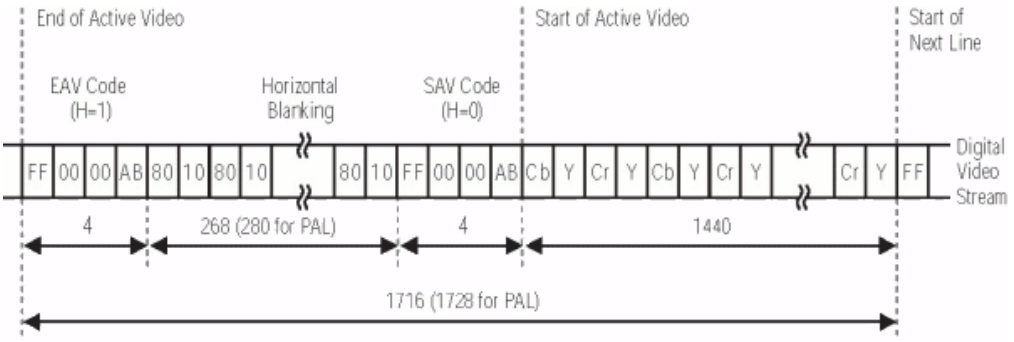
Er is duidelijk te zien dat er eerst een groep lijnen komt die de vertical blanking lijnen voorstellen. Deze regels gaven de vroegere CRT televisietoestellen de tijd om hun beam van beneden tot weer volledig boven te bewegen. Aangezien de nieuwere toestellen niet meer met een kathodestraalbuis werken, maar over een volledig digitaal verwerkingssysteem beschikken, kan deze data overschreven worden met timingintervallen of andere metadata. Er kan dus data meegestuurd worden

binnenin de blanking intervallen, maar deze wordt door vele toestellen uitgefilterd om interferentie van beeldlijnen te voorkomen.

De vertical blanking is opgevolgd door het eerste AVF. Deze eerste groep bevat de oneven lijnen van het zichtbare beeld. Hoe een lijn precies in elkaar zit wordt in het volgende onderdeel verder beschreven, hier beperken we ons tot de adressering van de lijnen binnen het frame. Dit eerste AVF is gevolgd door opnieuw een vertical blanking en een AVF. Een ITU-R BT656 frame bestaat uit 525 opeenvolgende lijnen. Aangezien het zictbare beeld 480 lijnen telt, blijven er 48 lijnen over die kunnen dienen voor vertical blanking, wat duidelijk overeenkomt met figuur 5.1 op pagina 9.

5.1.2 Structuur van een beeldlijn

Er zijn twee soorten lijnen, de zichtbare en de onzichtbare. Ze delen eenzelfde opbouw, maar hebben een groot verschil, namelijk dat de zichbare lijnen een AVF bevatten waar de onzichtbare lijnen een vertical blanking field bevatten. Zowel de headers als de lijnlengte zijn exact gelijk en kunnen dus beschouwd worden als eenzelfde opeenvolging van data. Men moet er bij het verwerken wel voor zorgen dat er niet over de vertical blanking geschreven wordt op willekeurige plaatsen, omdat dit ongewenste effecten kan hebben, afhankelijk van het type toestel waarop het beeld weergegeven wordt.



Figuur 5.2: Opbouw ITU-R BT656 lijn [1]

Een lijn bestaat uit verscheidene elementen: EAV code, horizontal blanking, SAV code en data. De EAV code heeft een lengte van 4 bytes en bestaat uit drie bytes met een vaste waarde (0xFF 0x00 0x00), gevolgd door een vierde byte die aanduidt of de data die volgt bestaat uit de even of oneven lijnen, alsook de status van de blanking. Op deze manier weet het toestel of de lijn een blanking, al dan niet een beeldlijn voorstelt. De blanking die erop volgt is opgebouwd uit een continue opvolging van 0x80 0x10, en dit voor een lengte van 268 bytes. Deze data zorgt voor de horizontale uitlijning van het beeld. De SAV code staat voor Start of Active Video en dient dus

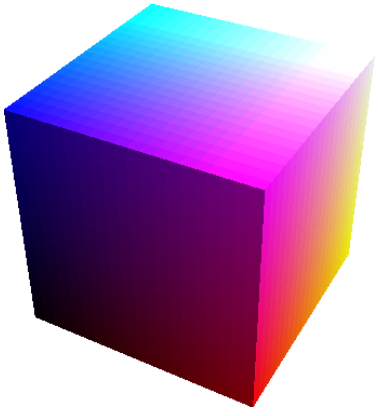
om aan te tonen dat de een AVF zal starten zodra de SAV code voorbij is. Deze heeft opnieuw een lengte van 4 bytes, bestaande uit drie vaste bytes (0xFF 0x00 0x00) gevolgd door een code die aanduidt welke data er zal volgen, zijnde welk field (even of oneven) en de blanking status. Als laatste is er dus de effectieve lijndata, met een lengte van 1440 bytes. Deze 1440 bytes stellen een volledige lijn voor, zij het even of oneven, en tellen dus 720 pixels, wat neerkomt op 2 bytes per pixel. Er is een opvolging van Cb Y Cr Y over de volledige lengte, hoe dit formaat precies gestructureerd is wordt in het volgende hoofdstuk besproken.

5.2 Kleurenruimte

5.2.1 RGB

RGB is de meest gekende en voor de hand liggende kleurenruimte, ze bestaat uit een combinatie van de 3 kleuren Rood, Groen en Blauw en is voor de gebruiker ook de meest eenvoudige manier om een kleur te omschrijven. Er bestaan echter verschillende formaten van deze RGB standaard, vaak gekozen om datacompressie toe te passen waar mogelijk. Als we RGB als een 24 bits getal zien van telkens 8 bits per kleurwaarde, kunnen we heel eenvoudig een kleur detecteren en deze aanpassen. Cfr. 5.2.3 voor de omzetting van RGB naar YCbCr en omgekeerd. Op figuur 5.3 is te zien hoe deze kleurruimte kan worden voorgesteld.

Een kleur aanpassen in deze ruimte is vrij voor de hand liggend, aangezien het een voorstelling is die we gewoon zijn. We kunnen de componenten namelijk perfect apart waarnemen en ons voorstellen. Op die manier kunnen we dan ook meteen voor ons zien wat er zal gebeuren als een aanpassing zouden doen aan één van de componenten.



FIGUUR 5.3: RGB kleurenruimte [3]

Stel dat we een RGB kleur (97,179,254) hebben, en we willen hierop een transformatie uitvoeren dan hebben we reeds alle kleuren ter beschikking. Indien we de helderheid willen opdrijven moeten we alle drie de waarden aanpassen, namelijk verhogen met eenzelfde factor. Het tegenovergestelde geldt voor het verdonkeren van de kleur.

Willen we een kleurverandering doorvoeren, zoals dit met deze opdracht het geval is met blauw en geel, blijft dit toch nog een vrij complexe operatie. In het geval we blauw willen converteren naar geel moeten we de rode en de groene component vergroten terwijl we de blauwe component drastisch laten inkrimpen. In pseudocode kan dit verwezenlijkt worden op de volgende manier:

```
1 if 0>R>115 and 100>G>200 and 0>B>255
2     R = 255
3     G = 255
4     B = 0
5 end
```

Listing 5.1: Pseudocode voor het vervangen van blauw met geel

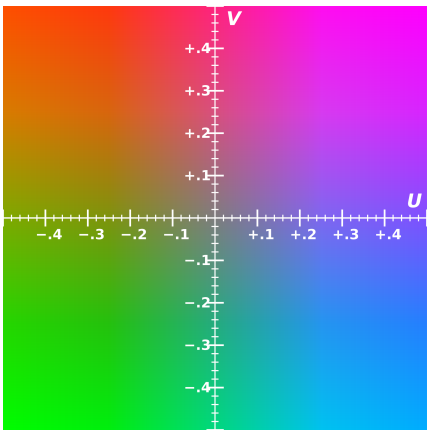
Dit levert echter een monotoon geel op, wat voor de ogen niet zo aangenaam is om naar te kijken. Daarom kiezen we om de (R,G,B) componenten geen exacte nieuwe waarde te geven, maar één die afhankelijk is van hun oorspronkelijke waarde. Dit kunnen we doen door de waarden met een vaste maat te vergroten of verkleinen, vertrekken van de waarde die de pixel reeds heeft.

```
1 if 0>R>115 and 100>G>200 and 0>B>255
2   R = 255
3   G = min(G+54, 255)
4   B = max(B-215,0)
5 end
```

Listing 5.2: Pseudocode voor het vervangen van blauw met geel met behoud van tinten

5.2.2 YCbCr

YCbCr is geen echte kleurenruimte, maar een formaat om een kleurenruimte te transporteren van een bron naar een scherm, het is gebruikt in het ITU-R BT656 formaat en is ontworpen om een zelfde kleurervaring te leveren in slechts 16 bits per pixel in plaats van 24 bij standaard RGB. Eigenlijk is het niet volledig correct om van 16 bits per pixel te spreken, aangezien we eigenlijk 32 bits per 2 pixels gebruiken [9]. Twee naast elkaar liggende pixels delen eenzelfde Cb en Cr waarde, terwijl ze elk een eigen Y-waarde bevatten. Het menselijk oog is namelijk veel gevoeliger voor veranderingen in helderheid dan voor veranderingen in kleur. Op figuur 5.4 is te zien hoe deze kleurruimte kan worden voorgesteld.



Figuur 5.4: YCbCr Kleurenruimte [4]

Het vervangen van een kleur in het YCbCr formaat is heel wat complexer omdat het kleurenbereik dat we nodig hebben om een Smurf volledig naar geel te wijzigen, zo groot is dat we andere kleuren ook mee converteren. Grijstinten zijn een voorbeeld van deze bijkomende kleuren. Het afstellen van het bereik is dus veel preciezer, maar door te precies te werk te gaan verliezen we een deel van het blauwe spectrum waardoor de kans bestaat dat niet alle blauw gezien wordt als binnen de grenzen liggend. Het algoritme ziet er in pseudocode als volgt uit:

```
1 if Y >= 130 and Y <= 180 and Cb > 150 and Cr > 50 and Cr < 100)
2   Y = 0xEB
3   Cb = 0x28
4   Cr = 0x98
5 end
```

Listing 5.3: Pseudocode voor het vervangen van blauw door geel

5.2.3 Conversie

Voor de conversie van RGB naar YCbCr en omgekeerd zijn er enkele vaste formules die gehanteerd kunnen worden. Aan de hand van de samenstelling van een lijn kunnen we dus uit 4 bytes data van het Active Video Frame 2 pixels halen. Deze waarden kunnen dan via de volgende formules [13] geconverteerd worden tussen beide formaten:

RGB to YCbCr	YCbCr to RGB
$R = Y + 1.140V$	$Y = 0.299R + 0.587G + 0.114B$
$G = Y - 0.395U - 0.581V$	$U = -0.147R - 0.289G + 0.436B$
$B = Y + 2.032U$	$V = 0.615R - 0.515G - 0.100B$

Tabel 5.1: Formules RGB en YCbCr conversie [13]

Kleurenconversie tussen deze twee ruimtes is dus prefect mogelijk in beide richtingen. In ideale omstandigheden kunnen we dus voor deze opdracht de YCbCr waardes binnenlezen voor twee naast elkaar liggende pixels, en deze converteren naar twee RGB pixels. We laten het algoritme om de blauwe kleur te veranderen in geel inwerken op beide van deze pixels, en converteren deze opnieuw naar de YCbCr ruimte om ze dan terug in het frame te plaatsen en weer uit te sturen.

```
1 R = Y + 1.140V
2 G = Y - 0.395U - 0.581V
3 B = Y + 2.032U
4
5 if 0>R>115 and 100>G>200 and 0>B>255
6   R = 255
7   G = min(G+54, 255)
8   B = max(B-215,0)
9 end
10
11 Y =  0.299R + 0.587G + 0.114B
12 U = -0.147R - 0.289G + 0.436B
13 V =  0.615R - 0.515G - 0.100B
```

Listing 5.4: Pseudocode voor een kleurconversie en -vervanging van blauw naar geel

Na wat experimenteren zijn we echter tot de conclusie gekoment dat deze ideale omstandigheden niet bestaan, of toch niet in deze situatie. De processor is namelijk niet in staat om binnen de tijd van één enkel frame over alle pixels te itereren en deze conversie uit te voeren. Aangezien het omzetten gebruikt maakt van floating point operaties neemt dit te veel tijd in beslag om op tijd klaar te zijn. Het zal dus noodzakelijk zijn om de kleurwijziging uit te voeren binnen de YCbCr kleurenruimte. Dit brengt echter heel wat nadelen met zich mee, cfr. 5.2.2 op pagina 13.

5.3 Kleurbepaling van een Smurf

De kleur van een Smurf werd aan de hand van het programma Photoshop bepaald in de RGB kleurenruimte. Een Smurf heeft geen solide kleur dus werd er een bereik bepaald waarbinnen een kleur moet vallen om als Smurfenblauw gedetecteerd te worden. Deze RGB kleurwaarden werden door middel van het programma Matlab geconverteerd naar de YCbCr kleurenruimte volgens de formules die eerder in dit hoofdstuk besproken werden. De grenzen worden weergegeven in tabel 5.2.

	Minimum waarde	Maximum waarde
Y	130	180
Cb	150	240
Cr	50	100
R	8	146
G	187	170
B	177	255

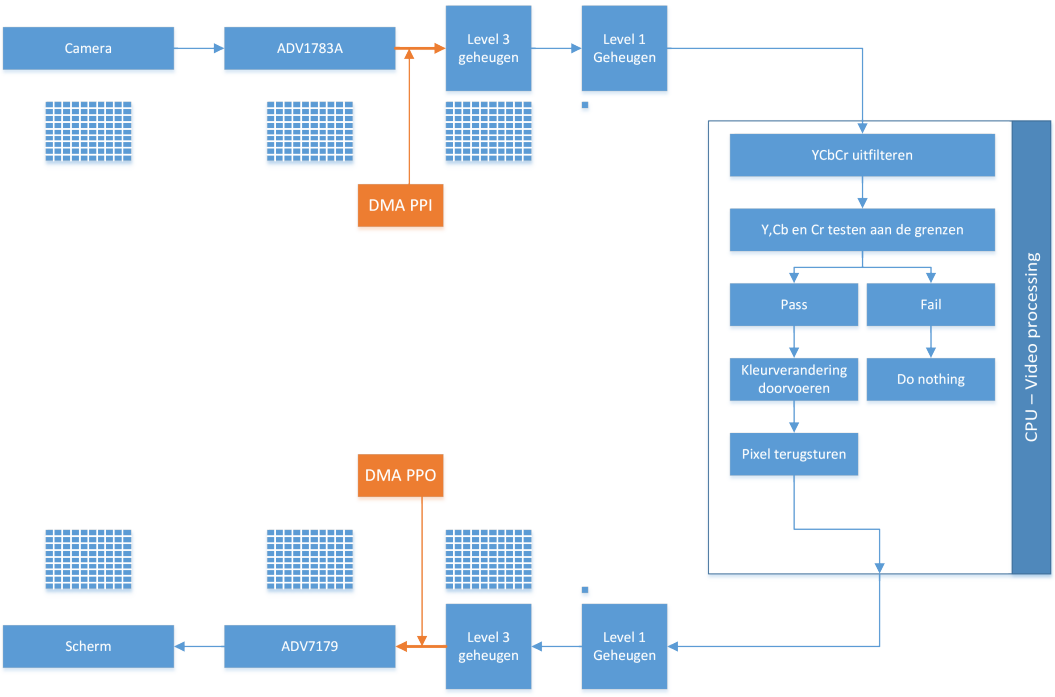
Tabel 5.2: Bereik van de huidskleurwaarden van een Smurf

Hoofdstuk 6

Implementatie

In de volgende hoofdstukken wordt besproken hoe het eindresultaat bereikt werd. De werkwijze is stapsgewijs en in elke stap wordt er verder geoptimaliseerd. In de eerste stappen was er nog sprake van een kleurenconversie van YCbCr naar RGB, maar deze werd later weggelaten om aan snelheid te winnen. In het definitieve ontwerp wordt dus volledig gewerkt in de YCbCr kleurenruimte met het oog op de snelheid van de frameprocessing.

6.1 Algemeen overzicht



FIGUUR 6.1: Algemeen overzicht van de implementatie op de BlackFin BF-561

Het algemeen opzet van de implementatie van het algoritme wordt weergegeven in figuur 6.1. Deze figuur is eveneens terug te vinden in appendix B.1. De videodata wordt binnengelezen van een digitaal fototoestel en via de video decoder die op het EZ-kit aanwezig is wordt deze videodata door middel van PPI opgeslagen in het level 3 SDRAM geheugen. Vervolgens zal door de processor telekens 1 pixel opgevraagd worden om verwerkt te worden. Deze pixel wordt dan gekopieerd naar het level 1 geheugen. Hierna wordt het verwerkingsalgoritme toegepast op de pixel. Dit algoritme zal in de hoofdstukken hieronder verder worden besproken. Indien de pixel van waarde veranderd is na toepassing van het algoritme, dan zal deze terug van het level 1 geheugen naar het level 3 geheugen gestuurd worden. Via de PPO wordt het beeld vervolgens van het level 3 geheugen naar de video encoder gezonden die op zijn beurt het videosignaal doorgeeft aan een scherm.

6.2 Frame iteratie

6.2.1 Active Video Frame

Om de AVFs te kunnen vinden, moeten we tweemaal met een verschillende offset door de array itereren. Aangezien een lijn ook nog bytes bevat waar men beter niets aan veranderd, is de eenvoudigste oplossing te itereren over de lijnen binnen een frame, en dus telkens de lijnindex te vermenigvuldigen met de lijnlengte (dit is hier 1716 bytes). In elke iteratie van deze loop verkrijgen we een nieuwe lijn. Het eerste AVF bevat de oneven lijnen, en dus zijn dit ook de eerste die we zullen bewerken.

```
#define AVSTART1 20
#define AVSTART2 281

for(int avField = 0; avField<2; avField++)
{
  int startFor = (avField==0)?AVSTART1:AVSTART2;
  int endFor = startFor+((avField==0)?AVLENGTH1:AVLENGTH2);
  for(int i=startFor; i<endFor; i++)
  {
    //...
  }
}
```

Listing 6.1: Code voor het itereren over een frame

6.2.2 Processing van een beeldlijn

Een lijn bevat eerst 276 bytes data die we niet mogen overschrijven (Cfr. 5.1.2). Pas daarna volgen de 1440 bytes data die wij moeten uitlezen en bewerken. We lezen per

iteratie 4 bytes uit omdat deze alle 4 nodig zijn om twee volledige pixels te kunnen bepalen en te converteren naar RGB. Dit zorgt er voor dat we per twee over de array lopen:

```
#define LINEOFFSET 276/2    // 4 EAV + 268 HB + 4 SAV
#define LINELENGTH 1716/2   // ITU-R BT656 line is 1716 bytes

for(j=LINEOFFSET; j<LINELENGTH; j+=2)
{
  //...
}
```

Listing 6.2: Code voor her itereren over een lijn

6.3 Kleurenconversie

6.3.1 YCbCr naar RGB

Zoals reeds aangegeven in 5.2.3, kunnen we YCbCr perfect converteren naar RGB als we beschikken over vier bytes data die respectievelijk Cb, Y1, Cr en Y2 voorstellen. De input komt binnen onder de vorm van twee shorts, waaruit CbY en CrY gehaald moeten worden. Om dit te bereiken zijn er twee operaties vereist, namelijk shiften en masken, en dit voor beide shorts. Aan de hand van de formules op pagina 14 kunnen we de 4 verkregen bytes omrekenen tot RGB waarden, of in C code:

```
char[][] YUVtoRGB(short CbY, short CrY)
{
  //First RGB by using Y from CbY
  char R1 = 1.164*( (CbY&0x00FF) - 16)
      + 1.596*( ((CrY>>8)&0x00FF) - 128);
  char G1 = 1.164*( (CbY&0x00FF) - 16)
      - 0.813*( ((CrY>>8)&0x00FF) - 128)
      - 0.391*( ((CbY>>8)&0x00FF) - 128);
  char B1 = 1.164*( (CbY&0x00FF) - 16)
      + 2.018*( ((CbY>>8)&0x00FF) - 128);

  //Second RGB by using Y from CrY
  char R2 = 1.164*( (CrY&0x00FF) - 16)
      + 1.596*( ((CrY>>8)&0x00FF) - 128);
  char G2 = 1.164*( (CrY&0x00FF) - 16)
      - 0.813*( ((CrY>>8)&0x00FF) - 128)
      - 0.391*( ((CbY>>8)&0x00FF) - 128);
  char B2 = 1.164*( (CrY&0x00FF) - 16)
      + 2.018*( ((CbY>>8)&0x00FF) - 128);

  return char[][] { {R1, G1, B1} , {R2, G2, B2} };
}
```

Listing 6.3: YUV naar RGB conversie geïmplementeert op de BF-561

We creëren een functie die de twee short als argumenten verkrijgt, en na een aantal operaties een jagged array teruggeeft met daarin twee RGB pixelwaarden. Zoals zichtbaar is moet er, om de Cb en Cr te bepalen eerst 8 bit geshift worden naar links. Dit is vereist aangezien ze werwerkt zit in de eerste 8 bits van de short. Om geen onverwachte resultaten te bekomen masken we nog eens met 0xFF zodat we zeker zijn dat de eerste bits allemaal op 0 staan en dus het resultaat niet kunnen beïnvloeden. Om Y1 en Y2 te bepalen is enkel de mask nodig die we als de laatste stap op Cb en Cr ook toegepast hebben. Als laatste combineren we de verkregen waarden in een array en geven deze terug naar waar de aanroep plaatsvond.

6.3.2 RGB naar YCbCr

RGB naar YCbCr is een heel wat complexere operatie dan de omgekeerde. Dit omdat het berekenen van Y, Cb en Cr een groter aantal operaties vergt. We maken opnieuw een functie, maar ditmaal ontvangt ze de jagged array die gegenereerd wordt in de YCbCr naar RGB conversie als argument, en geeft ze een array van 2 shorts terug, die dan de 2 pixels voorstellen in de YCbCr kleurenruimte.

```
1  short[] RGBtoYUV(char[][] RGB)
2  {
3    char Y1 = 16 + 0.257*RGB[0][0] + 0.504*RGB[0][1] + 0.098*RGB[0][2];
4    char Cb = 128- 0.148*RGB[0][0] - 0.291*RGB[0][1] + 0.439*RGb[0][2];
5
6    char Y2 = 16 + 0.257*RGB[1][0] + 0.504*RGB[1][1] + 0.098*RGB[1][2];
7    char Cr = 128+ 0.439*RGB[1][0] - 0.368*RGB[1][1] - 0.071*RGb[1][2];
8
9    short CbY = Cb;
10   CbY = CbY << 8;
11   CbY = CbY + Y1;
12
13   short CrY = Cr;
14   CrY = CrY << 8;
15   CrY = CrY + Y1;
16
17   return short[] {CbY, CrY};
18 }
```

Listing 6.4: todo

De berekening om de RGB waarden om te zetten naar YCbCr is te vinden in de tabel 5.1 op pagina 14. Deze passen we toe op de RGB waarden in de binnenkomende array. Deze array is van het formaat R,G,B,R,G,B en bevat dus telkens twee pixels. Als laatste stap moeten we de Y, Cb en Cr waarden samenvoegen tot twee shorts die respectievelijk CbY en CrY voorstellen. Om Cb toe te kennen aan de hoogste 8 bit van de short, stellen we CbY eraan gelijk en shiften we ditmaal 8 bits naar links, dus te tegenovergestelde operatie als in de YCbCr naar RGB conversie. Daarna tellen we Y erbij op om deze in de laagste 8 bits te plaatsen. Deze handeling doen we tweemaal, om beide pixels geconverteerd te hebben.

6.4 Eerste iteratie

Nu we de conversies apart bekeken hebben, en ook de iteratie over het frame volledig beschreven hebben, zijn we in staat om dit samen te voegen tot één werkend geheeld. Deze code is toegevoegd in bijlage A.1 op pagina 28. Het probleem dat zich manifesteerde bij het analyseren van deze code was dat de processing tijd per frame te lang was. Hierdoor verkregen we als uitvoer een flikkerend onstabiel beeld. Er werd gezocht naar een oplossing.

6.5 Tweede iteratie

In een tweede iteratie werd als eerste de conversie van RGB naar YCbCr en omgekeerd verwijderd uit het proces. Dit leverde al een aanzienlijke snelheidswinst op, maar in het worst case scenario waarbij getest werd met een volledig blauw beeld werd slechts 1/4 van het scherm van kleur veranderd. Er is dus meer tijd nodig om een frame volledig te verwerken. Twee mogenlijkheden boden zich aan:

1. Het toepassen van interleaving op de lijnen van het frame. Dit houd in dat in een eerste frame enkel de even lijnen verwerkt worden en in een tweede frame enkel de oneven lijnen.
2. Het vergroten van de framebuffer geeft de processor meer tijd om een frame te verwerken. Het huidige aantal buffers was 4.
3. Het samennemen van naast elkaar liggende pixels, om zo het oog te bedriegen.

Er werd gekozen voor de 2^e mogelijkheid. De buffer werd eerst vergroot naar 8 frames, maar dit bleek nog steeds niet voldoende. Wanneer een volledig blauw scherm aanlegden als videosignaal was de verkregen verwerkingstijd nog steeds te kort. Hierna werd overgeschakeld naar een buffering van 32 frames. De verwerkingstijd werd nu aanzienlijk vergroot. Als nadeel daalde de framerate met factor 4. De frame processing code aangepast zoals te zien is in bijlage A.2.

De derde mogelijkheid, die ervoor zorgt dat door het checken van een enkele pixel er meerdere pixels tegelijk aangepast worden, was perfect mogelijk en hebben we ook toegepast op deze code. Aan het resultaat was duidelijk te zien dat er een versnelling was in de verwerkingstijd, maar ook dat de resolutie van het schemr als het ware gedeeld werd door 4. Het werkt namelijk zo: er wordt geïtereerd over het frame zoals reeds gebeurde, ditmaal echter verspringt de teller steeds per 2 pixels per lijn. Op die manier wordt er dus telkens een pixel overgeslagen. Indien een pixel voldoet aan de eisen om van kleur verandert te worden, worden de naast-, onder- en rechtonderliggende pixels ook meteen vervangen. Dit wil dus ook zeggen dat we

enkel de oneven lijnen moeten doen. Zo kan de totale iteratietijd gedeeld worden door 4. De uiteindelijke resolutie is echter zo erbarmelijk dat we besloten dit niet in de uiteindelijke code te behouden.Aangezien we de code geschreven hebben, en ze ook effectief werkt, is ze te zien in bijlage A.7.

In de code in bijlage A.2 is niet alleen de kleurenconversie verdwenen, maar is ook een optimalisatie uitgevoerd om het aantal for-iteraties te verminderen. Zo worden AV field 1 en AV field 2 in eenzelfde for-iteratie overlopen. Van de 32 buffers die ter beschikking staan worden er slechts 8 overlopen om de processor meer tijd te geven om deze verwerking te voltooien. Hieraan is ook de framerate verlaging met factor 4 te wijten.

Om deze buffering te vergroten, zijn meer aanpassingen nodig dan enkel in de source files. De buffers worden namelijk op vaste adressen geplaatst om en zo ideaal mogelijk systeem te bekomen. Aangezien het aantal buffers sterk gestegen is, moet deze mapping volledig herzien worden, en dit gebeurt in de linker file. Om het overzicht te bewaren zijn enkel de regels die aangepast zijn weergegeven in bijlage A.3. Daar is te zien hoe er 32 banken voorzien worden van elk 2MB groot. De verklaring voor deze grootte is te lezen in hoofdstuk 7.2. De frames waarin de data opgeslagen wordt bevinden zich in in het level 3 geheugen, dus ook daar moeten aanpassingen gebeuren. Deze aanpassingen bevinden zich in de files L3_SDRAM.h, L3_SDRAM.c en uiteindelijk ook system.h. De bekomen code is te zien in respectievelijk bijlagen A.5, A.6 en A.4.

Bronnen: [6], [12], [14]

Hoofdstuk 7

Theoretische methodes

Een probleem dat zich kan voordoen bij de geïmplementeerde methode is dat ook de kleur van andere blauwe objecten gewijzigd wordt. Wanneer het blauw eenmaal binnen de ingestelde grenzen valt zal het van kleur veranderd worden. Om dit probleem op te lossen kan gebruik gemaakt worden van een techniek die het mogelijk maakt om objecten realtime in video te herkennen. Wanneer er dan een smurf op het beeldfragment herkend wordt zal enkel dit deel van het beeldfragment een koerswijziging ondergaan. Deze techniek brengt ook onmiddellijk een tweede voordeel met zich mee omdat niet elk beeld volledig, pixel per pixel, overlopen dient te worden om na te gaan of de pixel binnen de ingestelde kleurgrenzen past.

Wanneer men enkel werkt met de intensiteit van een pixel (RGB, YUV, YCbCr...) dan wordt dit naar aantal bewerkingen vaak groot en zwaar voor de processor wanneer men elke pixel één voor één moet doorlopen. Dit vormt een probleem wanneer er gebruik gemaakt wordt van realtime video. Om het aantal processor instructies te reduceren is dus een andere techniek nodig. Het gebruik van wavelets biedt een oplossing voor bovenstaand probleem, en maakt het tevens ook mogelijk om de objectdetectie uit te voeren.

Een andere weg die gevolgd kan worden is om wel pixel-per-pixel te werken maar de datastroom op een andere manier aan de processor aan te bieden. In de huidige implementatie wordt telkens een beeld opgevraagd uit het level 3 geheugen van de processor. Dit is een zeer dure operatie wat betreft instructies. Door slim om te springen met de DMA-controller kan men de toevoer van data versnellen en zo het aantal instructies aanzienlijk verminderen.

7.1 Objectdetectie aan de hand van Haar-like-features

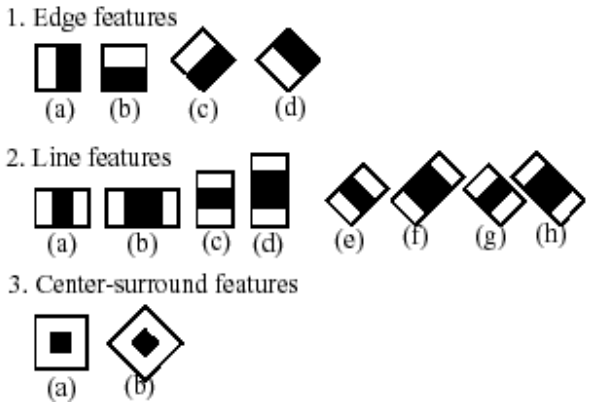
Een relatief eenvoudige techniek die gebruikt wordt om objecten te detecteren is de haar-cascade. Deze techniek maakt gebruik van Haar-like-features om objecten te detecteren. Een geïsoleerde pixel bevat enkel informatie over de luminantie en kleur van die bepaalde pixel. Het vertelt niets meer over de structuur of het uitzicht van het geheel. Om meer informatie te krijgen over het geheel dient men een aantal aangrenzende pixels te aanschouwen als een verzameling. Het Viola-Jones algoritme is een feature-based algoritme dat in tegenstelling tot een pixel-based algoritme gebruik maakt van features om objecten te detecteren. Het algoritme maakt gebruik

van Haar wavelets. Dit type wavelet laat toe om visuele patronen te beschrijven als een luminantieverandering op verschillende frequenties.

Het Viola-Jones algoritme is ook zelflerend en kan getraind worden door middel van een oefenset afbeeldingen die zowel positieve als negatieve afbeeldingen bevat. zo′n model wordt een classificator genoemd en bestaat typisch uit enkele honderdtallen positieve en enkele negatieve afbeeldingen.

Om een classificator aan te maken, wat ook wel training van een classificator genoemd wordt, wordt een wavelet transformatie uitgevoerd op de trainingsafbeelding aan de hand van een haar-like-feature. Om dit proces te versnellen wordt niet de originele afbeelding gebruikt, maar de integraalafbeelding. Een integraalafbeelding kan als volgt gegenereerd worden. De waarde van de pixel op positie x,y van de integraalafbeelding is gelijk aan de waarden van alle pixels met kleinere x- en y-waarden op de originele afbeelding. Vervolgens zullen de pixels van de integraalafbeelding getest worden door middel van een venster dat over de integraalafbeelding heenschuift. Elke feature heeft na deze fase een bepaalde wegingsfactor. Een geheel van deze wegingsfactoren omschrijft dan op zijn beurt een bepaald object.

De integraalafbeelding wordt getest met de haar-like features die weergegeven zijn in figuur 7.1 op pagina 23. Afhankelijk van de vereiste nauwkeurigheid zal al dan niet de hele set overlopen worden. Dit wordt ook wel eens haar-cascade genoemd. Een model bestaat dus uit een aaneenschakeling van verschillende haar-like-features.



Figuur 7.1: Set van haar-like features die gebruikt worden voor objectdetectie (Bron: http://docs.opencv.org)

Het Viola-Jones algoritme maakt gebruik van bovenstaand model dat opgebouwd is uit haar-like-features om te evalueren of een inkomend beeld het te detecteren object bevat. Alvorens de wavelet op een beeldfragment wordt toegepast, zal eerst een luminantie normalisatie doorgevoerd worden. Hierna zal het beeld verkleind worden tot een resolutie om het verwerkingsproces te versnellen. Vervolgens zal de

wavlettransformatie van de afbeelding, die analoog verloopt met het trainen van een model, doorlopen worden en vergeleken worden met het model om zo te evalueren of het te detecteren object op de afbeelding staat. Vervolgens kan ook de positie en grootte (onder de vorm van een rechthoekig gebied) weergegeven worden.

Wanneer er dus een smurf op de afbeelding gedetecteerd word dan pas dient er op het gedeelte van waar de smurf gedetecteerd is een kleurverandering toegepast te worden. Die detectie van smurfen kan uitgevoerd worden door Core A en de effectieve kleursverandering kan uitgevoerd worden door Core B.

Bovenstaand algoritme is relatief eenvoudig te implementeren op de BlackFin processor omdat de BlackFin Image Processing Toolbox een reeks tools bevat die speciefiek ontworpen zijn om Haar-features toe te passen. Deze toolbox is qua werking en opzet te vergelijken met de OpenCV bibliotheek.

7.2 Slim omspringen met de DMA-controller

Bij de huidige implementatie van de kleurverandering wordt een inkomend beeld telkens opgehaald vanuit het level 3 geheugen. Eén beeld heeft ongeveer een grootte van 1 á 2 MB. Met een beschikbaar geheugen van 64 MB SDRAM kunnen er dus maximum 32 frames gebufferd worden in het SDRAM. De BlackFin beschikt zelf over 32 kB level 1 cache geheugen en 128 kB level 2 cache geheugen. Het voordeel aan dit level 1 en 2 geheugen is dat zij veel sneller aanspreekbaar zijn door de processor dan het relatief trage level 3 geheugen.

Er valt op te merken dat een volledig frame niet past in het level 1 of 2 geheugen van de BlackFin. Het zou wel mogelijk zijn om een deel van een frame te verplaatsen naar het level 1 of 2 geheugen, dit te verwerken en vervolgens het resultaat weer verplaatsen naar het level 3 geheugen. Deze acties kunnen volbracht worden door de DMA-controller waardoor er geen processorkracht nodig is om deze verplaatsing teweeg te brengen. Dit principe heeft een groot voordeel qua tijd omdat de processor zich enkel moet bezighouden met het bewerken van data, en niet het opvragen ervan.

Wanneer men de frame-informatie door middel van de DMA lijn per lijn naar het level 2 geheugen kan verplaatsen heeft te processor minder tijd nodig om een lijn te verwerken. Wanneer een lijn eenmaal verwerkt is wordt deze terug naar het level 3 geheugen gestuurd door middel van de DMA-controller. Op deze manier kan mits een aanpassing van de DMA-controller toch de huidige werkwijze van kleurverandering behouden blijven en er toch een grote snelheidswinst geboekt worden. Dit zal een aanzienlijke verbetering van de framerate teweegbrengen.

Hoofdstuk 8

Besluit

We zijn vertrokken uit de template die gebruik maakt van de interframe techniek. Omdat de verwerking ons nog niet helemaal duidelijk was, zijn we daar ook bij gebleven. Het eerste wat we geprobeerd hebben is een ingangssignaal van de camera te tonen op een scherm via het BlackFin bord. Ook DMAs waren nog niet aan ons besteed. Deze zouden pas later hun nut en nood bewijzen. Hier was het grootste struikelblok het frame zelf. Want er zijn tegenstrijdige standaarden voor verschillende toestellen te vinden. Verder zijn er ook delen van de frame data die niet mogen overschreven worden. De meeste tijd in deze fase sloop in het begrijpen van de frame-indeling.

Eens we een stabiel beeld verkregen op het scherm was het tijd om een kleur te vervangen door een andere. Iedereen begrijpt dat dit inhoudt dat je kijkt naar elke pixel in het beeld en beslist of deze pixel binnen de grenzen valt van de gekozen kleur, zo ja vervang je ze, zo nee doe je er niks mee. Zoals voorzien werd code geschreven voor de kleurconversie. Toen nog in RGB. En vanaf dat we begonnen met de kleurconversie werd het beeld opnieuw onstabiel. Meer nog, het was afhankelijk van de hoeveelheid blauw die we wilden vervangen. Dus zomaar pixel na pixel verwerken was te traag.

Na wat goede raad van een oud docent, Jurgen Baert, zijn we opnieuw aan de slag gegaan, maar via een compleet nieuwe invalshoek. Namelijk, verlies geen tijd in overbodige berekeningen. We deden de kleurdetectie en -conversie zelf in RGB, terwijl het video signaal zelf YCbCr is. Dus verloren we twee keer tijd. En erg veel tijd zelfs, want de conversies tussen kleurenruimtes zijn floating point bewerkingen op een fixed point processor. Erg duur in tijd.

Een tweede idee was om meervoudige buffering toe te passen. Daarvoor zijn we dan ook eens erg minutieus de template gaan bestuderen. Initieel waren er 4 frames voorzien om te bufferen. En we hadden op verschillende plaatsen gelezen dat veel bufferen een goed idee was. Daarom werd het aantal buffers met de nodige hindernissen opgetrokken naar 32. Maar helaas niet meteen met een goed resultaat.

De betere resultaten kwamen er pas nadat we de schikking van de buffers ook zo wisten te regelen dat er nooit meer een onaangepast frame naar buiten gestuurd werd. Elk frame dat naar buiten ging werd dus verwerkt. Dit had wel een dramatische daling in framerate tot gevolg. Later werd dit echter wat weggewerkt door een betere schikking van de frameselectie voor verwerking. Er werd ook nog geoptimaliseerd

op de kleurverwerking, dit was mogelijk omdat Cb geen bovengrens nodig bleek te hebben, aangezien deze grens gewoon de maximale waarde was. Op empirische wijze zijn we dan tot de meest geschikte grenzen gekomen.

Verder is er ook een vrij steile leercurve om goed met de BlackFin borden om te leren gaan. Documentatie is moelijk tot onmogelijk te vinden. Maar hoe beter we begrijpen hoe het werkt hoe beter de resultaten zijn. We stellen dus vast dat het echt wel mogelijk is om met de BlackFin borden aan video verwerking en bewerking te doen, zoals te zien is in de code en het resultaat van onze opdracht.

Bijlagen

Bijlage A

Appendix 1

A.1 Codesnippet 1

```
#define AVLENGTH1 240
#define AVLENGTH2 240
#define AVSTART1 19
#define AVSTART2 281

#define LINEOFFSET 276/2    // 4 EAV + 268 HB + 4 SAV
#define LINELENGTH 1716/2   // ITU-R BT656 line is 1716 bytes

void process()
{
  volatile short * video_frame;

  switch(current_in_Frame)
  {
    case 0:
      video_frame = sFrame0;
      break;
    case 1:
      video_frame = sFrame1;
      break;
    case 2:
      video_frame = sFrame2;
      break;
    case 3:
      video_frame = sFrame3;
      break;
  }

  for(int avField = 0; avField<2; avField++)
  {
    for(int i=((avField==0)?AVSTART1:AVSTART2); i<((avField==0)?AVSTART1:
    AVSTART2)+((avField==0)?AVLENGTH1:AVLENGTH2); i++)
    {
      for(int j=LINEOFFSET; j<LINELENGTH; j+=2)
      {
        char[][] rgb = YUVtoRGB(video_frame[i*LINELENGTH+j], video_frame[i*
    LINELENGTH+j+1]);
        if(IsSmurfBlue(rgb[0])
          rgb[0] = ReplaceBlueWithYellow(rgb[0]);
        if(IsSmurfBlue(rgb[1])
          rgb[1] = ReplaceBlueWithYellow(rgb[1]);
        short[] yuv = RGBtoYUV(rgb);
```

```
41          video_frame[i*LINELENGTH+j] = yuv[0];
42          video_frame[i*LINELENGTH+j+1] = yuv[1];
43        }
44      }
45    }
46 }
47
48 bool IsSmurfBlue(char[] rgb)
49 {
50   if( rgb[0] > 60 && rgb[0] < 80 )
51     if( rgb[1] > 160 && rgb[1] < 180 )
52       if( rgb[2] > 240 )
53         return true;
54
55   return false;
56 }
57
58 char[] ReplaceBlueWithYellow(char[] rgb)
59 {
60   rgb[0] = 255;
61   rgb[1] = 255;
62   rgb[2] = 0;
63
64   return rgb;
65 }
66
67
68 char[][] YUVtoRGB(short CbY, short CrY)
69 {
70   //First RGB by using Y from CbY
71   char R1 = 1.164*( (CbY&0x00FF) - 16)
72       + 1.596*( ((CrY>>8)&0x00FF) - 128);
73   char G1 = 1.164*( (CbY&0x00FF) - 16)
74       - 0.813*( ((CrY>>8)&0x00FF) - 128)
75       - 0.391*( ((CbY>>8)&0x00FF) - 128);
76   char B1 = 1.164*( (CbY&0x00FF) - 16)
77       + 2.018*( ((CbY>>8)&0x00FF) - 128);
78
79   //Second RGB by using Y from CrY
80   char R2 = 1.164*( (CrY&0x00FF) - 16)
81       + 1.596*( ((CrY>>8)&0x00FF) - 128);
82   char G2 = 1.164*( (CrY&0x00FF) - 16)
83       - 0.813*( ((CrY>>8)&0x00FF) - 128)
84       - 0.391*( ((CbY>>8)&0x00FF) - 128);
85   char B2 = 1.164*( (CrY&0x00FF) - 16)
86       + 2.018*( ((CbY>>8)&0x00FF) - 128);
87
88   return char[][] { {R1, G1, B1} , {R2, G2, B2} };
89 }
90
91 short[] RGBtoYUV(char[][] RGB)
92 {
93   char Y1 = 16 + 0.257*RGB[0][0] + 0.504*RGB[0][1] + 0.098*RGB[0][2];
94   char Cb = 128- 0.148*RGB[0][0] - 0.291*RGB[0][1] + 0.439*RGb[0][2];
95
```

```
96   char Y2 = 16 + 0.257*RGB[1][0] + 0.504*RGB[1][1] + 0.098*RGB[1][2];
97   char Cr = 128+ 0.439*RGB[1][0] - 0.368*RGB[1][1] - 0.071*RGb[1][2];
98
99   short CbY = Cb;
100  CbY = CbY << 8;
101  CbY = CbY + Y1;
102
103  short CrY = Cr;
104  CrY = CrY << 8;
105  CrY = CrY + Y1;
106
107  return short[] {CbY, CrY};
108 }
```

Listing A.1: Totaalverzicht van de YUV-RGB conversie en frame-iteratie

A.2 Codesnippet 2

```
#include "main.h"
#include "video.h";

#define MIN_Y  130
#define MAX_Y  180
#define MIN_CB 150
#define MIN_CR 50
#define MAX_CR 100

#define LINEOFFSET 276/2    // 4 EAV + 268 HB + 4 SAV
#define LINELENGTH 1716/2   // ITU-R BT656 line is 1716 bytes

#define AVLENGTH 240
#define AVSTART 19
#define AVSOFFSET 262

void process()
{
  volatile short * video_frame;

  switch(current_in_Frame)
  {
    case 0:
      video_frame = sFrame0;
      break;
    case 1:
      video_frame = sFrame1;
      break;
    case 2:
      video_frame = sFrame2;
      break;
      /*
      ...
      */
    case 31:
      video_frame = sFrame31;
      break;
  }

  unsigned char Y1, Y2, Cr, Cb;
  int i, j;

  for(i=AVSTART; i<AVSTART+AVLENGTH; i++)
  {
    for(j=LINEOFFSET; j<LINELENGTH; j+=2)
    {
      Y1 = (video_frame[i*LINELENGTH+j] >> 8) & 0x00FF;
      Y2 = (video_frame[i*LINELENGTH+j+1] >> 8) & 0x00FF;
      Cb = (video_frame[i*LINELENGTH+j]) & 0x00FF;
      Cr = (video_frame[i*LINELENGTH+j+1]) & 0x00FF;

      if(Cb > MIN_CB && Cr > MIN_CR && Cr < MAX_CR)
```

```
53      {
54        if(Y1 >= MIN_Y && Y1 <= MAX_Y)
55        {
56          video_frame[i*LINELENGTH+j] = 0xEB28;
57          video_frame[i*LINELENGTH+j+1] = 0xEB98;
58        }
59
60        if(Y2 >= MIN_Y && Y2 <= MAX_Y)
61        {
62          video_frame[i*LINELENGTH+j] = 0xEB28;
63          video_frame[i*LINELENGTH+j+1] = 0xEB98;
64        }
65      }
66
67        Y1 = (video_frame[(i+AVOFFSET)*LINELENGTH+j] >> 8) & 0x00FF;
68      Y2 = (video_frame[(i+AVOFFSET)*LINELENGTH+j+1] >> 8) & 0x00FF;
69      Cb = (video_frame[(i+AVOFFSET)*LINELENGTH+j]) & 0x00FF;
70      Cr = (video_frame[(i+AVOFFSET)*LINELENGTH+j+1]) & 0x00FF;
71
72      if(Cb > MIN_CB && Cr > MIN_CR)
73      {
74        if(Y1 >= MIN_Y && Y1 <= MAX_Y)
75        {
76          video_frame[(i+AVOFFSET)*LINELENGTH+j] = 0xEB28;
77          video_frame[(i+AVOFFSET)*LINELENGTH+j+1] = 0xEB98;
78        }
79
80        if(Y2 >= MIN_Y && Y2 <= MAX_Y)
81        {
82          video_frame[(i+AVOFFSET)*LINELENGTH+j] = 0xEB28;
83          video_frame[(i+AVOFFSET)*LINELENGTH+j+1] = 0xEB98;
84        }
85      }
86    }
87  }
88 }
```

Listing A.2: Kleurenconversie met enkele iteratie

A.3 Codesnippet 3

```
1 MEM_SDRAM_BANK0  { TYPE(RAM) START(0000000004) END(0x001fffff) WIDTH(8) }
2 MEM_SDRAM_BANK1  { TYPE(RAM) START(0x00200000) END(0x003fffff) WIDTH(8) }
3 MEM_SDRAM_BANK2  { TYPE(RAM) START(0x00400000) END(0x005fffff) WIDTH(8) }
4 MEM_SDRAM_BANK3  { TYPE(RAM) START(0x00600000) END(0x007fffff) WIDTH(8) }
5 MEM_SDRAM_BANK4  { TYPE(RAM) START(0x00800000) END(0x009fffff) WIDTH(8) }
6 MEM_SDRAM_BANK5  { TYPE(RAM) START(0x00a00000) END(0x00bfffff) WIDTH(8) }
7 MEM_SDRAM_BANK6  { TYPE(RAM) START(0x00c00000) END(0x00dfffff) WIDTH(8) }
8 MEM_SDRAM_BANK7  { TYPE(RAM) START(0x00e00000) END(0x00ffffff) WIDTH(8) }
9 MEM_SDRAM_BANK8  { TYPE(RAM) START(0x01000000) END(0x011fffff) WIDTH(8) }
10 MEM_SDRAM_BANK9  { TYPE(RAM) START(0x01200000) END(0x013fffff) WIDTH(8) }
11 MEM_SDRAM_BANK10 { TYPE(RAM) START(0x01400000) END(0x015fffff) WIDTH(8) }
12 MEM_SDRAM_BANK11 { TYPE(RAM) START(0x01600000) END(0x017fffff) WIDTH(8) }
13 MEM_SDRAM_BANK12 { TYPE(RAM) START(0x01800000) END(0x019fffff) WIDTH(8) }
14 MEM_SDRAM_BANK13 { TYPE(RAM) START(0x01a00000) END(0x01bfffff) WIDTH(8) }
15 MEM_SDRAM_BANK14 { TYPE(RAM) START(0x01c00000) END(0x01dfffff) WIDTH(8) }
16 MEM_SDRAM_BANK15 { TYPE(RAM) START(0x01e00000) END(0x01ffffff) WIDTH(8) }
17 MEM_SDRAM_BANK16 { TYPE(RAM) START(0x02000000) END(0x021fffff) WIDTH(8) }
18 MEM_SDRAM_BANK17 { TYPE(RAM) START(0x02200000) END(0x023fffff) WIDTH(8) }
19 MEM_SDRAM_BANK18 { TYPE(RAM) START(0x02400000) END(0x025fffff) WIDTH(8) }
20 MEM_SDRAM_BANK19 { TYPE(RAM) START(0x02600000) END(0x027fffff) WIDTH(8) }
21 MEM_SDRAM_BANK20 { TYPE(RAM) START(0x02800000) END(0x029fffff) WIDTH(8) }
22 MEM_SDRAM_BANK21 { TYPE(RAM) START(0x02a00000) END(0x02bfffff) WIDTH(8) }
23 MEM_SDRAM_BANK22 { TYPE(RAM) START(0x02c00000) END(0x02dfffff) WIDTH(8) }
24 MEM_SDRAM_BANK23 { TYPE(RAM) START(0x02e00000) END(0x02ffffff) WIDTH(8) }
25 MEM_SDRAM_BANK24 { TYPE(RAM) START(0x03000000) END(0x031fffff) WIDTH(8) }
26 MEM_SDRAM_BANK25 { TYPE(RAM) START(0x03200000) END(0x033fffff) WIDTH(8) }
27 MEM_SDRAM_BANK26 { TYPE(RAM) START(0x03400000) END(0x035fffff) WIDTH(8) }
28 MEM_SDRAM_BANK27 { TYPE(RAM) START(0x03600000) END(0x037fffff) WIDTH(8) }
29 MEM_SDRAM_BANK28 { TYPE(RAM) START(0x03800000) END(0x039fffff) WIDTH(8) }
30 MEM_SDRAM_BANK29 { TYPE(RAM) START(0x03a00000) END(0x03bfffff) WIDTH(8) }
31 MEM_SDRAM_BANK30 { TYPE(RAM) START(0x03c00000) END(0x03dfffff) WIDTH(8) }
32 MEM_SDRAM_BANK31 { TYPE(RAM) START(0x03e00000) END(0x03ffffff) WIDTH(8) }
```

Listing A.3: Aanpassingen aan de linker file voor 32-dubbele buffering

A.4 Codesnippet 4

```
#ifndef _SYSTEM_DEFINED
#define _SYSTEM_DEFINED

#include "cdefBF561.h"
#include "ccblkfn.h"
#include <sys\exception.h>

#define TC_PER      0xFFC00B0C
#define pTC_PER     (volatile unsigned short *)TC_PER

/*************************************************************/
/***** Multipproject global variables and types       *****/
/*************************************************************/

// semaphores
extern volatile bool semaphore_frames_received;
extern volatile bool semaphoreResetVideo;

// DMA buffers for each frame
extern volatile short sFrame0[];
extern volatile short sFrame1[];
extern volatile short sFrame2[];
extern volatile short sFrame3[];
extern volatile short sFrame4[];
extern volatile short sFrame5[];
extern volatile short sFrame6[];
extern volatile short sFrame7[];
extern volatile short sFrame8[];
extern volatile short sFrame9[];
extern volatile short sFrame10[];
extern volatile short sFrame11[];
extern volatile short sFrame12[];
extern volatile short sFrame13[];
extern volatile short sFrame14[];
extern volatile short sFrame15[];
extern volatile short sFrame16[];
extern volatile short sFrame17[];
extern volatile short sFrame18[];
extern volatile short sFrame19[];
extern volatile short sFrame20[];
extern volatile short sFrame21[];
extern volatile short sFrame22[];
extern volatile short sFrame23[];
extern volatile short sFrame24[];
extern volatile short sFrame25[];
extern volatile short sFrame26[];
extern volatile short sFrame27[];
extern volatile short sFrame28[];
extern volatile short sFrame29[];
extern volatile short sFrame30[];
extern volatile short sFrame31[];

```

```
53 extern int _CORE;
54
55 /*************************************************************/
56 /***** Symbolic constants                  *****/
57 /*************************************************************/
58
59 // system constants
60 #define CLKIN   (30.0e6)    // clockin frequency in Hz
61 #define CORECLK  (600.0e6)    // core clock frequency in Hz
62 #define SYSCLK    (120.0e6)   // system clock frequency in Hz
63
64 // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
65 // ONLY ENTIRE FIELD MODE IS IMPLEMENTED AT THIS TIME
66 // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
67 #define ENTIRE_FIELD_MODE   // comment out for Active Field only
68 #define PACK_32         // comment out for unpacked DMA transfers
69
70 #ifdef ENTIRE_FIELD_MODE
71   #define PIXEL_PER_LINE  (858)
72 #else
73   #define PIXEL_PER_LINE  (720)
74 #endif
75
76 #define LINES_PER_FRAME   (525)
77
78 #define Frame_Size      (PIXEL_PER_LINE * LINES_PER_FRAME)
79
80 #endif
```

Listing A.4: Aanpassingen aan system.h voor 32-dubbele buffering

A.5 Codesnippet 5

```
#ifndef __L3_DEFINED
#define __L3_DEFINED

#include "..\system.h"

/*****************************************************************************
    */
/***** Global variables and types                    *****/
/***** All global variables are defined in main.c          *****/
/*****************************************************************************
    */

// DMA buffers for each frame
extern volatile short sFrame0[];
extern volatile short sFrame1[];
extern volatile short sFrame2[];
extern volatile short sFrame3[];
extern volatile short sFrame4[];
extern volatile short sFrame5[];
extern volatile short sFrame6[];
extern volatile short sFrame7[];
extern volatile short sFrame8[];
extern volatile short sFrame9[];
extern volatile short sFrame10[];
extern volatile short sFrame11[];
extern volatile short sFrame12[];
extern volatile short sFrame13[];
extern volatile short sFrame14[];
extern volatile short sFrame15[];
extern volatile short sFrame16[];
extern volatile short sFrame17[];
extern volatile short sFrame18[];
extern volatile short sFrame19[];
extern volatile short sFrame20[];
extern volatile short sFrame21[];
extern volatile short sFrame22[];
extern volatile short sFrame23[];
extern volatile short sFrame24[];
extern volatile short sFrame25[];
extern volatile short sFrame26[];
extern volatile short sFrame27[];
extern volatile short sFrame28[];
extern volatile short sFrame29[];
extern volatile short sFrame30[];
extern volatile short sFrame31[];

#endif
```

Listing A.5: Aanpassingen aan L3_SDRAM.h voor 32-dubbele buffering

A.6 Codesnippet 6

```
#include "L3_SDRAM.h"

// Define the DMA buffers for each frame.
// Because of SDRAM performance, each frame must be in a different bank.
// The placement in different banks is obtained by creating seperate memory bank
      placements in the ldf file!
//The NO_INIT pragma, avoids the memory initialization on a program load,
     reducing the executable loading time.

#pragma section("frame_buffer0",NO_INIT)
volatile short sFrame0[Frame_Size];
#pragma section("frame_buffer1",NO_INIT)
volatile short sFrame1[Frame_Size];
#pragma section("frame_buffer2",NO_INIT)
volatile short sFrame2[Frame_Size];
#pragma section("frame_buffer3",NO_INIT)
volatile short sFrame3[Frame_Size];
#pragma section("frame_buffer4",NO_INIT)
volatile short sFrame4[Frame_Size];
#pragma section("frame_buffer5",NO_INIT)
volatile short sFrame5[Frame_Size];
#pragma section("frame_buffer6",NO_INIT)
volatile short sFrame6[Frame_Size];
#pragma section("frame_buffer7",NO_INIT)
volatile short sFrame7[Frame_Size];
#pragma section("frame_buffer8",NO_INIT)
volatile short sFrame8[Frame_Size];
#pragma section("frame_buffer9",NO_INIT)
volatile short sFrame9[Frame_Size];
#pragma section("frame_buffer10",NO_INIT)
volatile short sFrame10[Frame_Size];
#pragma section("frame_buffer11",NO_INIT)
volatile short sFrame11[Frame_Size];
#pragma section("frame_buffer12",NO_INIT)
volatile short sFrame12[Frame_Size];
#pragma section("frame_buffer13",NO_INIT)
volatile short sFrame13[Frame_Size];
#pragma section("frame_buffer14",NO_INIT)
volatile short sFrame14[Frame_Size];
#pragma section("frame_buffer15",NO_INIT)
volatile short sFrame15[Frame_Size];
#pragma section("frame_buffer16",NO_INIT)
volatile short sFrame16[Frame_Size];
#pragma section("frame_buffer17",NO_INIT)
volatile short sFrame17[Frame_Size];
#pragma section("frame_buffer18",NO_INIT)
volatile short sFrame18[Frame_Size];
#pragma section("frame_buffer19",NO_INIT)
volatile short sFrame19[Frame_Size];
#pragma section("frame_buffer20",NO_INIT)
volatile short sFrame20[Frame_Size];
#pragma section("frame_buffer21",NO_INIT)
```

```
51 volatile short sFrame21[Frame_Size];
52 #pragma section("frame_buffer22",NO_INIT)
53 volatile short sFrame22[Frame_Size];
54 #pragma section("frame_buffer23",NO_INIT)
55 volatile short sFrame23[Frame_Size];
56 #pragma section("frame_buffer24",NO_INIT)
57 volatile short sFrame24[Frame_Size];
58 #pragma section("frame_buffer25",NO_INIT)
59 volatile short sFrame25[Frame_Size];
60 #pragma section("frame_buffer26",NO_INIT)
61 volatile short sFrame26[Frame_Size];
62 #pragma section("frame_buffer27",NO_INIT)
63 volatile short sFrame27[Frame_Size];
64 #pragma section("frame_buffer28",NO_INIT)
65 volatile short sFrame28[Frame_Size];
66 #pragma section("frame_buffer29",NO_INIT)
67 volatile short sFrame29[Frame_Size];
68 #pragma section("frame_buffer30",NO_INIT)
69 volatile short sFrame30[Frame_Size];
70 #pragma section("frame_buffer31",NO_INIT)
71 volatile short sFrame31[Frame_Size];
72 ///TOT HIER
73 volatile bool semaphore_frames_received;    // signals to core B that output
       stream can be started
74 volatile bool semaphoreResetVideo;
75 volatile unsigned char frameNumberToSend;
```

Listing A.6: Aanpassingen aan L3_SDRAM.c voor 32-dubbele buffering

A.7 Codesnippet 7

```
#include "main.h"
#include "video.h";

#define MIN_Y  130
#define MAX_Y  180
#define MIN_CB 150
#define MIN_CR 50
#define MAX_CR 100

#define LINEOFFSET 276/2    // 4 EAV + 268 HB + 4 SAV
#define LINELENGTH 1716/2   // ITU-R BT656 line is 1716 bytes

#define AVLENGTH 240
#define AVSTART 19
#define AVSOFFSET 262

void process()
{
  volatile short * video_frame;

  switch(current_in_Frame)
  {
    case 0:
      video_frame = sFrame0;
      break;
    case 1:
      video_frame = sFrame1;
      break;
    case 2:
      video_frame = sFrame2;
      break;
      /*
      ...
      */
    case 31:
      video_frame = sFrame31;
      break;
  }

  unsigned char Y1, Y2, Cr, Cb;
  int i, j;

  for(i=AVSTART; i<AVSTART+AVLENGTH; i+=2)
  {
    for(j=LINEOFFSET; j<LINELENGTH; j+=4)
    {
      Y1 = (video_frame[i*LINELENGTH+j] >> 8) & 0x00FF;
      Y2 = (video_frame[i*LINELENGTH+j+1] >> 8) & 0x00FF;
      Cb = (video_frame[i*LINELENGTH+j]) & 0x00FF;
      Cr = (video_frame[i*LINELENGTH+j+1]) & 0x00FF;

      if(Cb > MIN_CB && Cr > MIN_CR && Cr < MAX_CR)
```

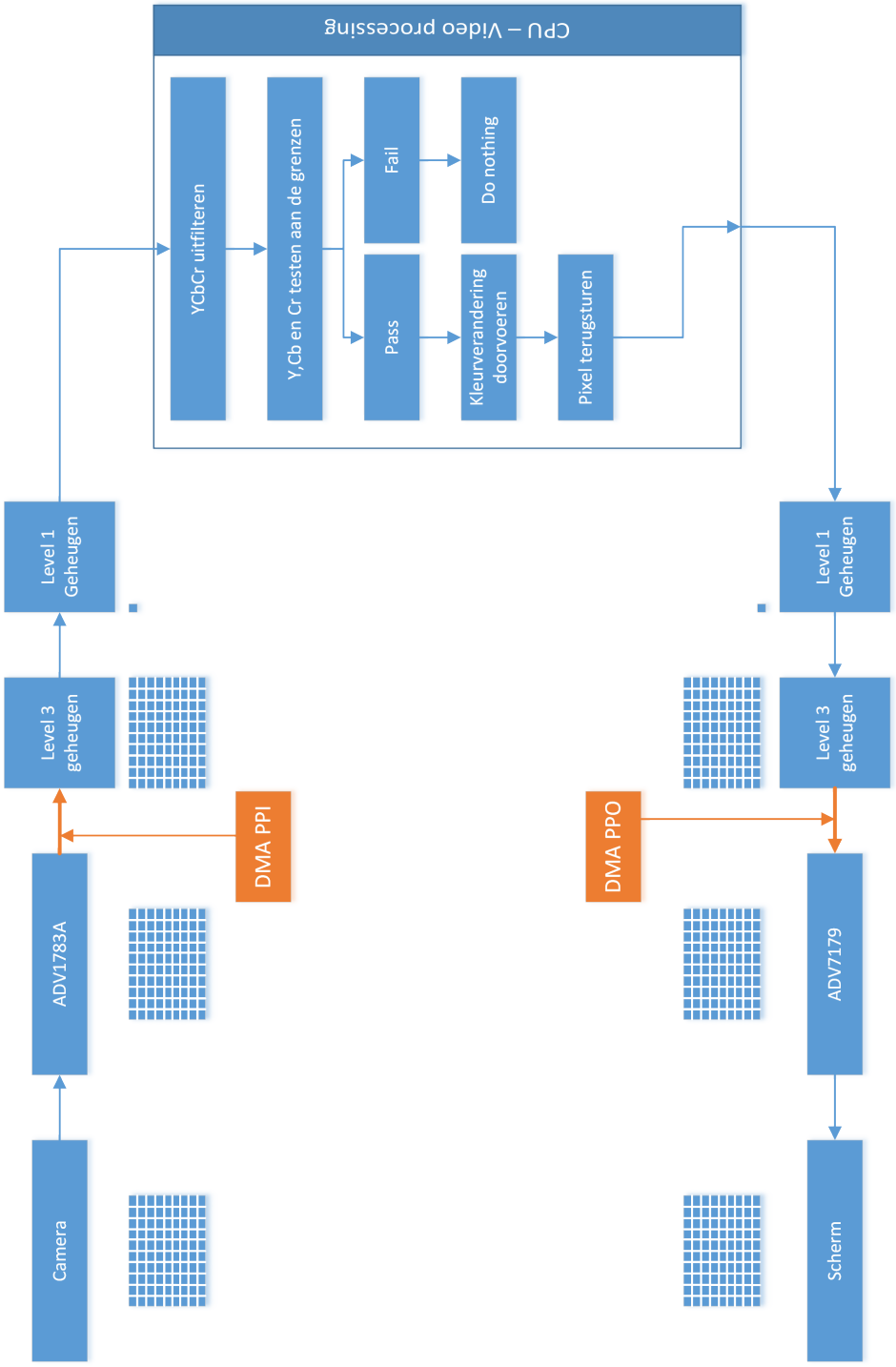
```
53      {
54        if(Y1 >= MIN_Y && Y1 <= MAX_Y)
55        {
56          video_frame[i*LINELENGTH+j] = 0xEB28;
57          video_frame[i*LINELENGTH+j+1] = 0xEB98;
58
59          video_frame[i*LINELENGTH+j+2] = 0xEB28;
60          video_frame[i*LINELENGTH+j+1+2] = 0xEB98;
61
62          video_frame[(i+AVOFFSET)*LINELENGTH+j] = 0xEB28;
63          video_frame[(i+AVOFFSET)*LINELENGTH+j+1] = 0xEB98;
64
65          video_frame[(i+AVOFFSET)*LINELENGTH+j+2] = 0xEB28;
66          video_frame[(i+AVOFFSET)*LINELENGTH+j+1+2] = 0xEB98;
67        }
68
69        if(Y2 >= MIN_Y && Y2 <= MAX_Y)
70        {
71          video_frame[i*LINELENGTH+j] = 0xEB28;
72          video_frame[i*LINELENGTH+j+1] = 0xEB98;
73
74          video_frame[i*LINELENGTH+j+2] = 0xEB28;
75          video_frame[i*LINELENGTH+j+1+2] = 0xEB98;
76
77          video_frame[(i+AVOFFSET)*LINELENGTH+j] = 0xEB28;
78          video_frame[(i+AVOFFSET)*LINELENGTH+j+1] = 0xEB98;
79
80          video_frame[(i+AVOFFSET)*LINELENGTH+j+2] = 0xEB28;
81          video_frame[(i+AVOFFSET)*LINELENGTH+j+1+2] = 0xEB98;
82        }
83      }
84    }
85  }
86 }
```

Listing A.7: Kleurenconversie met pixel skipping

Bijlage B

Appendix 2

B.1 Algemeen overzicht



FIGUUR B.1: Algemeen overzicht van de implementatie op de BlackFin BF-561

Bibliografie

[1] http://m.eet.com/media/1067259/adifigure9_big.gif.

[2] http://m.eet.com/media/1067260/adifigure10_big.gif.

[3] https://hbfs.files.wordpress.com/2010/07/rgb.png.

[4] http://upload.wikimedia.org/wikipedia/commons/thumb/f/f9/yuv_uv_plane.svg/2000px−yuv_uv_plane.svg.png.

[5] Bt.656 video interface for ics. http://www.intersil.com/content/dam/Intersil/documents/an97/an9728.pdf, 7 2002.

[6] Blackfin embedded symmetric multiprocessor. http://www.analog.com/static/imported−files/data_sheets/ADSP−BF561.pdf, 9 2009.

[7] Haar feature−based cascade classifier for object detection. http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html, 2013.

[8] Itu−r bt.656 protocol. http://techdocs.altium.com/display/FPGA/ITU−R+BT.656+Protocol, 11 2013.

[9] D. Katz and R. Gentile. Fundamentals of embedded video. http://www.edn.com/Home/PrintView?contentItemId=4017541, 10 2007.

[10] P. Menezes, J. C. Barreto, and J. Dias. Face tracking based on haar−like features and eigenfaces. Master's thesis, ISR−University of Coimbra, Portugal and LAAS−CNRS, Toulouse, France and ESA, Villafranca, Spain, unknown.

[11] S.-K. Pavani, D. Delgado, and A. F. Frangi. Haar−like features with optimally weighted rectangles for rapid object detection. *ScienceDirect*, 2009.

[12] K. Sanghai. Video templates for developing multimedia applications on blackfin processors. http://www.analog.com/static/imported−files/application_notes/EE_301.pdf, 9 2006.

[13] H. Steve. A brief introduction to digital video. http://spacewire.co.uk/video_standard.html, 2014.

[14] A. Thiriot. Goed beeld met weinig pixels. *Elektor*, 563:48−−53, 9 2010.