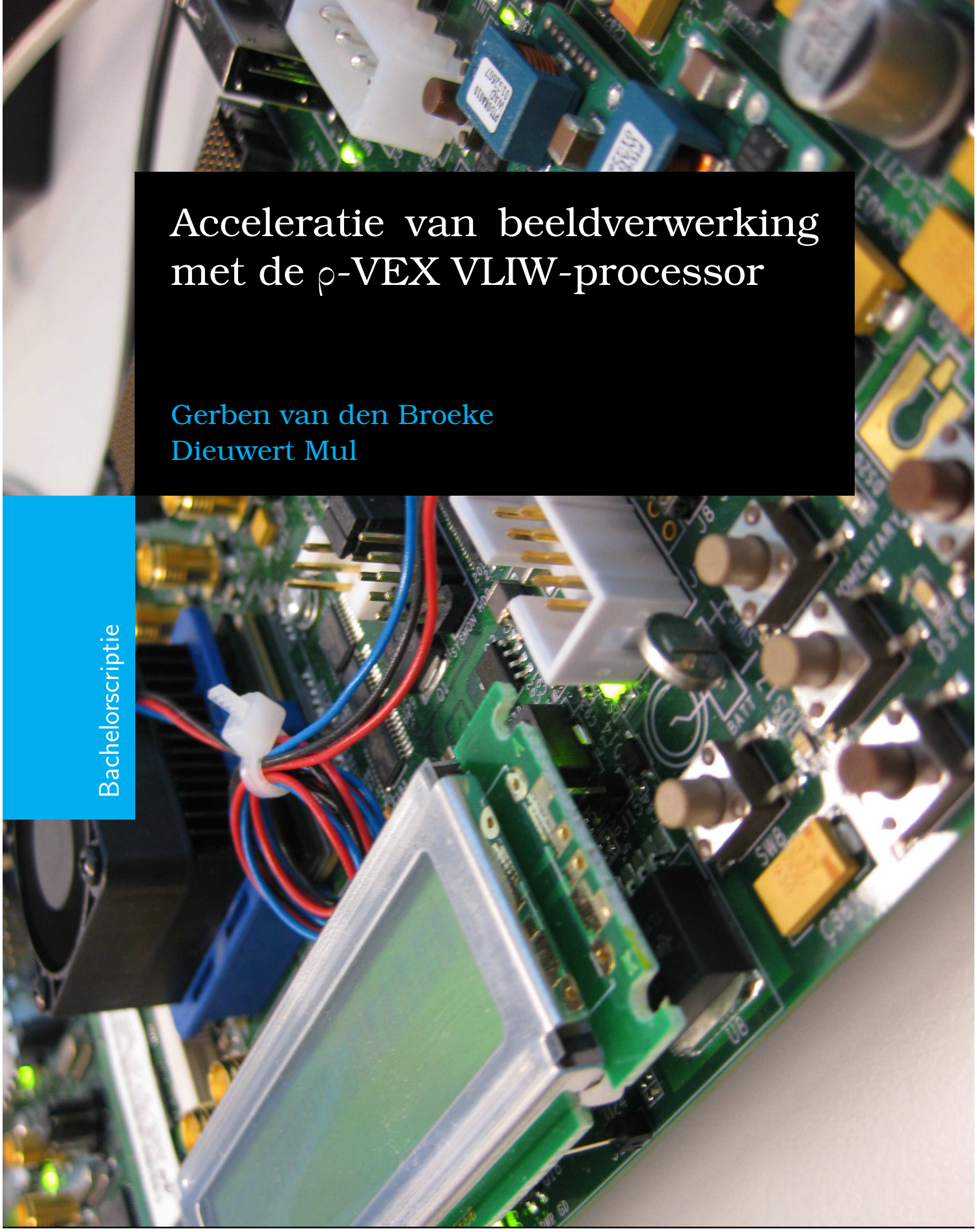


Acceleratie van beeldverwerking met de ρ-VEX VLIW-processor

Gerben van den Broeke
Dieuwert Mul

Bachelorscriptie

TU Delft Delft University of Technology

Bachelor Electrical Engineering

Acceleratie van beeldverwerking met de ρ-VEX VLIW-processor

G. van den Broeke
D.P.N. Mul

2 juli 2011

Samenvatting

Deze scriptie betreft een onderzoek naar het versnellen van een software-applicatie, gebruikmakend van een *Very Long Instruction Word*-processor als accelerator in een embedded systeem. Het gebruik van verschillende processoren binnen een computersysteem is een moderne trend waar dit onderzoek op aansluit.

De basis van dit project is de ρ-*VEX*, een reconfigureerbare VLIW-processor, ontwikkeld op de afdeling Computer Engineering van de TU Delft. De VLIW-architectuur biedt de mogelijkheid om operaties parallel uit te voeren, een eigenschap die vooral goed benut kan worden in software-applicaties die veel data verwerken, bijvoorbeeld bij Digital Signal Processing.

Het uiteindelijke doel van dit project was om een prototype te maken van een demonstrator voor de ρ-VEX. Met deze demonstrator moet zichtbaar worden gemaakt hoe met de ρ-VEX een applicatie kan worden versneld, en hoeveel snelheidswinst dit kan opleveren. Om een concrete casus te maken is een toepassing gezocht waarin het gebruik van de ρ-VEX een interessante optie is. De gekozen toepassing is beeldverwerking in digitale camera's, waar de parallelle rekenkracht, de configureerbaarheid en energie-efficiëntie van de ρ-VEX goed van pas kunnen komen.

In dit onderzoek wordt een bestaande beeldverwerkings-applicatie genomen, namelijk een applicatie die JPEG-afbeeldingen decodeert naar bitmaps. Middels een co-design-aanpak zijn de hardware en software op elkaar aangepast om de versnelling te maximaliseren.

De gekozen aanpak is om in een FPGA een MicroBlaze-processor en de ρ-VEX te zetten. De applicatie wordt uitgevoerd op de MicroBlaze, die een deel van de berekeningen kan uitbesteden aan de ρ-VEX. Om de applicatie te porten naar dit platform is eerst de applicatie-code aangepast, waarbij simpele file i/o is geïmplementeerd. Een profiel van de applicatie is gemaakt om de tijdsverdeling tussen de functies te analyseren. Vervolgens is met simulaties de beste hardware-configuratie voor de ρ-VEX gezocht. Een issue width van vier is gekozen, want de extra winst bij nog hogere issue width bleek verwaarloosbaar. Ook is een keuze gemaakt voor de taakverdeling tussen de twee processoren. Hierbij bleek de beste optie om de hele decodering op de accelerator uit te voeren. Vervolgens zijn extra aanpassingen aan de software gedaan, om deze zo goed mogelijk gebruik te laten maken van de VLIW-processor. De VEX-compiler bleek vaak inefficiënte code te genereren. Door handmatig loop unrolling toe te passen bleek het parallellisme, en dus de snelheid, significant te kunnen worden verhoogd. In totaal is met de optimalisaties de snelheid 20% verhoogd.

De resulterende code is uitgevoerd op het gemaakte platform van MicroBlaze met ρ-VEX, en ter vergelijking ook op enkel een MicroBlaze processor. Door problemen met de ρ-VEX is het niet gelukt de applicatie te laten werken op het platform met accelerator. Met behulp van een VEX-simulatie is toch een schatting gemaakt van de haalbare versnelling. Om de versnelling te berekenen is rekening gehouden met de tijd die verloren gaat bij de communicatie tussen de processoren. Door met de ρ-VEX de applicatie te accelereren, is een totale versnelling van een factor zes bereikt.

Inhoudsopgave

Lijst met afkortingen

ALU	Arithmetic Logic Unit
BMP	Bitmap
CISC	Complex Instruction Set Architecture
CPU	Central Processing Unit
(I)DCT	(Inverse) Discrete cosinustransformatie
DFT	Discrete Fouriertransformatie
DSP	Digital Signal Processing
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FU	Functional Unit, bijvoorbeeld een ALU of MUL
ILP	Instructie-level parallellisme
JPEG	Joint Photographic Experts Group
MCU	Minimal Coding Unit, blok van 8×8 pixels
MUL	Multiplier
RGB	Rood-groen-blauw, representatie voor afbeelding
RISC	Reduced Instruction Set Computing
SDK	Software Development Kit
UART	Universal Asynchronous Receiver/Transmitter
VEX	VLIW Example
VHDL	Very-high-speed-integrated-circuits Hardware Description Language
VLIW	Very Long Instrucion Word
YUV	Kleurrepresentatie met componenten luma (Y) en chroma (U en V)

Voorwoord

Deze scriptie is geschreven in het kader van het Bachelor afstudeerproject van de studie Elektrotechniek aan de TU Delft, met als projectnaam 'Acceleration of embedded applications using a VLIW processor'. Het project omvat het ontwerp van een demonstrator van een VLIW-processor die aan de TU Delft ontwikkeld is. Deze processor moet een antwoord bieden op de vraag naar meer en snellere signaalverwerking binnen de embedded markt. De demonstrator die tijdens dit project ontwikkeld is, zal de voordelen van deze VLIW-processor kunnen tonen.

De keuze van de toepassing die uitgewerkt zal worden op de VLIW-processor is te lezen in hoofdstuk 2. Voor achtergrondinformatie over de gebruikte technieken en een breder wetenschappelijk beeld wordt de lezer verwezen naar hoofdstuk 3. In hoofdstuk 4 en 5 zijn de applicatie en de gebruikte hardware uitgebreid beschreven. In hoofdstuk 6 worden de stappen genomen om de software en hardware te combineren, en worden de resultaten van het onderzoek verkregen.

Onze dank gaat uit naar Roël Seedorf, voor de flexibiliteit waarmee hij tijd voor ons wist vrij te maken, de goede begeleiding en technische hulp, en naar Anthony Brandon en Fakhar Anjam voor de technische ondersteuning. Speciaal bedanken wij Koen Bertels voor het inspringen en de organisatorische hulp.

Gerben van den Broeke en Dieuwert Mul, juni 2011

Hoofdstuk 1

Inleiding

De ontwikkeling van processoren is al jaren gericht op het sneller en efficiënter maken van computersystemen. Een groot deel van deze computers is verwerkt in *embedded systems*, waaronder telefoons, wasmachines en digitale camera's. In hedendaagse embedded systems wordt steeds meer data verwerkt. Dit is bijvoorbeeld zichtbaar in de groeiende hoeveelheid apparaten die in staat zijn beeld en geluid te verwerken en te netwerken; een mobiele telefoon met camera die gezichten kan herkennen is al geen uitzondering meer. De hieruit volgende vraag naar alsmaar snellere processoren creëert de noodzaak vernieuwende ontwerpen te ontwikkelen en toe te passen. Er wordt daarom veel onderzoek gedaan naar betere processoren. Deze scriptie beschrijft een onderzoek naar de toepassing van een nieuwe processor, de ρ-*VEX*.

Aanleiding

De ρ-VEX is een implementatie van de *VEX*-architectuur, en wordt ontwikkeld aan de TU Delft. Deze processor is bedoeld voor embedded systems die veel data moeten verwerken. Het gaat hierbij om een zogenaamde *Very Long Instruction Word (VLIW)*-processor, een processorontwerp dat extra parallellisme mogelijk maakt. Door operaties in een programma parallel te verwerken kan bij veel applicaties een snelheidswinst worden behaald. Doordat veel eigenschappen van de processor configureerbaar zijn, is het een interessante optie voor zowel grote als kleine embedded systems.

De ontwikkeling van de ρ-VEX is in een ver stadium. De processor is reeds getest met verschillende applicaties, maar een demonstrator is nog niet ontwikkeld. Om verdere interesse te wekken bij de markt is het nodig de kwaliteiten van deze processor door middel van een casus te kunnen demonstreren. Deze casus moet representatief zijn voor hoe potentiële afnemers van de ρ-VEX deze processor zouden gebruiken, om zo een beeld te geven van wat de processor voor hen kan betekenen.

Projectdefinitie

De vraag naar een demonstrator voor deze processor is de aanleiding geweest voor dit project. Deze demonstrator zal illustreren welke snelheidswinst te behalen valt met behulp van de ρ-VEX. Het doel van dit project is een voorbeeldtoepassing te ontwikkelen die de kwaliteiten van de ρ-VEX uitbuit en de meerwaarde van deze processor overtuigend demonstreert. Onderzocht wordt hoe een bestaande applicatie het best kan worden uitgevoerd, gebruikmakend van de ρ-VEX.

Er zal een toepassing worden gekozen worden die goed aansluit op de mogelijkheden van de ρ-VEX, en duidelijk kan tonen wat met de ρ-VEX kan worden bereikt. Hierdoor moet de demonstrator kunnen fungeren

als een voorbeeld voor andere toepassingen, en de kwaliteiten van de ρ-VEX goed benadrukken.

Aanpak

Allereerst zal de toepassing worden gekozen die als casus zal worden gebruikt. Bestaande applicatie-code zal dan worden gezocht, die voor deze toepassing gebruikt zou kunnen worden. De volgende stap is om deze applicatie op een gangbaar embedded platform te laten werken. Als platform wordt de MicroBlaze-processor van Xilinx gebruikt, geprogrammeerd in een FPGA. Vervolgens wordt de ρ-VEX toegevoegd aan dit platform, waar deze als *accelerator* voor de MicroBlaze wordt gebruikt. Uit zal worden gezocht hoe deze applicatie met behulp van de ρ-VEX kan worden versneld. Ten slotte zal geanalyseerd worden welke resultaten te behalen vallen met het ontwikkelde systeem. De applicatie wordt op de combinatie van MicroBlaze en ρ-VEX uitgevoerd, en de snelheid hiervan wordt vergeleken met het uitvoeren op enkel een MicroBlaze-processor.

Om deze versnelling te bereiken worden keuzes gemaakt voor de hardware-configuratie en de taakverdeling binnen de hardware, en worden aanpassingen gedaan om de software te optimaliseren voor de hardware. Hiervoor zal worden uitgezocht welke configuratie van de ρ-VEX het meest geschikt is voor deze applicatie. Uitgezocht wordt hoe de code het best kan worden verdeeld tussen de MicroBlaze en ρ-VEX. Ook wordt bekeken hoe de code het best kan worden geoptimaliseerd om de ρ-VEX maximaal te benutten.

Onderstaande stappen zullen worden genomen in dit onderzoek:

1. Een toepassing wordt gekozen, en een software-applicatie wordt daarbij gezocht.
2. De applicatie wordt bestudeerd, en getest direct op de pc en in een VEX-simulator.
3. De applicatie wordt gestript om op de MicroBlaze en ρ-VEX te kunnen draaien.
4. Een MicroBlaze-platform wordt gegenereerd, de applicatie wordt hierop getest.
5. Aan de hand van simulaties wordt een keuze gemaakt voor de configuratie van de ρ-VEX.
6. Een platform wordt gemaakt bestaande uit de ρ-VEX en MicroBlaze.
7. De applicatie wordt gepartitioneerd over deze twee processoren.
8. De applicatie wordt geoptimaliseerd om de mogelijkheden van de ρ-VEX te benutten.
9. Er wordt gemeten hoeveel versnelling de ρ-VEX kan leveren op het uitvoeren van de applicatie.

Overzicht scriptie

De indeling van deze scriptie is als volgt. Hoofdstuk 2 zal de keuze voor de toepassing van de ρ-VEX behandelen. In hoofdstuk 3 wordt achtergrondinformatie over de gebruikte technieken gegeven, en worden vergelijkbare onderzoeken beschouwd. Hoofdstuk 4 beschouwt de gebruikte applicatie en de gemaakte aanpassingen daaraan. Hoofdstuk 5 behandelt de gebruikte computerarchitectuur en gekozen configuraties. In hoofdstuk 6 wordt de applicatie met de processoren gecombineerd, worden optimalisaties gedaan om de applicatie maximaal te versnellen, en worden de resultaten van de versnelling beschouwd.

Hoofdstuk 2

Keuze van toepassing voor ρ-VEX

De eerste stap in dit project is de keuze van de toepassing voor de demonstrator. Hierop zal in dit hoofdstuk in worden gegaan. Er zijn verschillende toepassingen waarvoor het gebruik van de ρ-VEX interessant kan zijn. De ρ-VEX is een programmeerbare, geparametriseerde *Very Long Instruction Word (VLIW)*-processor. De bijzondere kwaliteit van VLIW-processoren ligt bij het parallel uitvoeren van operaties. Afhankelijk van de gekozen configuratie kunnen op de ρ-VEX tussen 2 en 8 operaties parallel uitgevoerd worden. Doordat de ρ-VEX geparametriseerd is kan de configuratie simpel aangepast worden.

Het moet binnen de applicatie uiteraard wel mogelijk zijn om meerdere berekeningen tegelijk uit te laten voeren. Niet elke applicatie heeft de mogelijkheid om effectief gebruik te maken van deze mogelijkheid voor parallellisme. Voor de demonstrator dient een applicatie gebruikt te worden waarbinnen veel berekeningen onafhankelijk van elkaar zijn. Een belangrijke groep applicaties ligt in het gebied van de *Digital Signal Processing (DSP)*.

Paragraaf 2.1 beschrijft de eisen die gesteld worden aan de demonstrator. Vervolgens zal paragraaf 2.2 verschillende opties behandelen voor de toepassing die uitgewerkt wordt het project. In de laatste paragraaf (2.3) zal ingegaan worden op de keuze die gemaakt is.

2.1 Eisen aan demonstrator

Op basis van de projectbeschrijving is een programma van eisen voor de demonstrator opgesteld. Het project gaat uit van de ρ-VEX en heeft als doel daar een kansrijke toepassing voor te ontwikkelen. Zoals gesteld ligt het voor de hand een product te zoeken binnen DSP-toepassingen. Hierbinnen zijn nog vele mogelijkheden. Om een goede afweging te kunnen maken zijn hieronder de vijf belangrijkste eisen die aan de demonstrator worden gesteld opgesomd:

- De demonstrator maakt gebruik van de kwaliteiten van de ρ-VEX.
- De verbeteringen door de processor moeten duidelijk meetbaar zijn.
- De invloed van processor moet duidelijk zichtbaar zijn.
- De werking van het prototype moet zo min mogelijk afhankelijk zijn van externe factoren.
- Na verdere ontwikkeling moet de toepassing interessant zijn voor de markt.

Deze eisen zullen in de volgende alinea's elk apart behandeld worden.

Maakt gebruik van kwaliteiten van de ρ-VEX

Niet elke applicatie benut de mogelijkheden van de ρ-VEX even goed. Om optimaal gebruik te maken van de ρ-VEX moeten de berekeningen in de applicatie goed parallelliseerbaar zijn. Hiervoor moeten er veel operaties aanwezig zijn die niet afhankelijk zijn van de uitkomst van de vorige operatie. Binnen de *Digital Signal Processing (DSP)* is hier bij veel applicaties sprake van. Vaak dienen de stappen binnen het algoritme immers voor elk deel van het signaal opnieuw gedaan te worden. Tussen deze databewerkingen bestaat dan geen afhankelijkheid, wat het parallelliseerbaar maakt.

De toepassing van de ρ-VEX wordt gezocht in draagbare apparaten. Wanneer het energieverbruik een kleine of geen rol speelt zijn er processoren beschikbaar die door een hogere kloksnelheid en andere architectuur betere prestaties leveren. Het energieverbruik van de ρ-VEX is afhankelijk van de configuratie, maar kan laag genoeg zijn om te gebruiken in apparaten die door een batterij worden gevoed[1].

Verbeteringen door de processor moeten duidelijk meetbaar zijn

Belangrijk is dat de prestatie van de ρ-VEX goed te vergelijken valt met die van andere processoren. Een vergelijking tussen processoren moet kunnen worden gemaakt waarbij andere componenten en omstandigheden zo veel mogelijk gelijk kunnen blijven.

Invloed van de processor moet duidelijk zichtbaar zijn

Om een goede demonstrator te maken, moet bij het uitvoeren van de applicatie de invloed van de processor naast goed meetbaar, ook goed zichtbaar zijn. Daarom wordt een toepassing gezocht waarvan duidelijk zichtbaar is wat deze doet en welk aandeel de processor daarin heeft. Dit betekent in dit geval dat het een voordeel is als de processor niet een klein onderdeel is van een groter systeem. Daarbij zou immers de kwaliteit van de demonstrator van meer factoren afhankelijk zijn dan van de processor. Wanneer de toepassing gedemonstreerd wordt is in dat geval lastiger te zien waar in hoeverre de kwaliteit van de processor het product beïnvloedt.

Werking van prototype moet zo min mogelijk afhankelijk zijn van externe factoren

Wanneer de werking afhankelijk is van externe factoren kunnen deze de beperkende factor zijn binnen het systeem. Hierdoor is het niet goed meet- en herkenbaar hoeveel winst geboekt wordt door gebruik te maken van de ρ-VEX. Ook moet de applicatie niet afhangen van veranderlijke invoerdata, omdat de reproduceerbaarheid van metingen daarmee wordt belemmerd. Deze eis komt gedeeltelijk voort uit de twee bovenstaande eisen aan de meetbaarheid en zichtbaarheid.

Na verdere ontwikkeling interessant voor de markt

Het doel van de demonstrator is niet alleen het technische voordeel te demonstreren, maar ook te illustreren wat een uiteindelijke toepassing zou kunnen zijn die interessant is voor de markt. Er zijn veel toepassingen waar de ρ-VEX wel goed zou scoren, maar die voor de markt niet interessant zijn, bijvoorbeeld omdat andere processoren reeds aanwezig zijn die ook een hoge snelheid kunnen bereiken. Ook is het handig een welbekende markt te kiezen. Bij specialistische toepassingen zou de demonstratie minder goed overkomen op mensen die niet bekend zijn met de toepassing.

2.2 Mogelijke toepassingen

Er zijn verschillende, mogelijk interessante toepassingen overwogen. De drie meest kansrijke toepassingen waren *Software Defined Radio (SDR)*, *Beeldverwerking in digitale camera's*, en *vingerafdrukherkenning*. Deze paragraaf zal deze toepassingen behandelen en afsluiten met een samenvattende afweging in de vorm van een tabel.

Software Defined Radio

De processor is in principe geschikt voor SDR. In SDR worden veel frequentietransformaties en filters toegepast. Deze dataverwerkingen kunnen goed geparallelliseerd worden. Aan de eerste eis waar een goede demonstrator aan moet voldoen is hiermee voldaan. Ook de reconfigureerbaarheid van de ρ-VEX kan voordelen bieden voor SDR. Binnen dit project zal slechts ruimte zijn om een ontvanger te bouwen die één protocol ondersteunt, bijvoorbeeld AM radio. Verder uitgewerkt zou de processor meerdere protocollen kunnen ondersteunen. In dat geval zou de reconfigureerbaarheid een positieve uitwerking kunnen hebben op de prestaties van de processor. De flexibiliteit in combinatie met de snelheid van de processor worden op deze manier goed benut.

Echter, er zijn ook nadelen bij gebruiken van SDR als toepassing voor de demonstrator. Om een werkend systeem te krijgen, zullen altijd een ontvanger en andere analoge elektronica nodig zijn. Hierdoor wordt de ρ-VEX verborgen in een groter systeem. De kwaliteit van het systeem hangt af van meer factoren dan alleen de processor en de applicatie. Hierdoor is de invloed van de processor zelf minder duidelijk zichtbaar.

Ook zijn de verbeteringen minder goed meetbaar. De kwaliteit is immers ook afhankelijk van andere onderdelen en ingangssignaal is veranderlijk. Een bijkomend probleem, dat echter niet zozeer wat te maken heeft met de geschiktheid als demonstrator, maar wel met de geschiktheid voor dit project, is dat de kosten van SDR niet zeker binnen het budget vallen. Ook zou het te veel werk kunnen worden voor de beschikbare tijd. Hierbij speelt mee dat er geen specifieke kennis is van SDR op de afdeling. Dit is bij de andere toepassingen wel het geval.

Voor de markt is Software Defined Radio interessant voor bijvoorbeeld toepassingen in smartphones. Doordat smartphones veel verschillende protocollen ondersteunen, en de trend is dat ook steeds meer andere draagbare apparaten meerdere protocollen ondersteunen, zijn er volop kansen in de markt voor deze toepassing van de ρ-VEX.

Beeldverwerking in digitale camera's

Een andere mogelijk interessante toepassing van de processor is beeldverwerking in digitale camera's. Hierbij kan gedacht worden aan het snel opslaan en weergeven van foto's. Ook zouden vervolgens functies kunnen worden toegevoegd die bewerking van de foto's mogelijk maken, bijvoorbeeld belichtingscorrectie en digitaal focusseren.

Het voordeel van SDR, namelijk de aansluiting van de applicatie op de kwaliteiten van de processor, geldt ook bij deze toepassing. Omdat beeldverwerking een ruim begrip is omvat het meerdere applicaties, en per applicatie kan de geschiktheid verschillen. Omdat bij de meeste bewerkingen veel operaties onafhankelijk van elkaar kunnen worden uitgevoerd, bijvoorbeeld op elke pixel apart, zijn de meeste beeldbewerkingsapplicaties geschikt. Voor dit project is vanwege de beschikbare tijd slechts de implementatie van één applicatie in beeld. Wanneer meerdere applicaties op de ρ-VEX uitgevoerd zullen worden, zal door de programmeerbaarheid tussen de applicaties kunnen worden gewisseld, en zal de reconfigureerbaarheid van de processor ervoor zorgen dat voor elke applicatie de meest efficiënte configuratie kan worden gebruikt.

Voor de applicatie ligt het meest voor de hand om het decoderen van een JPEG-afbeelding te implementeren. Bij dit decoderingsproces worden verschillende stappen gedaan, zodat wordt getoond dat de processor voor verschillende algoritmes goed werkt. Ook is er specifieke kennis van JPEG-decoderen aanwezig op de afdeling, wat van pas zou kunnen komen bij eventuele problemen.

Bij het decoderen is sprake is van een zichtbaar eindresultaat, namelijk de afbeelding. Het implementeren van beeldbewerkingen zou natuurlijk nog beter zichtbaar zijn, dit zou later aan de demonstrator kunnen worden toegevoegd. Deze toepassing is niet afhankelijk van verdere electronica of andere factoren. Hierdoor komt de invloed van de processor op het proces goed naar voren. Deze optie sluit dus goed aan op de eis dat de invloed van de processor duidelijk zichtbaar moet zijn.

De winst is bij deze toepassing goed meetbaar. Het decoderen van de afbeelding kan altijd gedaan worden, en altijd zal hetzelfde resultaat verkregen worden. Door dezelfde afbeelding op een andere processor te decoderen kan een eenvoudige en duidelijke vergelijking gemaakt worden.

De laatste eis is dat de toepassingen kansen moet hebben op de markt. Huidige digitale camera's worden uitgerust met steeds meer mogelijkheden voor beeldbewerking. Met de ρ-VEX kan deze beeldbewerking versneld worden en kunnen de mogelijkheden uitgebreid worden. Het nut van deze toepassing is ook voor iedereen duidelijk.

Vingerafdrukherkenning

De laatste onderzochte mogelijkheid is een toepassing voor vingerafdrukherkenning. Ook deze toepassing zal goed gebruik kunnen maken van de mogelijkheden van extra parallellisme.

Het systeem hangt grotendeels van de processor af. Slechts in het geval van real-time herkenning zou er gebruik moeten worden gemaakt van externe elektronica om de vingerafdruk uit te lezen. Binnen dit project zou er slechts tijd zijn om de demonstrator zo ver te implementeren dat twee afbeeldingen van vingerafdrukken met elkaar vergeleken worden. Hiervoor kunnen twee vingerafdrukken gebruikt worden die al in het systeem geprogrammeerd zijn. Op deze manier voldoet deze toepassing, net als de beeldverwerking in digitale camera's, aan de eisen wat betreft meetbaarheid en onafhankelijkheid van externe factoren.

De zichtbaarheid van de demonstrator is iets minder goed dan bij de beeldverwerking, omdat niet zo duidelijk is wat voor verwerking wordt uitgevoerd. Bij beeldverwerking is het makkelijker om in te schatten hoeveel berekeningen er nodig zijn om het resultaat te verkrijgen, terwijl bij vingerafdrukherkenning meer kennis van de algoritmes nodig is om te begrijpen wat er precies gebeurt.

Er is een vermoeden dat de reconfigureerbaarheid van de ρ-VEX beter uitkomt in de beeldverwerkingstoepassing, waarbij verschillende bewerkingen kunnen profiteren van andere configuraties. Bij vingerafdrukherkenning is reconfigureren waarschijnlijk moeilijker te demonstreren. Een voordeel van deze toepassing is wel dat deze al eens getest is op de ρ-VEX[2].

De markt voor vingerafdrukherkenning is vermoedelijk iets kleiner dan voor de andere twee voorbeelden. In de toekomst zijn er echter toepassingen te bedenken voor bijvoorbeeld draagbare apparaten die uitgerust zijn met een vingerafdrukslot. Hiervoor worden dan uiteraard hoge eisen gesteld aan de snelheid van de herkenning, wat het interessant maakt om te versnellen. Een bijkomend klein minpunt van vingerafdrukherkenning kan zijn dat het negatievere associaties opwekt dan de andere twee toepassingen, omdat het vaak geassocieerd wordt met privacy-inbreuk. Dit maakt het ook minder interessant als demonstrator.

In tabel 2.1 is een samenvatting te vinden van de voor- en nadelen van de verchillende toepassingen. De Software Defined Radio heeft als voornaamste nadeel dat het afhankelijk is van externe factoren. Beeldverwerking en vingerafdrukherkenning hebben beide geen minpunten en zijn beide een geschikte kandidaat.

Tabel 2.1: Verschillende toepassingen getoetst aan eisen aan demonstrator

	Software Defined Radio	Beeldverwerking	Vingerafdruk-herkenning
Gebruik van kwaliteiten van ρ-VEX	+	+	+/±
Meetbaarheid verbeteringen	±	+	+
Zichtbaarheid invloed	±	+	±
Onafhankelijkheid externe factoren	−	+	+
Interessant voor markt	+	+	±

2.3 Gekozen toepassing

Aan de hand van de vorige paragraaf is de conclusie getrokken dat de toepassing beeldverwerking in digitale camera's het beste lijkt aan te sluiten op de eisen die binnen dit project zijn gesteld. De kwaliteiten en de invloed van de processor worden meer dan bij de andere toepassingen zichtbaar en meetbaar getoond. Belangrijk voor de keuze van de toepassing, was dat deze binnen een domein valt waar de ρ-VEX voor ontworpen is: Digital Signal Processing. Bij het decoderen en bewerken van afbeeldingen kan veel parallellisatie worden toegepast omdat veel bewerkingen op elke pixel worden uitgevoerd. Het gebruik van een VLIW-processor kan hier dus voordelig zijn.

In dit onderzoek zal slechts één beeldverwerkingsapplicatie worden onderzocht. Hiervoor is het decoderen van een JPEG-afbeelding gekozen. Voor een volledige demonstrator is het de bedoeling dat ook andere applicaties worden gebruikt. Bij het versnellen van beeldbewerkingen zal het parallellisme van de ρ-VEX zelfs nog beter van pas komen dan bij de JPEG-decoder.

JPEG is een complex bestandsformaat. Bij het decoderen van JPEG moeten verschillende stappen worden uitgevoerd, zoals beschreven in 3.3.1. Deze stappen kunnen als losse algoritmes worden gezien. Als deze applicatie goed kan worden versneld met de ρ-VEX, dan laat dit zien dat deze processor voor meerdere algoritmes een winst kan opleveren. Daarom geeft het versnellen ervan een redelijke indicatie dat ook veel andere algoritmes met vergelijkbaar resultaat kunnen worden versneld. Het decoderen van JPEG is dus ook een representatief voorbeeld van het gebruik van de ρ-VEX.

Hoofdstuk 3

Achtergrond

Dit hoofdstuk behandelt achtergrondinformatie voor dit project en gaat in op al uitgevoerde, gerelateerde onderzoeken. Eerst zal in paragraaf 3.1 de context worden gegeven waarin dit onderzoek plaatsvindt. In 3.2 wordt een methodiek beschreven om applicaties te versnellen, die ook in dit onderzoek zal worden gebruikt. Het hoofdstuk zal afsluiten met een paragraaf over JPEG (3.3). Hierin wordt het principe van de codering behandeld, en worden enkele vergelijkbare onderzoeken met JPEG-applicaties besproken.

3.1 Parallelle processorarchitecturen

De laatste decennia hebben de prestaties van microprocessoren een grote vlucht gemaakt. De snelheidswinst werd voor een groot gedeelte bereikt door de ontwikkeling van steeds kleinere, snellere transistoren. Deze winst kan vanwege fysieke grenzen echter niet ongelimiteerd gecontinueerd worden. Dit creëert de vraag naar nieuwe oplossingen om de prestatieverhogingen van de laatste decennia voort te kunnen zetten. Om deze prestatieverhogingen te bereiken is de huidige trend om de snelheidswinst te halen uit het parallel uitvoeren van berekeningen[3].

Het parallelliseren van berekeningen kan op meerdere niveaus worden uitgevoerd, bijvoorbeeld door meerdere taken parallel te doen, of door de operaties binnen een programma parallel uit te voeren. In paragraaf 3.1.1 zal hier meer over worden gezegd.

Een parallellisatie op hoog niveau is het uitvoeren van verschillende *threads* op aparte processoren of *cores*. Een populaire, simpele methode hiervoor is om meer van dezelfde cores naast elkaar te zetten, dit heet *symmetric multiprocessing*. Een andere, veelbelovende methode is het gebruik van heterogene multicore platformen en acceleratoren [4]. Dit concept wordt beschreven in 3.1.2.

Ook binnen een processor zijn technieken beschikbaar om parallellisme te verkrijgen bij het uitvoeren van een programma. Twee architecturen die hier gebruik van maken zijn superscalaire en VLIW-processoren, beschreven in paragraaf 3.1.3.

3.1.1 Parallellisme

Parallellisme is het uitvoeren van meerdere berekeningen tegelijkertijd, of de theoretische mogelijkheid hiertoe[5]. De eigenschap die nodig is voor parallellisme is onafhankelijkheid tussen de berekeningen. Twee berekeningen zijn onafhankelijk als voor het uitvoeren van één van beide het resultaat van de ander niet nodig is.

Pipelining is bijvoorbeeld een vorm van parallellisme, waarbij al met een nieuwe instructie wordt begonnen voordat de voorgaande is afgemaakt. Parallellisme op hoog niveau, namelijk meerdere programma's die parallel draaien zoals bij multi-threading, wordt *thread-level* of *job-level* parallellisme genoemd. Omdat slechts één programma met één thread wordt gedraaid, is hier geen sprake van in dit onderzoek. Twee vormen van parallellisme op lager niveau zijn data-level en instructie-level parallellisme, deze worden hieronder besproken.

Data-level parallellisme

Wanneer er een grote hoeveelheid data is die op dezelfde manier is gestructureerd, kan data-level parallellisme toegepast worden. De data die parallel worden verwerkt, dienen onafhankelijk te zijn van elkaar. Mogelijkheden liggen hier bij digitale signaalbewerking, waarbij er vaak sprake is van grote hoeveelheden data die verwerkt moeten worden, zonder dat de uitkomst van deze verwerking invloed heeft op de verwerking van de rest van de data. Als bijvoorbeeld van elke pixel in een plaatje de kleur moeten worden aangepast, dan kan dit goed parallel gebeuren omdat de operatie op elke pixel onafhankelijk is van de kleur van de andere pixels.

Instructie-level parallellisme

Veel instructies of operaties die in een processor verwerkt worden zijn niet afhankelijk van elkaar. Dit geeft de mogelijkheid om meerdere operaties op hetzelfde moment uit te voeren. Instruction-level parallelism (ILP) is de mate waarin meerdere operaties tegelijkertijd uitgevoerd worden. Per programma verschilt hoeveel afhankelijkheid er tussen de operaties bestaat. De potentiële ILP van een programma kan op verschillende manieren worden uitgebuit. Pipelining is hier een voorbeeld van, en VLIW-processoren buiten deze vorm van parallellisme ook uit.

Om ILP te gebruiken zijn VLIW- en superscalaire processoren in staat om meerdere operaties parallel uit te voeren. Door de operaties goed te ordenen, kunnen onafhankelijke operaties naast elkaar gezet worden, zodat deze tegelijkertijd verwerkt kunnen worden. Als er te veel afhankelijkheid tussen is, moet de processor de operaties na elkaar, serieel, uitvoeren. Een deel van de processorcapaciteit blijft dan onbenut.

3.1.2 Heterogene multicore platformen

Heterogene multicore platformen bestaan uit meerdere, verschillende processorkernen. Het achterliggende idee is dat verschillende berekeningen van andere eigenschappen van een processor profiteren, en in een heterogeen platform voor elke taak de juiste processor kan worden gebruikt.

Een voorbeeld is een multicore platform met grote cores waarbij één core vervangen wordt door meerdere kleine, simpelere cores. De functies binnen de applicatie die geen voordeel hebben van de extra mogelijkheden van de grotere core, kunnen worden uitgevoerd op de kleinere cores. Wanneer een applicatie optimaal wordt ingeroosterd, kan het heterogene platform qua snelheid de prestaties van een homogeen platform met hetzelfde aantal cores benaderen [6].

Een voorbeeld van een bestaand heterogeen multicore platform is de *Cell* processor, die onder andere in spelcomputers wordt gebruikt. Deze processor heeft naast een groot *Power Processing Element*, meerdere simpelere *Synergistic Processing Elements*. Deze SPE's zijn vectorprocessoren, waarbij dataparallellisme wordt benut door elke instructie op een reeks data te laten werken[7].

Een benadering om meerdere, verschillende processoren te gebruiken is om één processor als CPU te gebruiken, en andere processoren als acceleratoren, die voor een specifieke taak geschikt zijn. Een bekend voorbeeld is het gebruik van een grafische kaart om videotaken op uit te voeren in een pc. Een accelerator

kan gemaakt zijn om altijd dezelfde taak uitvoeren, of kan programmeerbaar zijn. Over acceleratoren volgt meer in subparagraaf 3.2.2.

3.1.3 Very Long Instruction Word (VLIW)

Om meerdere operaties parallel uit te voeren moeten elke cyclus meerdere operaties worden ingelezen. Meerdere *issue slots* zijn nodig om de operaties tegelijk te kunnen uitvoeren. In een VLIW-processor bestaat elke instructie uit meerdere operaties, namelijk evenveel als de *issue width*, dus het aantal issue slots. Elke cyclus wordt een instructie uitgelezen, en de operaties daarin worden parallel uitgevoerd.

Voor een VLIW-processor moet tijdens het compileren al bepaald worden welke operaties parallel worden uitgevoerd. Dit in contrast met een superscalaire processor, waarbij door de hardware wordt uitgezocht welke operaties onafhankelijk zijn [8]. Deze aanpak maakt de VLIW-processor relatief klein en daarmee snel en energiezuinig[9]. Er is immers extra hardware nodig wanneer dit inroosteren tijdens het uitvoeren van het programma gedaan moet worden.

3.2 Versnelling van software

Versnelling of acceleratie van software houdt in dat de hardware wordt aangepast om een applicatie te versnellen. Hiervoor wordt de hardware dus samen met de software ontwikkeld, ofwel *co-designed*. De hier gebruikte methode van co-design is het toevoegen van een *hardware accelerator* die een deel van het werk van de processor overneemt. Bij veel applicaties neemt een klein stuk van de code een groot deel van de tijd in beslag. Door de applicatie te profileren wordt uitgezocht welk deel van de software het best geaccelereerd kan worden.

Deze paragraaf gaat in op de verschillende aspecten van versnelling van software. Eerst zal hardware/-software co-design behandeld worden (3.2.1). Hierna komen acceleratoren aan bod (3.2.2), en daarna volgt een uitleg over het profileren van een applicatie (3.2.3).

3.2.1 Hardware/software co-design

Embedded systems worden samengesteld uit subsystemen, namelijk hardware zoals ASICs, single- en multicore-processoren, en software. Omdat een systeem met de gewenste functionaliteit op verschillende manieren kan worden opgebouwd, moet worden gekozen welke configuratie van subsystemen gebruikt wordt en welke functies deze subsystemen precies krijgen. Het opdelen van het te implementeren algoritme in hardware en software is een ontwerpprobleem waarbij afwegingen moeten worden gemaakt tussen verschillende factoren zoals snelheid, ruimtegebruik, energieverbruik en ontwikkelkosten. Normaliter worden de hardware en software apart ontwikkeld. De hardware, bijvoorbeeld een bepaalde processor, wordt gekozen of ontwikkeld en de software wordt hier dan voor geschreven.

Bij co-design worden de hardware en software samen ontwikkeld, in plaats van het probleem al in een vroeg stadium te splitsen[10]. De ontwerpers kunnen dan verschillende opties voor de hardware/software-structuur evalueren alvorens een indeling te kiezen. Deze evaluatie kan gedaan worden door simulaties van zowel hardware als software, liefst door een co-simulatie waarbij de combinatie als geheel wordt gesimuleerd.

De meest geschikte hardware/software-verdeling hangt af van de specificaties. Hardware-implementaties zijn bijvoorbeeld sneller dan software-implementaties, maar nemen meer ruimte in, waardoor complexe functies meestal beter in software kunnen worden gemaakt. Ook andere randvoorwaarden zoals energiegebruik, herprogrammeerbaarheid en ontwikkelingskosten kunnen een rol spelen[11].

Een kwestie bij co-design is wat voor ontwerp gemaakt wordt zolang nog niet tot een indeling is besloten. Een optie is om met een software-algoritme te beginnen en delen hiervan over te zetten naar hardware. Het omgekeerde is ook een optie, namelijk om delen van een hardware-ontwerp naar software te vertalen. Dit project heeft onder andere als doel de mogelijke versnelling op een ρ-VEX te onderzoeken. Omdat hier de nadruk op zal liggen wordt er gebruik gemaakt van een bestaande software-implementatie van het algoritme[10].

3.2.2 Acceleratoren

De kern van een embedded system bestaat meestal uit een processor (*Central Processing Unit, CPU*) die het programma uitvoert, en via een communicatiebus verbonden is met een geheugen en *peripherals*. Een accelerator is een peripheral die een bepaalde berekening efficiënt kan uitvoeren zodat deze niet in software hoeft te worden gedaan. Deze accelerator is gespecialiseerd in een specifieke taak waarvoor veel rekenwerk nodig is, bijvoorbeeld data-encryptie of Fouriertransformaties.

De accelerator kan een stuk specifieke hardware zijn, eventueel zelfs een puur combinatorische schakeling. Bij complexere acceleratoren wordt in de accelerator een processor met eigen programma gebruikt. Hiervoor wordt dan een processor gebruikt die beter geschikt is voor de benodigde dataverwerking dan de CPU, bijvoorbeeld een vector- of VLIW-processor. In dit project wordt de ρ-VEX gebruikt als accelerator.

De accelerator is via de communicatiebus verbonden met de processor en het geheugen. Vanuit de processor gezien is de accelerator dus gelijk aan een I/O peripheral. Om de accelerator zijn invoerdata te geven wordt meestal shared memory gebruikt. Zo hoeft de processor de data niet uit te lezen en door te geven maar kan de accelerator zelf de data uit het geheugen lezen. De processor hoeft dan alleen een commando te geven om de accelerator aan het werk te zetten.

In de tijd dat de accelerator bezig is met rekenen zijn er twee opties voor de processor: hij kan in een lus wachten tot de accelerator klaar is, of hij kan alvast doorgaan met andere taken. De eerste optie is simpeler te programmeren, maar de tweede is uiteraard efficiënter, in het geval dat er andere taken te doen zijn.

De snelheidswinst die een accelerator oplevert, hangt af van van twee factoren. De eerste is de tijdswinst die geboekt wordt doordat de accelerator de data sneller verwerkt. De tweede, beperkende, factor is de tijd die nodig is voor communicatie tussen de processor en accelerator. Als de accelerator tijd t_x nodig heeft voor de dataverwerking, en t_{in} en t_{out} nodig zijn voor de invoer en uitvoer van de data, dan is de zogenaamde absolute *speedup*

$$S_a = n\,(t_{CPU} - (t_{in} + t_x + t_{out})) \tag{3.1}$$

waarbij t_{CPU} de tijd is die de hoofdprocessor zonder de accelerator nodig zou hebben gehad voor de geaccelereerde functie, en n het aantal keer dat deze functie wordt aangeroepen.[12]

Deze speedup is dus de tijd die gewonnen wordt bij gebruik van de accelerator. Voor een zo groot mogelijke speedup moet gezocht worden naar een deel van het algoritme dat veel tijd kost op de processor (hoge $n \cdot t_{CPU}$), veel minder tijd kost in de accelerator ($t_x \ll t_{CPU}$), en weinig tijd zal verspillen aan de communicatie (lage t_{in} en t_{out}, en $t_{in} + t_{out} \ll t_{CPU} - t_x$). De duur van de communicatie is sterk afhankelijk de hoeveelheid data die uitgewisseld moet worden. Om de communicatietijd zo laag mogelijk te houden is het dus van belang om een deel van het algoritme te accelereren dat niet veel invoer- en uitvoerdata heeft.

In veel gevallen wordt niet bovenstaande absolute speedup gebruikt maar de algemenere *relatieve speedup*.

Deze is gedefinieerd als

$$S_r = \frac{t_{oud}}{t_{nieuw}} \tag{3.2}$$

waarbij t_{oud} de tijd is die eerst was voor het uitvoeren van de functie of applicatie, en t_{nieuw} de tijd die het systeem na de betreffende aanpassing nodig heeft[13].

3.2.3 Profileren

Om uit te vinden welk deel van het algoritme het best geaccelereerd kan worden, wordt een analyse gemaakt van de tijdsverdeling in het programma. Dit wordt gedaan door een profiler, een gereedschap dat bij het uitvoeren van het programma bijhoudt hoe vaak elke functie wordt aangeroepen, en hoe veel tijd er in elke functie wordt besteed.

Om te meten hoe vaak elke functie wordt uitgevoerd, moeten tijdens het compileren stukjes code worden toegevoegd die elke aanroep registreren. Met deze informatie is een diagram of tabel te maken die weergeeft hoe vaak functies elkaar aanroepen.

Het is lastiger te meten hoeveel tijd in elke functie wordt besteed. Een deterministische aanpak is mogelijk, waarbij toegevoegde stukjes code timers starten die in hardware gerealiseerd zijn. Na het uitvoeren van een functie worden deze timers weer uitgelezen. Deze methode wordt echter gehinderd doordat hardware timers niet pauzeren als er van thread wordt gewisseld, waardoor de methode lastig uit te voeren is op de meeste systemen. Daarom wordt vaak een statistische methode gebruikt, waarbij met regelmaat een timer interrupt wordt geactiveerd. In de interrupt-routine wordt gekeken wat de huidige waarde van de program counter van het programma is, waaruit volgt welke functie op dat moment bezig is. Door vaak genoeg deze steekproef te doen, valt een redelijk beeld te krijgen van de tijdsverdeling binnen het programma. Functies die slechts sporadisch actief zijn zullen een relatief grotere meetfout vertonen, maar dit zijn natuurlijk ook de functies die minder interessant zijn wanneer gezocht wordt naar de bottleneck in het programma[14].

Om te beslissen welk deel van het algoritme zal worden uitbesteed aan de accelerator, moet rekening worden gehouden met hoeveel versnelling en hoeveel communicatie-overhead elke optie oplevert. Met het profiel kan van elke functie worden beschouwd hoeveel data er in en uit gaat en hoeveel tijd er in wordt besteed. Het doel is dan om te zoeken naar een functie waarin veel tijd wordt besteed, maar waar niet veel dataverkeer voor nodig is. Deze functie is dan een geschikte keuze om in de accelerator uit te voeren.

Het kan zijn dat een programma grote functies bevat, waarbij maar een stukje van een bepaalde functie veel tijd vergt. Door het programma in meer, kleinere functies te verdelen kan een preciezer profiel gemaakt worden. Een voorbeeld is om elke lus in een aparte functie te zetten.

3.3 JPEG

Joint Photographic Expert Group (JPEG) is een veelgebruikte standaard om afbeeldingen gecomprimeerd op te slaan. Het coderen naar en decoderen van JPEG is erg rekenintensief. Er zijn al meerdere studies gedaan naar het versnellen van deze operaties. Twee daarvan worden hier besproken in de tweede subparagraaf. In de eerste subparagraaf wordt het bestandsformaat JPEG beschreven.

3.3.1 JPEG bestandsformaat

De compressie van JPEG is 'lossy', er gaat dus informatie verloren bij het coderen. De codering is gebaseerd op de eigenschap van het menselijk oog dat het gevoeliger is voor helderheid dan voor kleur, en dat hoge

spatiële frequenties niet zo opvallen. Vooral voor foto's is JPEG zeer geschikt. Tekst, tekeningen en grafieken hebben in JPEG meer last van een zichtbaar kwaliteitsverschil[15].

JPEG-codering bestaat uit drie stappen, die hier elk zullen worden uitgelegd. Bij het decoderen worden deze stappen, uiteraard in omgekeerde volgorde, stap voor stap teruggedraaid.

Stap 1: YUV kleurcodering

De eerste stap is een conversie in kleurcodering. Afbeeldingen zijn meestal *RGB*-gecodeerd, waarbij elke pixel een waarde heeft voor de rode, groene en blauwe componenten. Hierin is de helderheids- en de kleurinformatie dus niet gescheiden. Door de afbeelding om te zetten in YUV wordt deze informatie wel gescheiden. In de Y-component staat de helderheid van de pixel. Vervolgens vormen de U- en V-componenten samen de informatie over de kleur. Omrekenen van RGB naar YUV gaat als volgt:

$$Y = 0,299R + 0,587G + 0,114B \tag{3.3}$$

$$U = 0,492(B - Y) = -0,147R - 0,289G + 0,436B \tag{3.4}$$

$$V = 0,877(R - Y) = 0,615R - 0,515G - 0,100B \tag{3.5}$$

Normaliter worden de RGB-componenten elk opgeslagen als een getal tussen 0 en 255. Door de U en V grover te kwantificeren dan de Y kan de kleurinformatie verder gecomprimeerd worden dan de helderheid. Zo kan meer informatie weg worden gegooid terwijl de zichtbare kwaliteit van het plaatje nauwelijks achteruit gaat.

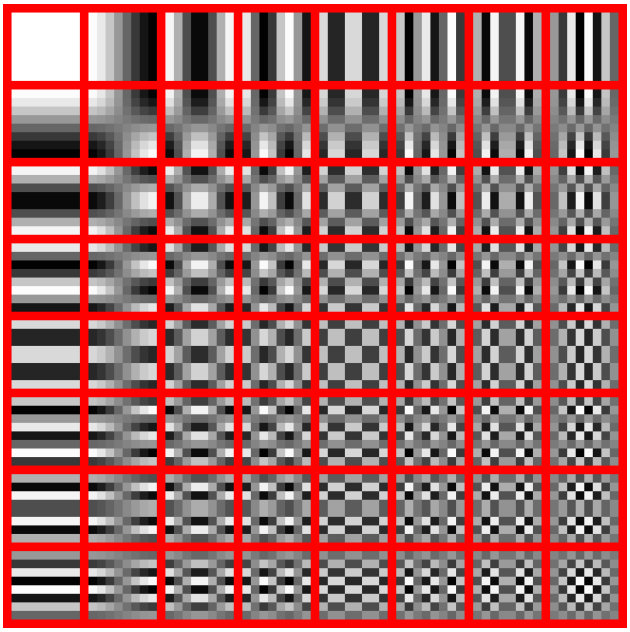
Stap 2: Discrete cosinustransformatie

Op dit moment zijn er nog steeds drie matrices, één voor elk component (Y, U en V). Van elk van deze matrices wordt de discrete cosinus-transformatie (DCT) genomen. Dit gebeurt per blokje van 8×8 pixels, een zogenaamd *Minimal Coding Unit (MCU)*. Bij de DCT wordt de informatie, net als bij de discrete Fouriertransformatie (DFT) weergegeven in het frequentiedomein. Waar DFT het signaal weergeeft in sinussen en cosinussen, wordt er bij DCT alleen gebruik gemaakt van cosinussen. De 1-dimensionale discrete cosinustransformatie van een functie f_n wordt als volgt gedefinieerd:

$$F_u = \sum_{n=0}^{N-1} f_n \cdot cos\left(\frac{\pi(2n+1)u}{2N}\right) \tag{3.6}$$

Door deze 1-dimensionale transformatie horizontaal en verticaal uit te voeren kan een tweedimensionale afbeelding worden getransformeerd. De blokjes van 8×8 pixels leveren na DCT nieuwe matrices op van weer 8×8 pixels. Hierbij staat per matrix de informatie van lage spatiële frequenties linksboven. De hoge frequenties zijn rechtsonder te vinden, zoals gevisualiseerd in figuur 3.1. In deze figuur is te zien hoe elk element in de matrix een frequentiecomponent van de afbeelding representeert.

Door elk element in elke matrix te delen door een bepaald getal en vervolgens af te ronden wordt de afbeelding gecomprimeerd. Hier kan per element een ander getal gekozen worden. Deze getallen staan in een zogenaamde kwantisatiematrix. De hogere frequenties worden door grotere getallen gedeeld, daardoor worden veel van deze elementen nul.



Figuur 3.1: Frequentieverdeling na DCT (bron: en.wikipedia.org/wiki/JPEG)

Stap 3: Entropy encoding

Het uitvoeren van de vorige stap resulteert in drie matrices die elk de frequentiespectra van één component bevatten. Deze worden nu met exact omkeerbare (*lossless*) compressie nog verder gecomprimeerd.

De matrices worden elk omgezet naar een string. Hierbij worden de elementen schuin, zigzaggend uitgelezen. Hierdoor worden de elementen met ongeveer dezelfde frequenties achter elkaar gezet in de string, zoals getekend in figuur 3.2.

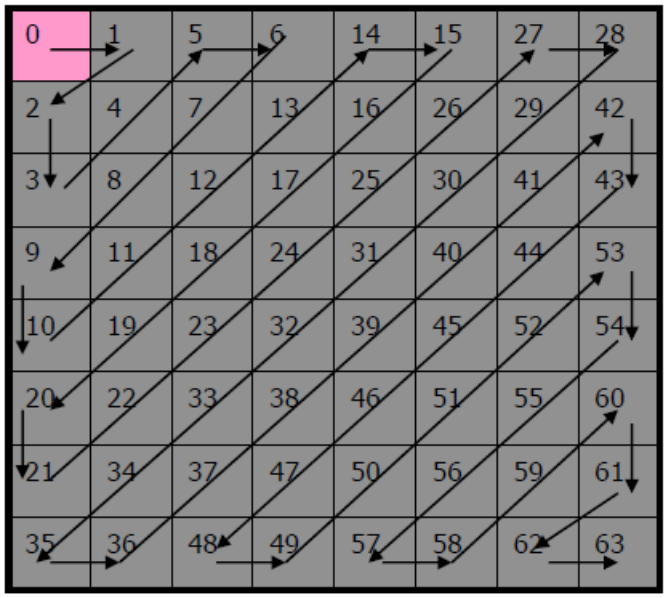
Omdat de gekwantiseerde waardes van de hogere frequenties vaak nul zijn, zal de string veel nullen achter elkaar bevatten. Daarom wordt *run-length encoding* toegepast waarmee het bestand verder verkleind wordt. Tot slot wordt *Huffman encoding* toegepast. Hierbij krijgen de meest voorkomende symbolen in de string kortere representaties dan de minder vaak voorkomende symbolen.

3.3.2 Onderzoeken naar JPEG-acceleratie

Er zijn meerdere studies gedaan naar het versnellen van de verwerking van JPEG met behulp van een accelerator. Het bedrijf Critical Blue heeft dit gedaan met een ARM7 als host[16]. Hier hebben ze een enkele processor als accelerator gebruikt. Verder is aan de Linköpings Universiteit in Zweden onderzoek gedaan naar een JPEG-encoder en -decoder, die gebruik maakte van een hardware-accelerator[17].

Critical Blue

Critical Blue heeft van elke stap in het decoderen van JPEG onderzocht hoeveel snelheidswinst er in te behalen viel. Er is uitgegaan van de open-source JPEG-software gemaakt door de Independent JPEG Group [18]. Een belangrijke eis binnen het onderzoek was dat er geen grote aanpassingen gedaan zouden worden aan de



Figuur 3.2: Zigzaggend uitlezen (bron: projectrhea.org)

code zelf. Een totale versnelling van een factor 2,4 is behaald. De limiterende factor was de Huffman decoder. De seriële aard van deze compressie beperkt de mogelijkheid tot meer parellellisme. De implementaties van de RGB-decoder en de inverse DCT leverde een versnelling op van 7 tot 8 keer. De code was nog niet optimaal. Wanneer de code werd geoptimaliseerd om ideaal gebruik te kunnen maken van pipelining kon de totale versnelling oplopen tot 4 keer.

Linköping

De studie van Linköpings Universiteit had als doel om een processor, die al geoptimaliseerd was als Digital Signal Processor, verder te optimaliseren door extra hardware toe te voegen aan het ontwerp. Om deze extra hardware aan te sturen is er gebruik gemaakt van een extra set instructies. Behalve snelheidswinst is er ook winst gemaakt op het gebied van energieconsumptie. Er werd niet alleen winst geboekt doordat de hardware de data sneller kan verwerken, maar ook doordat de CPU parallel met de hardware wordt gebruikt. De gebruikte code is specifiek voor dit onderzoek geschreven en betreft zowel een JPEG encoder als decoder. Bij het decoderen zorgde het gebruik van de accelerator voor een gemiddelde versnelling van een factor 5,6.

Hoofdstuk 4

Applicatie

De applicatie die in dit project wordt gebruikt implementeert het decoderen van een afbeelding die in JPEG-formaat is opgeslagen. De applicatie leest het JPEG-bestand in en decodeert het om er een bitmap-plaatje van te maken. Deze wordt opgeslagen als BMP-bestand.

Na de keuze voor het decoderen van JPEG is bestaande applicatiecode gezocht. De structuur van deze code is bestudeerd, en vervolgens is een profiel van de applicatie gemaakt, om inzicht te krijgen in de tijdsverdeling van de functies. Dit profiel is later, zoals beschreven in hoofdstuk 6, gebruikt om de bottle-necks van het programma te analyseren. De code van de applicatie diende ook aangepast te worden, om de applicatie op de MicroBlaze en de ρ-VEX te laten werken.

Dit hoofdstuk gaat achtereenvolgens in op de keuze voor de applicatie (4.1), beschrijft de interne structuur ervan (4.2) en toont haar profiel (4.3). Vervolgens wordt uitgelegd welke aanpassingen aan de code nodig zijn geweest (4.4).

4.1 Keuze van applicatie

Er zijn meerdere implementaties te vinden voor het decoderen van JPEG-bestanden. Gezocht is naar een implementatie in C, omdat de VEX en MicroBlaze toolchain deze taal beide ondersteunen. Bij de VEX toolchain van HP is standaard een testapplicatie meegeleverd die JPEG-bestanden decodeert en omzet naar bitmaps, gebaseerd op oude code van de Indepedent JPEG Group. Bij het testen hiervan bleek deze applicatie niet de correcte bitmaps te genereren, maar gestippelde plaatjes. Daarom is besloten om een implementatie te gebruiken geleverd door Roël Seedorf, gebaseerd op code geschreven door Pierre Guerrier. Een andere keuze had een recentere implementatie van Independent JPEG Group kunnen zijn. Hoewel deze laatste implementatie wel werkt is het een erg uitgebreid programma, met veel mogelijkheden die voor de demonstrator niet gebruikt worden. Dit maakt grote delen van de code overbodig, en het maakt de code onoverzichtelijk en daardoor lastiger om mee te werken.

Het gebruikte programma heeft weinig opties en is slechts geschikt om JPEG-afbeeldingen te decoderen. Omdat dit de enige functionaliteit is die in dit onderzoek gebruikt zal worden, is deze beperking geen probleem maar juist handig. Deze beperking komt de overzichtelijkheid van het programma ten goede, wat het gebruiken en aanpassen van de code een stuk gemakkelijker maakt.

4.2 Structuur van de code

De kern van de gebruikte applicatie bestaat uit zeven C-bronbestanden. De verschillende stappen van het decoderen van JPEG zoals beschreven in paragraaf 3.3.1 worden elk in een bestand uitgevoerd. De rest van de bestanden regelt het uitlezen van het JPEG-bestand en het correct aanroepen van de decoderingsfuncties. Hier volgt een kort overzicht van de functies van de bestanden.

jpegtobmp.c Hierin is *JpegToBmp* gedefinieerd, de hoofdfunctie van de applicatie.

parse.c Hierin bevinden zich allerlei functies om het JPEG-bestand uit te lezen. Deze functies dienen onder andere om de volgende marker te zoeken in het JPEG-bestand (*get_next_MK*), segmenten over te slaan (*skip_segment*), en MCU's te verwerken (*process_MCU*).

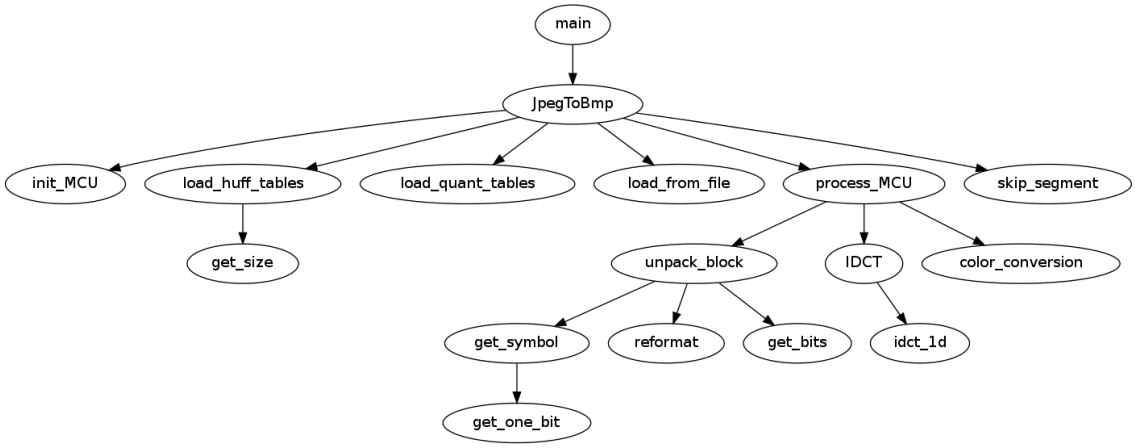
table_vld.c Bevat de functie *load_huff_tables* om de Huffman-tabel te creëren die gebruikt wordt voor de eerste stap van de decompressie.

huffman.c Bevat de functie *unpack_block* waarmee de eerste decompressiestap wordt uitgevoerd.

fast_int_idct.c Hierin staat de functie *IDCT* om de inverse discrete cosinustransformatie uit te voeren op een MCU-blok.

color.c Hierin staat de code om de kleurconversie te doen, *color_conversion.*

utils.c Dit bestand bevat enkele kleine hulpfuncties.



Figuur 4.1: Callgraph JPEG-decoderapplicatie

Hieronder zal stap voor stap uitgelegd worden hoe en waarvandaan de belangrijkste functies worden aangeroepen. Ook zal ingegaan worden op de taak van deze functies. Ter ondersteuning is in figuur 4.1 een *callgraph* bijgevoegd.

Wanneer de applicatie gestart wordt, wordt de functie *JpegToBmp* direct aangeroepen door *main.* Deze functie opent het JPEG-bestand en kijkt steeds wat de volgende *marker* in het bestand is die aangeeft wat voor segment er volgt. Het bijbehorende segment wordt dan verwerkt. De eerste marker is altijd 'Start Of

File', waarbij informatie, zoals de afmetingen van het plaatje staan. Uit de volgende segmenten worden de kwantisatietabel en de Huffman-tabel gelezen door load_huff_tables en load_quant_tables. Na deze initialisatiestappen begint, bij het vinden van de 'Start Of Scan' marker, het decoderen van de MCU's.

Voor het decoderen wordt per MCU *process_MCU* aangeroepen. Hierbinnen wordt het MCU-blok uit het bestand gelezen, en worden alle verwerkingsstappen hierop uitgevoerd. De eerste stap wordt gedaan in de functie *unpack_block*. Hierin wordt de Huffman-compressie gedecodeerd, wordt de entropy-encoding uit het MCU-blok gehaald en wordt de kwantisatie gecorrigeerd. Wanneer dit gebeurd is, wordt de inverse genomen van de discrete cosinus-transformatie door de functie *IDCT*. Deze functie voert de transformatie in twee dimensies uit, door eerst voor elke rij en dan voor elke kolom de functie *idct_1d* aan te roepen, die een ééndimensionale transformatie uitvoert. Op dit moment is de frequentie-codering uit de afbeelding gehaald. Wat rest is de conversie van YUV-kleurcodering naar RGB. Hiervoor dient de functie *color_conversion*. Na deze conversie is de JPEG-decodering voltooid. Het MCU-blok bevat nu de RGB-waarden van de 64 pixels.

Door deze stappen voor elk MCU-blok uit te voeren is de complete afbeelding in RGB-waarden omgezet, resulterend in drie matrices voor de drie kleurcomponenten. Ten slotte worden de matrices per pixel samengevoegd en opgeslagen als bitmap-bestand *(.bmp)*.

4.3 Profileren van de applicatie

Voor het accelereren van de applicatie is het belangrijk een goed beeld te hebben van de tijdsverdeling tussen verschillende functies. Hiervoor is de applicatie geprofileerd met behulp van het programma *gprof*. Het profileren is gedaan op een simulatie van de VEX-processor. Hiermee kan op de pc worden geanalyseerd hoe de applicatie op een VEX presteert. De werking van deze simulator wordt beschreven in paragraaf 5.2.3.

Voor het maken van het profiel is de aangepaste versie van de applicatie gebruikt, aangezien deze ook op de MicroBlaze en ρ-VEX worden gebruikt. Het profileren van de applicatie dient grotendeels om uit te vinden hoe de functies het best verdeeld kunnen worden over de twee processoren. Hiervoor is het nodig een beeld te krijgen van waar in het programma de tijdverslindende *kernels* zitten bij het uitvoeren op een Microblaze, opdat het executeren van deze kernels kan worden uitbesteed aan de ρ-VEX. Omdat het profileren op de MicroBlaze lastiger bleek dan op de VEX-simulator, is gekozen om de VEX-simulator te gebruiken. Hiervoor is een VEX-configuratie gekozen met één *issue slot*, waardoor het dus eigenlijk een gewone RISC-processor wordt in plaats van een VLIW-processor. De MicroBlaze heeft 32 *general purpose registers*. De VEX-simulator is, om een zo goed mogelijk model van de Microblaze te krijgen, dus ook hierop ingesteld. Aangenomen wordt dat deze VEX-configuratie genoeg lijkt op een MicroBlaze om de profiel-resultaten te vergelijken. Voor het profiel is het niet belangrijk dat de cijfers precies kloppen. Het profiel is slechts een benadering, en per ingevoerd plaatje zal dit ook iets verschillen.

De simulator creëert zelf een paar extra functies, deze zijn herkenbaar aan de naam die steeds begint met een laag streepje. Deze functies zullen niet uitgevoerd worden op de ρ-VEX of de MicroBlaze maar komen soms wel terug in het profiel, waar ze dus genegeerd kunnen worden.

Figuur 4.2: *surfer.jpg*, 32×24 pixels testplaatje

In figuur 4.3a (op pagina 23) is het gemeten profiel van de applicatie weergegeven, uitgevoerd op een VEX simulator met één issue slot. De totale executietijd was $5,91 \cdot 10^5$ klokcycli. Als invoer is een klein plaatje van 32 bij 24 pixels gebruikt (getoond in figuur 4.2), deze wordt de rest van het onderzoek ook gebruikt.

In de profielen zijn alleen de functies die meer dan 1% van de tijd innemen weergegeven. Bij elke functie staat hoeveel procent van de tijd in deze functie wordt besteed. Daaronder staat tussen haakjes hoeveel tijd in de functie zelf, dus niet in haar subfuncties, wordt besteed. De tijd die wel in de subfuncties wordt besteed staat langs de pijlen naar deze functies, samen met hoe vaak deze subfunctie aan wordt geroepen.

Wanneer ook een profiel gemaakt wordt op een simulatie van een VEX met vier issue slots, dan kan deze vergeleken worden met het profiel op de één-issue slot-configuratie. Dit profiel is weergegeven in figuur 4.3b. Behalve de issue width zijn alle instellingen gelijk gehouden. De executietijd was $3,36 \cdot 10^5$ klokcycli, zoals verwacht een stuk sneller dan met één issue slot, maar ook nog lang niet vier keer zo snel omdat de ILP van de applicatie daarvoor niet hoog genoeg is.

Door de twee profielen in figuur 4.3 te vergelijken kan geconcludeerd worden welke functies beter en welke minder goed parallelliseerbaar zijn. De functies die op vier issue slots relatief aan de andere functies minder tijd gaan gebruiken, zijn beter parallelliseerbaar dan de functies die relatief meer tijd kosten. Deze informatie kan invloed hebben op hoe de partitionering van de software tussen de MicroBlaze en de ρ-VEX wordt gekozen, zoals in paragraaf 6.1.2 wordt besproken.

4.4 Aanpassingen aan software

Het uitvoeren van een applicatie op een embedded platform gaat net anders dan op een uitgebreider systeem zoals een pc. De gebruikte applicatie is geschreven uitgaande van de aanwezigheid van een besturingssysteem en bijbehorende functies. Deze aannames gaan niet op voor het hier gebruikte processorplatform. Om de applicatie te *porten* naar de MicroBlaze en ρ-VEX zijn aanpassingen aan de applicatie nodig. Op de pc en op de VEX-simulator werkte de applicatie wel vrijwel zonder aanpassingen.

In deze paragraaf zal eerst in worden gegaan op de geldende restricties. Vervolgens wordt aangegeven welke functies binnen het programma problemen opleveren. Tot slot zal de gekozen oplossing worden behandeld.

4.4.1 Restricties

De MicroBlaze en ρ-VEX worden zonder besturingssysteem gebruikt. Hierdoor is het niet mogelijk om gebruik te maken van *system calls*. System calls bieden normaal gesproken de interface tussen het programma en het besturingssysteem. Ze worden gebruikt om het programma te laten communiceren met andere onderdelen van het systeem, zoals met het bestandsyssteem en eventueel met andere programma's.

Het programma rekent op de aanwezigheid van enkele functies uit de standaard C-bibliotheek (*libc*), bijvoorbeeld *fopen* en *fgetc* voor het openen en uitlezen van het JPEG-bestand en *malloc* voor het reserveren van geheugenruimte. In de meeste systemen is de compiler bekend met deze bibliotheek en kunnen deze functies zonder probleem gebruikt worden.

Voor de ρ-VEX is libc niet beschikbaar, dus deze functies kunnen niet worden aangeroepen in het programma. Voor de MicroBlaze is libc wel beschikbaar, maar werken niet alle functies daaruit. Sommige functies zijn namelijk *wrappers* die system calls aanroepen, bijvoorbeeld de functie *fopen* die de system call *open* aanroept. Omdat er geen besturingssysteem is werken deze functies dus niet. Voor beide processoren moet de software dus aangepast worden om er op te kunnen draaien.

4.4.2 Problematische functies

In de applicatie wordt een aantal functies uit libc aangeroepen. Deze geven problemen bij het draaien op de ρ-VEX en MicroBlaze. De betreffende libc-functies zijn hieronder aangegeven en omschreven.

fprintf Dit is een functie om een string tekst of tekens te printen. Behalve naar het scherm wordt deze functie ook gebruikt om naar het BMP-bestand te schrijven.

fopen Deze functie wordt in het begin aangeroepen om het JPEG-bestand te openen, en aan het eind nog eens, om het BMP-bestand te openen. Bij het aanroepen wordt de bestandsnaam meegegeven, en wordt aangegeven of het in lees- of schrijfmodus moet worden geopend. De functie retourneert een *file pointer* die door andere functies kan worden gebruikt om operaties op het bestand te doen.

fgetc Aan deze functie wordt een *file pointer* meegegeven van een bestand dat geopend is om te lezen. *fgetc* leest één byte uit het aangewezen bestand. Omdat bestanden niet als zodanig geopend kunnen worden is het niet mogelijk deze functie uit te voeren.

putc Aan deze functie wordt een *char* (één byte) en een file pointer meegegeven. Deze functie schrijft deze byte in het bestand, mits deze in schrijfmodus is geopend.

fseek Hiermee wordt de positie-indicator van het aangewezen bestand naar een aangegeven plek verplaatst. Het maakt het mogelijk om naar een andere plek binnen een bestand te gaan om daar iets neer te schrijven of uit te lezen. Bij het aanroepen van *fgetc* en *putc* wordt de positie-indicator altijd een plek opgeschoven.

ftell *ftell* geeft de huidige plek van de positie-indicator van een bestand terug.

fclose Sluit het aangegeven bestand.

malloc *malloc* zoekt en reserveert een vrij stuk geheugen voor het programma. De grootte van dit stuk wordt bij het aanroepen meegegeven. De compiler voor de MicroBlaze biedt deze functie wel, hoewel deze hierop niet lijkt te werken bij grote reserveringen. Voor de ρ-VEX is deze functie niet beschikbaar.

memmove Deze functie verplaatst een aantal bytes in het geheugen, waarbij meegegeven wordt van waar naar waar, en hoeveel bytes verplaatst moeten worden. Overlap wordt mogelijk gemaakt door het verplaatsen via een buffer te doen. Deze functie is er wel op de MicroBlaze maar niet op de ρ-VEX.

Naast deze aanroepen in de applicatie zelf, was er bij het compileren voor de ρ-VEX nog een probleem. De VEX-compiler genereert namelijk aanroepen naar enkele interne functies, merkwaardig genoeg zonder ook de implementatie van deze functies te leveren. Bij deze applicatie werden aanroepen naar de functies *_bcopy, i_div* en *i_udiv* gegenereerd.

4.4.3 Gekozen oplossingen

Om de JPEG-code op de MicroBlaze en de ρ-VEX te laten draaien moeten de functies die in de vorige subparagraaf beschreven staan worden vervangen door andere functies. Hier zijn drie oplossingen voor gebruikt. De functie *fprintf* is op een aantal plekken vervangen en op een aantal plekken verwijderd, *malloc* en *memmove* zijn omzeild, en de functies die met bestanden werken zijn geïmplementeerd.

Met deze aanpassingen gebruikt de applicatie geen functies meer die niet in de broncode zijn gedefinieerd. Ook zijn kleine aanpassingen gedaan om te zorgen dat de compiler geen functie-aanroepen genereert. Hieronder volgen de gebruikte oplossingen.

fprintf

Waar de functie *fprintf* wordt aangeroepen om tekst op het beeld te tonen, is deze op de MicroBlaze simpel te vervangen door de beschikbare *xil_printf*. Bovendien is deze functie niet essentieel om het programma laten werken, het levert alleen informatie over het decoderingsproces. Dit neemt niet weg dat het voor de terugkoppeling die het programma kan geven een waardevolle functie is. Op de ϱ-VEX moest *fprintf* wel worden verwijderd, dus daar kan bij het uitvoeren geen terugkoppeling worden gegeven naar de gebruiker.

Op één plek werd *fprintf* gebruikt om naar het BMP-bestand te schrijven. Hier is deze aanroep vervangen door een simpele lus, die voor elk teken in de string *putc* aanroept, wat hier hetzelfde resultaat oplevert.

Bestandsoperaties

Omdat niet mogelijk is om in de applicatie tijdens het uitvoeren bestanden te openen, dient dit te gebeuren vóór het compileren. Dit probleem is opgelost door de inhoud van het bestand in de broncode te zetten. Hiervoor wordt de inhoud van het JPEG-bestand via een script omgezet in een rij hexadecimale getallen. Een C-bronbestand, genaamd *jpegfile*.c, wordt gegenereerd waarin deze rij getallen in een array wordt gezet. Deze C-code wordt meegegeven aan de compiler, die zo dus het hele JPEG-bestand ín het programma zet. Zo hoeft het bestand niet meer te worden ingelezen tijdens het uitvoeren.

Wanneer de inhoud van het bestand in een array te wordt geplaatst, is ook bekend wat de grootte en de afmetingen van de afbeelding zijn. Deze informatie was nodig om één van de *malloc*-aanroepen te vervangen. Hierbij werd ruimte voor een buffer gereserveerd ter grootte van de bitmap-afbeelding. Wanneer de afmetingen van de afbeelding bekend zijn, is ook bekend hoe groot de bitmap-afbeelding zal zijn. Hiervoor wordt in het gegenereerde bronbestand ook een array gedeclareerd van het benodigde formaat. Het eindproduct van de decoder is ook een array, aangezien de bitmap niet in een bestand kan worden opgeslagen. De C-code wordt gegenereerd door een script, dat zelf wordt aangeroepen in de *Makefile*. Hierdoor kan bij het aanroepen van *make* worden aangegeven welk JPEG-bestand gedecodeerd moet worden, waarna dit bestand met de broncodes gecompileerd wordt.

Om de 'bestanden', dus de arrays, uit te lezen en te schrijven is gekozen de functies *putc*, *fgetc*, *fseek*, en *ftell* zelf te implementeren. Hiervoor is een extra C-bestand, *fileio.c* genaamd, geschreven. In appendix A is dit bestand te vinden. In de applicatie hoeven de aanroepen dus niet aangepast te worden. De implementaties zijn erg simpel gemaakt, ze werken elk maar voor één bestand. De meegegeven file pointer wordt dus genegeerd. Dit kan omdat in de applicatie de functies *fgetc*, *fseek*, en *ftell* alleen voor het invoerbestand worden gebruikt, en *putc* alleen voor het uitvoerbestand.

De gemaakte implementaties worden op dezelfde wijze aangeroepen als die van libc. De aanpassingen zijn dus transparant voor de applicatie, er hoeft in de applicatie-code niets te worden aangepast.

Om wel meerdere bestanden te kunnen openen zou er een complexer *file-handling*-systeem gemaakt moeten worden. Dit is echter, door de beperkte tijd van dit project, niet mogelijk geweest te implementeren, en was voor deze demonstrator ook nog niet nodig.

Deze implementatie van de bestanden bleek een probleem te geven bij het compileren met de VEX compiler. Deze compiler is niet capabel om extreem grote expressies te verwerken, wat het definiëren van een array met vele duizenden bytes lastig maakt. Een nettere manier om het JPEG-bestand in het programma te compileren zou zijn om het niet naar C-broncode te converteren en vervolgens te compileren, maar om het

zelf direct in een *object file* te zetten. Nadeel hiervan is dat object files verschillen per systeem, en per platform dus moet worden uitgezocht hoe deze bestanden gegenereerd moet worden. De gegenereerde C-code daarentegen is wel voor elk platform bruikbaar.

Ook bij het genereren van C-code is het probleem te omzeilen, door elk element van de array in een aparte expressie te definiëren. Als oplossing is echter gekozen om voor dit project simpelweg niet al te grote afbeeldingen te gebruiken. Voor het onderzoek maakt het niet uit of de afbeelding twee kilobytes of tweehonderd kilobytes groot is, omdat verder alle methodes schaalbaar zijn.

Malloc en *memmove*

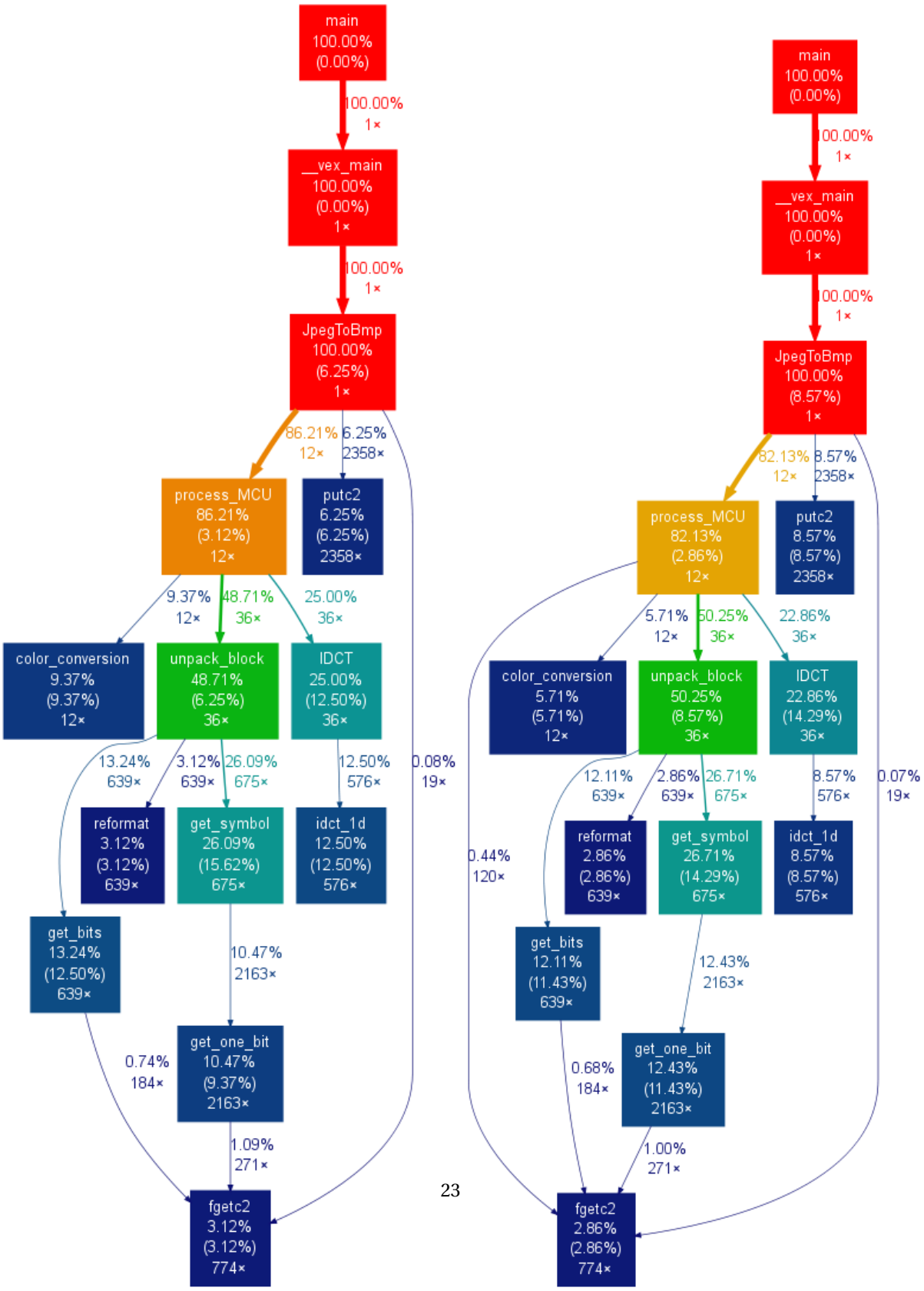
Om *malloc* niet meer te hoeven gebruiken is onderzocht hoe vaak deze wordt aangeroepen en hoeveel geheugen dan steeds gealloceerd werd. Hierbij bleek dat deze slechts werd aangeroepen om ruimte voor de buffers te maken waarin de decodering werd uitgevoerd. De hoeveelheid ruimte die steeds werdt gealloceerd is voorspelbaar als de afmetingen van het plaatje bekend zijn. Daarom kon *malloc* makkelijk worden omzeild door deze ruimte statisch te laten alloceren. Dat wil zeggen dat deze buffers al van tevoren als globale variabelen zijn gedeclareerd, in plaats van tijdens het programma nog geheugenruimte te zoeken.

De functie *memmove* werd op één plek aangeroepen, in *process_MCU*. Deze aanroep is vervangen door een lus die de bytes stuk voor stuk verplaatst.

Door compiler gegenereerde hulpfuncties

De door de compiler gegenereerde aanroepen naar *i_div* en *i_udiv* werden daar gebruikt waar delingen moesten worden uitgevoerd. Ze bleken uiteindelijk simpel op te lossen door de compiler de optie *-fexpand_div* mee te geven, waardoor hij de delingen compileerde zonder functies aan te roepen.

De functie *_bcopy* werd maar eenmaal aangeroepen. Deze aanroep bleek te worden gebruikt om data uit het programma naar de stack te kopiëren, omdat in de functie *jpegtobmp* een array werd gedefinieerd met daarin de header van het BMP-bestand. Om van deze *_bcopy* af te komen is van deze array een globale variabele gemaakt.



(a) Uitgevoerd op één issue slot, executietijd is $5,91 \cdot 10^5$ cycli

(b) Uitgevoerd op vier issue slots, executietijd is $3,36 \cdot 10^5$ cycli

Figuur 4.3: Profielen van de applicatie op twee VEX-simulaties met verschillende issue width

Hoofdstuk 5

Architectuur

Om de demonstrator te maken moest een keuze worden gemaakt voor de processoren en de configuratie daarvan. Dit hoofdstuk gaat in op de gebruikte computerarchitectuur.

Uiteraard stond al vast dat de ρ-VEX zou worden gebruikt. Deze processor is echter niet geschikt om als losstaand systeem te worden gebruikt. Naast de ρ-VEX wordt daarom ook gebruik gemaakt van de MicroBlaze-processor van Xilinx. De MicroBlaze en ρ-VEX worden samen in een FPGA gezet, waarbij de gekozen aanpak is om de MicroBlaze als CPU te gebruiken en de ρ-VEX als accelerator. De decoder-applicatie wordt dus uitgevoerd op de MicroBlaze, die (een deel van) de berekeningen uit kan besteden aan de ρ-VEX. Deze opstelling maakt het makkelijk om de snelheid te vergelijken met de situatie waarin enkel een MicroBlaze wordt gebruikt. Het hele systeem wordt in een FPGA gezet, hiervoor wordt een *ML605 Evaluation Board* van Xilinx gebruikt met daarop een *Virtex 6* FPGA.

De ρ-VEX is een configureerbare processor, gebaseerd op de VEX-architectuur. Welke configuratie voor de ρ-VEX wordt gebruikt wordt bepaald aan de hand van de in hoofdstuk 4 beschreven applicatie. In hoofdstuk 6 zal worden beschreven hoe de applicatie en de architectuur worden gecombineerd.

In paragraaf 5.1 wordt de MicroBlaze beschreven. Vervolgens worden de VEX-architectuur en toolchain beschreven in paragraaf 5.2. Dan wordt beschreven hoe de ρ-VEX werkt en welke configuratie hiervoor gekozen is (5.3). Ten slotte wordt uitgelegd hoe de verbinding tussen de MicroBlaze en ρ-VEX is georganiseerd om het totale systeem te maken (5.4).

5.1 Beschrijving MicroBlaze

De keuze voor welke processor als CPU wordt gebruikt is niet erg kritiek, aangezien het echte rekenwerk door de ρ-VEX zal worden gedaan. De CPU hoeft op zijn minst slechts de applicatie in de ρ-VEX te zetten, de JPEG-afbeelding aan te leveren en de BMP-afbeelding weer uit te lezen. Voor de demonstrator is vooral belangrijk dat deze processor vergelijkbaar is met wat in de meeste camera's en andere embedded systems wordt gebruikt. Een 32-bits RISC-processor leek daarom de beste keus.

De MicroBlaze is een *soft-core* processor van Xilinx en wordt standaard meegeleverd bij de *Embedded Development Kit*. Omdat deze kit op de afdeling Computer Engineering al gebruikt werd om de ρ-VEX te testen, lag het voor de hand om deze processor te gebruiken.

De MicroBlaze is een 32-bits RISC processor gebaseerd op een Harvard-architectuur. De processor heeft 32 *general purpose registers* en is uitgerust met een pipeline. Om een systeem met MicroBlaze te maken dat

in de FPGA kan worden gezet, wordt Xilinx Platform Studio gebruikt. Hierin wordt een platform gegenereerd bestaande uit de MicroBlaze, een klein intern geheugen, groter extern geheugen, interrupt controller en *UART* interface.

Om de MicroBlaze te programmeren wordt de *Xilinx Software Development Kit* (*SDK)* gebruikt. In dit programma kan de code worden geschreven en gecompileerd. Ook kan hiermee het hardware-ontwerp in de FPGA worden geladen, en vervolgens de gecompileerde applicatie in de geheugens van de MicroBlaze worden geprogrammeerd. Tijdens het uitvoeren van de applicatie op de MicroBlaze worden alle *print-statements* die de applicatie uitvoert via een UART verbinding naar de SDK gestuurd, die de tekens op het scherm weergeeft. Zo kan de programmeur zichzelf informatie laten verschaffen over de status van het systeem, wat erg nuttig is voor het testen van de applicatie, en voor het foutzoeken indien de applicatie niet werkt.

5.2 Beschrijving VEX-architectuur

De *VEX* (*VLIW Example*) is ontwikkeld door HP. De VEX omvat de specificatie van een klasse VLIW-processor-architecturen, en een *toolchain* om met deze architecturen te experimenteren[19]. De toolchain is een compilatie- en simulatiesysteem, waarmee programma's kunnen worden gecompileerd, gesimuleerd en geanalyseerd.

De VEX-specificatie definieert slechts een instructieset en kan dus op verschillende wijzen geïmplementeerd worden. De VEX definieert daarbij niet de hele architectuur exact, maar houdt enkele variabelen open, die bij de implementatie mogen worden gekozen. Zo staat bijvoorbeeld de *issue width* niet vast. Bij het ontwerpen van de implementatie mag dus worden gekozen hoeveel operaties parallel kunnen worden uitgevoerd.

Deze paragraaf zal eerst de VEX instructieset beschrijven (5.2.1), dan de compiler (5.2.2) en de simulator (5.2.3).

5.2.1 VEX instructieset

De VEX instructieset of *instruction set architecture* (*ISA*) is gebaseerd op een RISC-architectuur. Omdat de VEX een VLIW-architectuur is, zijn er iets andere termen dan gebruikelijk. Een instructiewoord bevat meerdere *syllables*, die elk een operatie coderen. Een syllable op de VEX komt op een gewone, niet-VLIW RISC-processor overeen met een *operatie* of *instructie*. De lengte van elke syllable is gespecificeerd als 32-bits. De lengte van de instructie hangt af van de gebruikte issue width.

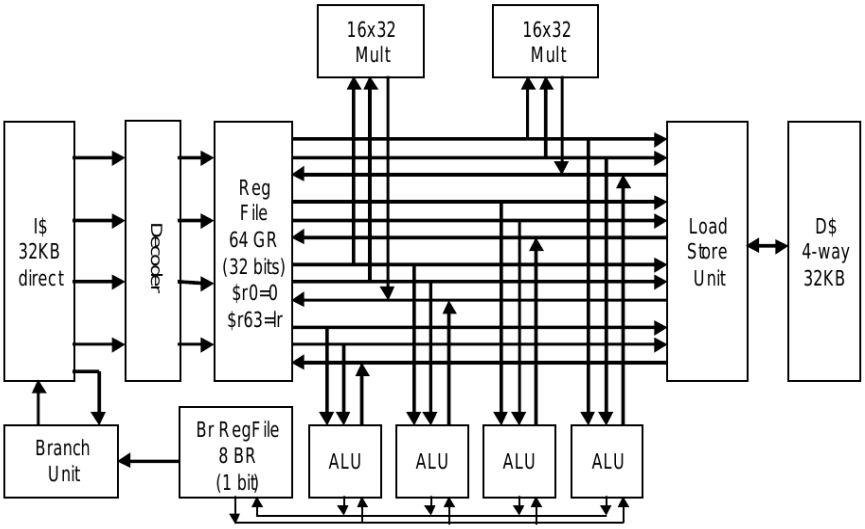
De operaties in een instructie worden parallel uitgevoerd. Elke operatie leest haar operanden uit het registergeheugen voordat er resultaten terug worden geschreven. Elk register bevat 32 bits, en het aantal registers is instelbaar tot maximaal 64 stuks. Om meerdere operaties parallel te verwerken moeten de operands parallel kunnen worden uitgelezen. Voor elk issue slot heeft het registergeheugen twee *read ports* en één *write port*, zodat elke operatie twee operands kan lezen en één resultaat kan schrijven.

De VEX kan meerdere *clusters* bevatten, waarbij elk cluster een eigen registergeheugen heeft. Voor elk cluster kan een issue width worden gekozen. In dit project wordt slechts één cluster gebruikt, meerdere clusters ondersteunt de ρ-VEX niet.

Elk cluster heeft een aantal *functional units* (*FU*). Een functional unit kan een *branch unit* zijn, een *load-store unit*, een *arithmetic logic unit* (*ALU*) of een *multiplier* (*MUL)*. Ook laat de specificatie ruimte over om zelf FU's toe te voegen aan de implementatie, bijvoorbeeld een *floating-point unit* of een eigen ontwerp. Elke VEX heeft één branch unit, aangezien het programma nooit naar twee verschillende plekken tegelijk kan springen. Verder is vrij te kiezen welke functional units worden geplaatst.

De load-store ofwel memory unit wordt gebruikt om data uit het geheugen in een register te plaatsen, en om uit een register naar het geheugen te schrijven. Omdat de VEX een RISC-architectuur is, zijn de load- en store-operaties de enige manier om het geheugen te bereiken.

In figuur 5.1 is de standaardconfiguratie van de VEX weergegeven. Deze heeft vier issue slots, vier ALU's, twee multipliers, en één load-store unit.



Figuur 5.1: De standaardconfiguratie van de VEX (bron: [19])

5.2.2 VEX compiler

In de toolchain van de VEX zit een C-compiler die assembly-code voor de VEX genereert. Aan deze compiler wordt een *machine model file* meegegeven waarin de gekozen configuratie van de VEX staat. In dit configuratiebestand kunnen de volgende variabelen worden gedefinieerd:

- Aantal clusters
- Aantal issue-slots (issue width)
- Aantal ALU's
- Aantal MUL's
- Aantal registers
- De delay van elke functional unit

De gegeven configuratie bepaalt hoe de compiler de operaties kan inroosteren. Het doel van de compiler is om het programma zo te compileren dat er zo min mogelijk klokcycli nodig zijn om het uit te voeren, en dat het programma niet overbodig veel geheugenruimte inneemt. De compiler probeert de code hiervoor te optimaliseren, om de gegeven configuratie zo goed mogelijk te gebruiken.

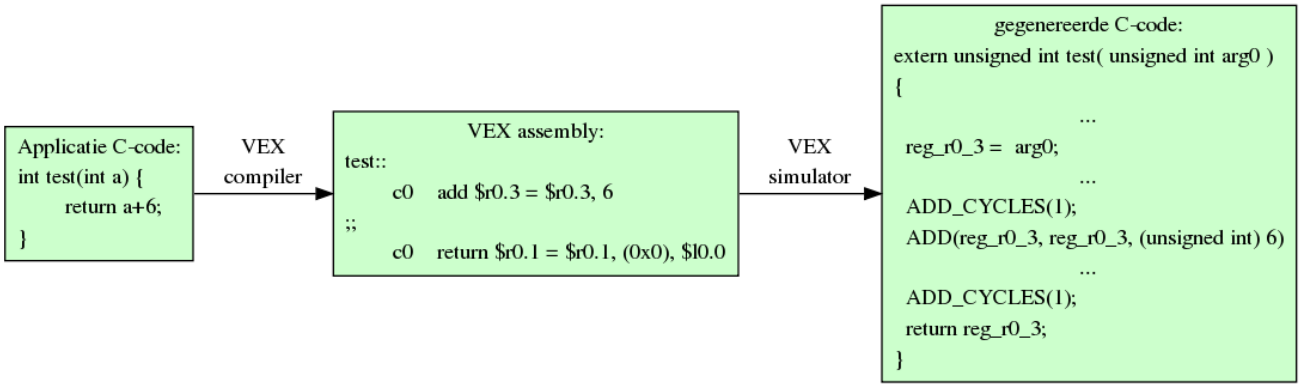
Een VLIW-processor biedt de mogelijkheid om operaties parallel uit te voeren. Het is echter een uitdaging voor de compiler om dit parallellisme goed te benutten, want alle operaties kunnen niet zomaar parallel in een instructie worden gezet. Bij het compileren wordt uitgezocht welke operaties onafhankelijk van elkaar zijn, deze kunnen dan in principe tegelijk uitgevoerd worden. Elk issue slot in een instructie dat geen operatie heeft om uit te voeren wordt gevuld met een *nop*. Elke operatie zal in de processor naar de bijbehorende functional unit worden doorgespeeld, om daarin uitgevoerd te worden. De compiler moet er voor zorgen dat in een instructie niet meer FU's worden gebruikt dan aanwezig zijn.

Van elke FU kan in de configuratie de *delay* worden aangegeven, dus hoeveel klokcycli de FU nodig heeft voor het uitvoeren van de operatie. Hiermee moet rekening gehouden worden bij het inroosteren van de operaties, omdat afhankelijke operaties dit aantal cycli moeten wachten op het resultaat van de voorgaande operatie.

5.2.3 VEX simulator

De eerste stap in het ontwerp van de demonstrator was de applicatie uitvoeren op de simulator. Door te profileren met de simulator kan een goed beeld gekregen worden van de applicatie en welke functies daarbinnen de bottle-neck vormen. Een ander doel van het simuleren is om uit te zoeken welke VEX-configuratie het meest geschikt is voor de applicatie.

De simulator van de VEX maakt het mogelijk op een pc een VEX-processor te simuleren, en biedt daarbij mogelijkheden om uitgebreide statistieken bij te houden. De simulator betreft een *compiled simulator*. Om een programma te simuleren wordt eerst de compiler gebruikt om de C-code te compileren naar VEX-assembly (de *.s*-bestanden), voor een meegegeven VEX-configuratie. De VEX-assembly wordt vervolgens weer omgezet naar C-code. Hierbij wordt elke assembly-instructie omgezet naar enkele regels C-code met dezelfde semantiek, en worden veel extra code toegevoegd om de statistieken bij te houden. In figuur 5.2 zijn deze stappen weergegeven, met versimpelde voorbeeldcode bij elk stap. Deze gegenereerde C-code wordt ten slotte gecompileerd voor het host-platform, oftewel de pc waarop de simulatie wordt uitgevoerd. Dit gebeurt met de compiler van de host, bijvoorbeeld *gcc*. Via deze omweg kan de applicatie op de pc worden uitgevoerd, maar worden daarbinnen de VEX-instructies gesimuleerd. Om het mogelijk te maken om bij het simuleren functies uit libc aan te roepen worden de libraries van de pc gebruikt.



Figuur 5.2: Stappen van de VEX simulator

Aan de gegenereerde C-code wordt extra code toegevoegd om statistieken van de simulatie bij te houden. Hiermee wordt bijvoorbeeld bijgehouden hoeveel operaties worden uitgevoerd en hoeveel branches worden

genomen. Zo kan worden geanalyseerd hoe de applicatie presteert op de gekozen VEX-configuratie. De simulator maakt het ook mogelijk om de applicatie deterministisch te profileren. Tijdens het uitvoeren van de simulatie wordt een bestand genaamd *gmon.out* aangemaakt, waarin de tijdsstatistieken van elke functie worden gezet, die met behulp van *gprof* kunnen worden uitgelezen. De instrumentatie voor het profileren met de VEX simulator is niet invasief, en heeft dus geen invloed op de meting zelf.

5.3 Beschrijving ρ-VEX

De ρ-VEX is een reconfigureerbare implementatie van de VEX-architectuur, ontwikkeld op de afdeling Computer Engineering van de TU Delft. Een aantal variabelen die bij de VEX-specificatie open zijn gehouden, zijn ook bij de ρ-VEX instelbaar. Om uit te zoeken welke configuratie voor de ρ-VEX zal worden gebruikt, is de applicatie gesimuleerd op verschillende VEX-configuraties. Eerst zal de architectuur van de ρ-VEX worden besproken (5.3.1). Vervolgens zal aan de hand van simulatieresultaten een keuze worden gemaakt voor de configuratie van de ρ-VEX (5.3.2).

5.3.1 Architectuur van implementatie

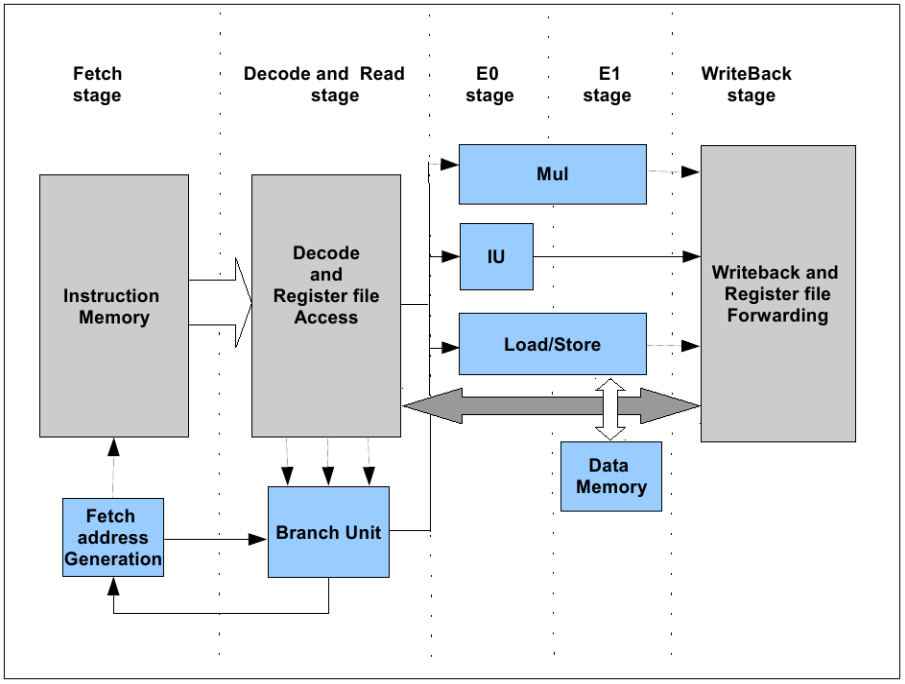
De ρ-VEX is een Harvard-architectuur, dus met gescheiden instructie- en datageheugen. De eerste versie was een multi-cycle machine. De versie die in dit project wordt gebruikt heeft een pipeline van vijf stappen. In figuur 5.3 is de pipeline schematisch weergegeven. De stappen zijn:

1. Instructie ophalen
2. Instructie decoderen, operands inlezen
3. Executie0: Operaties uitvoeren in de betreffende FU's
4. Executie1: Tweede stap van uitvoeren (voor multiplicaties), interactie met geheugen
5. Terugschrijven naar registergeheugen

Doordat hij geparametriseerd is, is het mogelijk de configuratie van de ρ-VEX aan te passen aan de gebruikte applicatie. De volgende eigenschappen zijn geparametriseerd op de ρ-VEX:

- Variabele issue-width
- Mogelijkheid tot forwarding
- Aantal en type van FU's per issue slot

Voor elk issue slot kan worden aangegeven welke FU's in dat slot aanwezig zijn. Het aantal load-store units staat wel vast, dit kan er maar één zijn. Normaal gesproken wordt bij elk issue slot een ALU neergezet, bij sommigen ook een multiplier. Op de ρ-VEX is de mogelijkheid tot *forwarding* geïmplementeerd. Forwarding is een techniek om het ILP dat de pipeline uitbuit te verhogen, door de uitkomsten van de functional units direct beschikbaar te stellen voor de volgende operatie. De uitkomst hoeft hierdoor niet meer eerst weggeschreven te worden in een register.



Figuur 5.3: Pipeline van de ρ-VEX (bron: [2])

5.3.2 Gekozen configuratie

Om de configuratie voor de ρ-VEX te kiezen is de applicatie op verschillende VEX-configuraties gesimuleerd. De belangrijkste keuze is de issue width, daarom zijn simulaties met verschillende issue widths gedraaid. Naar het gebruik van forwarding is later gekeken. Het aantal ALU's is steeds gelijk gehouden aan het aantal issue slots, en het aantal multipliers op de helft daarvan. De andere parameters zijn gelijk gehouden, de standaardinstellingen zijn hiervoor gebruikt. Gekeken is naar hoeveel klokcycli de applicatie in de simulatie nodig had om een afbeelding van 1024 bij 768 pixels te decoderen. De resultaten zijn uitgezet in figuur 5.4.

De grootste winst wordt geboekt met de stap van één naar twee issue-slots. De winst die gemaakt werd wanneer er vier issue-slots gebruikt werden in plaats van twee, is ongeveer vijf procent. Daarna wordt de winst snel insignificant, omdat de parallellisatie beperkt wordt door de afhankelijkheid tussen de operaties. Vooral de beperking dat er maar één load-store unit is zal hier waarschijnlijk voor de limiet zorgen. De exacte getallen zijn te vinden in tabel 5.1.

Gekozen is om vier issue slots te gebruiken. Het snelheidsvoordeel boven twee issue slots is niet enorm, maar de hoeveelheid hardware die dit kost is nog aanvaardbaar. Een belangrijke reden is ook dat het ruimte geeft om met optimalisaties aan de software de applicatie verder te versnellen. Bij gebruik van twee issue slots zou de processor meestal al volledig in gebruik zijn en de snelheidslimiet dus al bijna bereikt.

Het was geen interessante optie om van vier naar acht issue slots te gaan. De winst die daarmee gemaakt kon worden was nog geen drie procent, terwijl er vier extra issue slots nodig zijn.

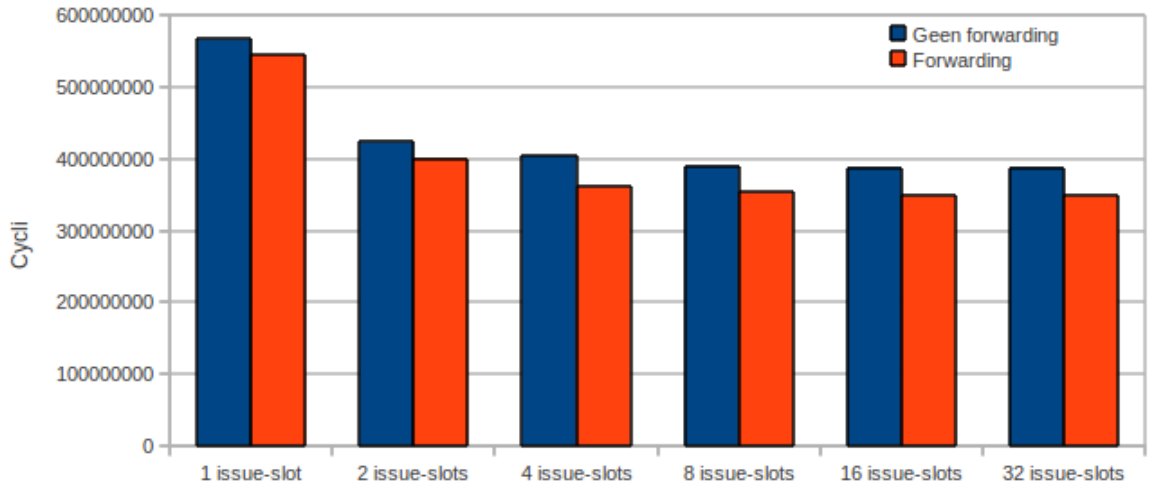
Tabel 5.1: Resultaten VEX simulaties

Aantal klokcycli dat de applicatie nodig heeft

Aantal issue-slots:	1	2	4	8	16	32
Geen forwarding	$5,67 \cdot 10^8$	$4,24 \cdot 10^8$	$4,04 \cdot 10^8$	$3,90 \cdot 10^8$	$3,88 \cdot 10^8$	$3,88 \cdot 10^8$
Forwarding	$5,45 \cdot 10^8$	$4,01 \cdot 10^8$	$3,61 \cdot 10^8$	$3,53 \cdot 10^8$	$3,48 \cdot 10^8$	$3,48 \cdot 10^8$

Naast de configuratie met standaardinstellingen is ook gesimuleerd met kortere delays in de FU's, corresponderend met forwarding in de ρ-VEX. De juiste instellingen zijn bepaald aan de hand van het configuratiebestand dat beschreven is in [2], met de correctie dat de delay van de multiplier op één klokcyclus is gezet.

De simulaties van de geforwarde core zijn ook in de tabel en grafiek opgenomen. Er was een vermoeden dat het kiezen voor een geforwarde core de keuze voor de issue width had kunnen beïnvloeden. Daarom zijn de combinaties van de twee parameters onderzocht. Forwarding bleek voor alle instellingen wat betreft issue width een behoorlijke versnelling op te leveren. Er is dus gekozen voor een core van vier issue-slots met forwarding.



Figuur 5.4: Resultaten VEX simulaties

De keuze voor deze configuratie is gebaseerd op een simulatie van de hele applicatie. Verschillende functies binnen de applicatie kunnen een andere hoeveelheid ILP bevatten. Wanneer niet het hele programma op de ρ-VEX uitgevoerd wordt, maar een andere partitionering gekozen wordt, kan dit invloed hebben op de overweging van het aantal issue-slots. Er wordt uitgegaan van de aanname dat deze invloed niet zo groot is dat het een andere keuze zal opleveren voor het aantal issue-slots. Eventueel zou na het partitioneren van de code opnieuw kunnen worden gesimuleerd, om te controleren of de keuze voor de issue width nog optimaal is, en om eventueel het aantal issue slots bij te stellen. Daarna zou zelfs de partitionering weer beschouwd kunnen worden. Zo zou een iteratief ontwerpproces kunnen worden aangehouden.

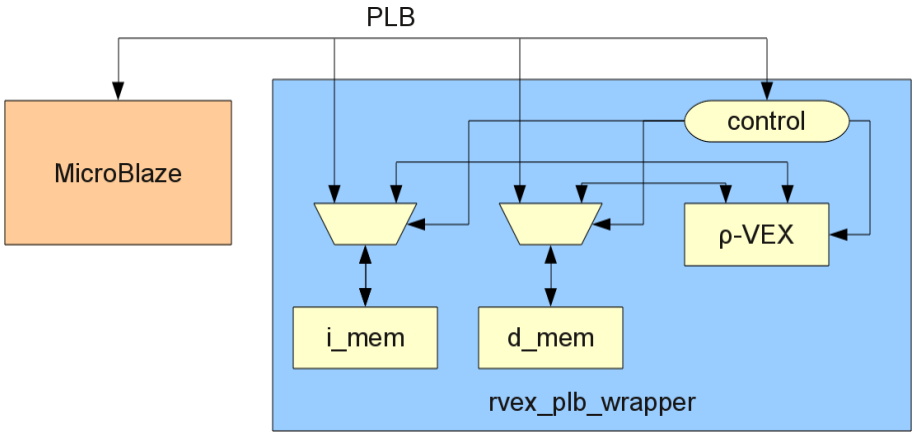
5.4 Beschrijving systeemarchitectuur

Om de ρ-VEX als accelerator te gebruiken naast de MicroBlaze, moet een platform worden gemaakt waarin deze processoren kunnen communiceren, zodat de invoerdata en resultaten kunnen worden uitgewisseld. Om de MicroBlaze en ρ-VEX met elkaar te verbinden wordt de *Processor Local Bus* (*PLB*) gebruikt. Via deze bus kan de MicroBlaze de geheugens van de ρ-VEX lezen en schrijven, en de ρ-VEX starten en stoppen. Zo kan de MicroBlaze een programma en invoerdata in de ρ-VEX zetten, de ρ-VEX starten, en als deze klaar is het resultaat weer uitlezen. Eerst zal nu in worden gegaan op de werking van deze verbinding (5.4.1), daarna op de methode waarmee dit systeem wordt geprogrammeerd (5.4.2). Dan volgt de methode om de executietijd te meten (5.4.3).

5.4.1 Verbinding tussen de processoren

Er zijn meerdere mogelijkheden om de communicatie tussen de processoren te bewerkstelligen. Een gedeeld geheugen zou kunnen worden gebruikt waaruit beide processoren kunnen lezen en in kunnen schrijven. Nadeel daarvan is dat beide processoren dezelfde geheugenbus zouden moeten gebruiken, waarvoor een arbiter nodig is, of het geheugen moet meerdere poorten hebben en met meerdere bussen verbonden zijn, wat complex wordt. Een andere optie is om twee *pipes* (FIFO buffers) te gebruiken, één per communicatierichting. In dit onderzoek is een simpelere aanpak aangehouden, met als reden dat deze reeds geïmplementeerd was. Bij deze methode kan de MicroBlaze de geheugens van de ρ-VEX zelf lezen en schrijven, zolang de ρ-VEX niet actief is. Deze geheugens zijn daarvoor aangesloten aan de communicatiebus van het platform, de PLB. Een nadeel hiervan is dat tijdens de communicatie de ρ-VEX niets kan doen, hoewel dit het gebruik wel iets makkelijker maakt.

Om dit systeem te maken wordt met *Xilinx Platform Studio* een platform gegenereerd. Dit bestaat uit een MicroBlaze processor met toebehoren zoals beschreven in 5.1, en de *rvex_plb_wrapper* met daarin de ρ-VEX. De *Processor Local Bus* verbindt de rvex_plb_wrapper met de MicroBlaze. De MicroBlaze is de *master* op dit busnetwerk. Aan elk component dat aan de PLB hangt worden adresgebieden toegekend. Via deze adressen kan de master de gewenste componenten aanspreken. De rvex_plb_wrapper heeft drie adresgebieden.



Figuur 5.5: Verbinding tussen MicroBlaze en ρ-VEX

In figuur 5.5 is de verbinding tussen de MicroBlaze en ρ-VEX schematisch weergegeven. De wrapper

verbindt het instructiegeheugen (*i_mem*) en datageheugen (*d_mem*) van de ρ-VEX met de PLB. Ook biedt het een *control* register aan om de ρ-VEX te bedienen. Door de juiste bits naar dit register te schrijven kan de ρ-VEX via een simpele finite state machine gestart, gestopt of gereset worden. Ook kan via dit register worden gelezen of de ρ-VEX al klaar is en wat de waarde van zijn program counter is.

Het instructiegeheugen en datageheugen van de ρ-VEX kunnen of met de PLB, of met de ρ-VEX worden verbonden. Ze zijn verbonden met de ρ-VEX als het *run*-bit aan is, dus als de ρ-VEX aan het werk is. Zodra de ρ-VEX klaar is met het uitvoeren van zijn programma, krijgt de MicroBlaze een interruptsignaal, en worden deze geheugens aan de PLB verbonden. De MicroBlaze kan ze dan weer lezen en schrijven. Terwijl de ρ-VEX bezig is, is geen communicatie mogelijk tussen de twee processoren.

5.4.2 Programmeermethode

De MicroBlaze wordt gebruikt als CPU in het systeem, hierop wordt met de SDK het totale programma geladen. Het programmeren van de ρ-VEX moet via de MicroBlaze gebeuren. Een stuk code, geschreven door Martijn de Zeeuw, wordt in de MicroBlaze gezet, en verzorgt de bediening van de ρ-VEX. Deze code laadt het ρ-VEX-programma in het instructiegeheugen van de ρ-VEX, laadt de initiële data in zijn datageheugen, verifieert eventueel of deze geheugens correct geprogrammeerd zijn, en start de ρ-VEX. Vervolgens wacht de MicroBlaze op het interruptsignaal, dat door de rvex_plb_wrapper wordt genereerd zodra de ρ-VEX klaar is en een *stop*-operatie uitvoert. Intussen telt het programma in de MicroBlaze ook of het uitvoeren niet te lang duurt, en kapt hij het af als zijn teller een ingesteld maximum overschrijft.

In het programma van de MicroBlaze moet dus het programma van de ρ-VEX opgeslagen zijn. Om alles te compileren wordt eerst het programma van de ρ-VEX gecompileerd met de VEX compiler, waarbij de gebruikte configuratie van de ρ-VEX wordt meegegeven. Door de ρ-VEX-toolchain wordt de assembly-code die de VEX compiler produceert geassembleerd, en omgezet in een C-broncode, *prog.h*. In deze C-code worden twee arrays gedefinieerd met daarin de inhoud van het instructiegeheugen en van het datageheugen. Deze C-code wordt opgenomen in het programma van de MicroBlaze, zodat deze de arrays kan inladen in de ρ-VEX. Als de ρ-VEX klaar is met uitvoeren, kan de MicroBlaze de resultaten uit het datageheugen van de ρ-VEX lezen.

5.4.3 Executietijd meten

Voor het testen van het systeem is het wenselijk om te kunnen meten hoeveel klokcycli nodig zijn om de applicatie uit te voeren. Een simpele methode is om in software een teller bij te houden. Deze methode werkt echter alleen om op de MicroBlaze te tellen hoeveel cycli de ρ-VEX nodig heeft. Om te weten hoelang het versturen van de data tussen MicroBlaze en ρ-VEX duurt, en om te kunnen meten hoe lang de MicroBlaze over iets doet, is een externe teller nodig. Hiervoor is aan de PLB een *timer/counter* peripheral verbonden. Vanuit de MicroBlaze kan deze worden gereset en gestart, en wanneer gewenst weer gestopt en uitgelezen. De teller telt met 100MHz, wat ook de klokfrequentie van de MicroBlaze en ρ-VEX is, dus de uitkomst van de meting is het aantal klokcycli dat voorbij is gegaan. Enkele cycli worden meegeteld die gebruikt worden voor het starten en stoppen van de teller, maar deze hoeveelheid is insignificant.

Hoofdstuk 6

Software/Hardware Mapping

Dit hoofdstuk beschrijft de stappen die genomen zijn om de software op de hardware toe te passen. Het doel is de hardware zo goed mogelijk te benutten, om de applicatie zo snel mogelijk te maken.

In hoofdstuk 4 is beschreven hoe de applicatie is aangepast om haar werkend te krijgen op te ρ-VEX en MicroBlaze. In hoofdstuk 5 is de keuze voor de hardware-configuratie gemaakt. De laatste stap is nu om de applicatie op deze hardware te laten werken, en haar ook zo snel mogelijk te laten werken. In paragraaf 6.1 worden de stappen beschreven die genomen zijn om de applicatie op het platform te laten werken. Hierin wordt de keuze gemaakt hoe de applicatie wordt verdeeld tussen ρ-VEX en MicroBlaze, en wordt beschreven tegen welke problemen aan is gelopen. Paragraaf 6.2 geeft antwoord op de vraag of, en in welke mate, de software geoptimaliseerd kan worden om de VLIW-eigenschap van de ρ-VEX beter te benutten. Het is uiteindelijk niet gelukt om de applicatie werkend te krijgen op de ρ-VEX, waardoor het niet gelukt is om de totale speedup in praktijk te meten. In paragraaf 6.3 worden daarom de VEX-simulatieresultaten vergeleken met hoe de applicatie op de MicroBlaze presteert.

6.1 Realisatie

Het implementeren van de applicatie op de hardware-platformen verliep niet zo soepel als gehoopt. Op de VEX simulator en MicroBlaze werkte de applicatie wel, maar het is helaas niet gelukt om haar werkend te krijgen op de ρ-VEX.

Zoals beschreven in 4.4, is de applicatie eerst aangepast om geen libc-functies meer aan te roepen. Van deze gestripte versie van de applicatie wordt in dit hoofdstuk uitgegaan. Eerst zal worden beschreven hoe de applicatie op de MicroBlaze is uitgevoerd (6.1.1). Vervolgens wordt beschreven hoe de software over het systeem van twee processoren wordt verdeeld (6.1.2), en wordt uitgelegd hoe de implementatie hiervan is verlopen (6.1.1).

6.1.1 Applicatie op MicroBlaze

Na het analyseren en simuleren van de applicatie, was de eerstvolgende stap om de applicatie op de MicroBlaze uit te voeren. Een platform met een MicroBlaze is gegenereerd, zoals beschreven in 5.1. In de SDK kan gekozen worden of het programma in het interne geheugen in de FPGA wordt gezet, of in het externe geheugen naast de FPGA. Dezelfde keuze kan gemaakt worden voor de plaatsing van het datageheugen.

Omdat het programma en datageheugen niet samen in het interne geheugen passen, zijn ze in het externe geplaatst.

Toen de nodige aanpassingen aan de applicatie stuk voor stuk waren gedaan, liep de applicatie goed op de MicroBlaze. Het enige probleem was dat het lastig was om te controleren of de resulterende bitmap correct was. Hiervoor wordt de array die het bestand representeert via de UART in hexadecimale vorm naar de pc gestuurd. Een scriptje is geschreven dat deze representatie omzet in een bitmap-bestand, om deze vervolgens op het scherm van de pc te kunnen tonen. Vanwege de lage bitrate van de UART (9600 bits per seconde), is besloten alleen kleine afbeeldingen te gebruiken. Het maken van een andere vorm van connectie met de computer, of het aansluiten van een scherm aan het FPGA-bord, zou veel tijd kosten, en het concept blijft het zelfde met groot of klein plaatje. In het hele onderzoek is als afbeelding *surfer.jpg* gebruikt (figuur 4.2).

6.1.2 Partitionering ρ-VEX/MicroBlaze

Bij het partitioneren wordt een verdeling van de applicatie over de twee processoren gezocht. Een deel van de berekeningen wordt uitgevoerd op de accelerator, in dit geval de ρ-VEX.

Methode

De code van de applicatie wordt in twee delen gesplitst, waarbij het deel uit de applicatie wordt genomen dat geaccelereerd moet worden. Als een hele functie wordt geaccelereerd is het simpel om de code te splitsen, en anders moet de code iets meer aangepast worden. Er wordt nu zonder verlies van algemeenheid van uitgegaan dat een hele functie genomen wordt. Het deel van de applicatie dat op de MicroBlaze blijft draaien krijgt een extra stuk code op de plaats waar de te accelereren functie is uitgenomen. Door deze code wordt de data die de geaccelereerde functie nodig heeft naar de ρ-VEX gestuurd, wordt gewacht tot deze klaar is met zijn deel, en wordt het resultaat weer uit de ρ-VEX gelezen. De data die moeten worden overgestuurd zijn de argumenten die mee werden gegeven bij het aanroepen van de functie, globale variabelen die in de geaccelereerde functie worden gelezen, en data die door middel van pointers worden gelezen. Voor deze laatste categorie is het nodig om de code goed te begrijpen, omdat dit in principe elke geheugenplek zou kunnen zijn. Het programma dat op de ρ-VEX draait, roept vanuit *main* de geaccelereerde functie aan. De argumenten voor de functie moeten uit globale variabelen gelezen worden, die door de MicroBlaze met de juiste data gevuld zijn. Na het uitvoeren van de functie wordt de eventuele *return value* in een globale variabele gezet om uitgelezen te kunnen worden door de MicroBlaze.

Terwijl de accelerator bezig is wordt in dit onderzoek de CPU niet gebruikt. Hierdoor blijft het partitioneren overzichtelijk, en kan de snelheid van het systeem met accelerator eerlijk vergeleken worden met het uitvoeren zonder accelerator.

Keuze

Bij het kiezen van een verdeling is belangrijk hoeveel sneller dan de CPU de accelerator de te accelereren functie kan uitvoeren. Zoals in formule 3.1 omschreven moet echter ook rekening gehouden worden met de communicatie-overhead die het partitioneren oplevert.

Het is niet mogelijk om data naar het geheugen van de ρ-VEX te schrijven terwijl deze een functie uitvoert. Wanneer dit wel het geval zou zijn, zou een goede keus kunnen zijn om een functie te versnellen die vaak aangeroepen wordt. Een voorbeeld is een functie die per MCU wordt aangeroepen. Wanneer de data voor de eerste MCU verstuurd is, zou de ρ-VEX deze MCU kunnen verwerken, terwijl de volgende MCU reeds verstuurd wordt. Zo zou de communicatie-overhead beperkt blijven.

Dit is voor de ρ-VEX (nog) niet mogelijk. De communicatie-overhead is dus slechts afhankelijk van de hoeveelheid data die wordt gebruikt in de gekozen functie, en de hoeveelheid die deze functie teruggeeft. Of de data in stukjes kan worden verstuurd maakt dus niet uit. Bij het decoderen van JPEG zou een logische partitionering kunnen zijn om een van de drie decodeer-stappen te versnellen, of een deel van een stap, aangezien in deze functies veel tijd zit. Elk van deze stappen werkt op alle pixels, en heeft dus veel data nodig. Om deze functies te versnellen moeten de pixelmatrices bij de ρ-VEX aankomen, of daar uitgerekend worden. Het JPEG-bestand is de kleinste representatie van de afbeelding die in de applicatie voorkomt, immers op dat moment is de compressie nog volledig. Na de eerste stap van het decoderen (de run-length en Huffman-decodering), is de representatie van de afbeelding groter geworden, en deze zal even groot blijven bij latere stappen. Per pixel blijven er drie waardes van één byte nodig. De minste communicatie kan dus verkregen worden door de JPEG-afbeelding nog onverwerkt van de MicroBlaze naar de ρ-VEX te versturen.

Relatief zijn juist de laatste functies (kleurconversie en IDCT) goed parallelliseerbaar, zoals te zien in figuur 4.3. Dit maakt het interessant om ook deze functies op de ρ-VEX uit te voeren. De MicroBlaze zal functies waarschijnlijk ook niet sneller uitvoeren dan de ρ-VEX, dus als de hoeveelheid data niet verandert, dan is er geen reden om niet op de ρ-VEX verder te gaan met de dataverwerking. Er is dus gekozen om alle decoderingsstappen op de ρ-VEX uit te voeren. De accelerator wordt dus ook maar één keer aangeroepen.

6.1.3 Applicatie op MicroBlaze en ρ-VEX

Opsplitsing applicatie

Bij het partitioneren is gekozen vrijwel de hele code van de applicatie op de ρ-VEX te zetten. In het programma voor de MicroBlaze wordt de code gezet voor de bediening van de ρ-VEX. Dit programma laadt eerst het ρ-VEX-programma in zijn instructiegeheugen, en vervolgens zijn globale variabelen, inclusief het JPEG-plaatje, in het datageheugen. De ρ-VEX kan vervolgens gestart worden.

Om het resultaat van de ρ-VEX te verkrijgen leest de MicroBlaze, zodra de ρ-VEX klaar is, zijn datageheugen uit. Hiervoor moet natuurlijk wel bekend zijn waar in het datageheugen het gezochte resultaat staat, in dit geval waar de array *bmp_file* is neergezet. Om dit adres te weten te komen wordt, na het compileren van het programma van de ρ-VEX, middels *objdump* het adres van *bmp_file* uit het programma geëxtraheerd. Dit adres wordt vervolgens ingevuld in de broncode van het programma van de MicroBlaze. Op deze manier weet de MicroBlaze waar in het datageheugen van de ρ-VEX hij deze array kan vinden. Ook kan met *objdump* de plaats van *jpeg_file* worden verkregen. Door vanuit de MicroBlaze deze array te vullen met een nieuw bestand, kan een volgend plaatje worden gedecodeerd. Het instructiegeheugen hoeft maar één keer te worden volgeladen.

Aanpassingen aan ρ-VEX-geheugen

Om de MicroBlaze en ρ-VEX te koppelen is het platform gegenereerd zoals beschreven in 5.4. Het genereren van dit platform lukte, en het uitvoeren van een simpele testapplicatie werkte zoals verwacht. Bij het uitvoeren van de decoder-applicatie bleken het instructiegeheugen en datageheugen van de ρ-VEX te klein te zijn. De limiet was 512 instructies, terwijl ruim 2000 instructies werden gebruikt. Ook het datageheugen was veel te klein. In de VHDL-code van de ρ-VEX zijn de geheugeninstanties aangepast om een groter adresbereik aan te kunnen. Het instructiegeheugen is geconfigureerd om ruimte te hebben voor 4096 instructies, dus gebruikmakend van 12-bits-adressen in plaats van 10-bits. Het datageheugen is vergroot naar 16384 bytes, met 14-bits-adressen. Het lukte de synthese echter niet om dit in de FPGA te passen in combinatie met de geldende snelheidseis, 100MHz-operatie. Om dit te overkomen is overgestapt naar de huidige werkversie

van de ρ-VEX, die bij dezelfde geheugenvergroting niet dit probleem vertoonde. Deze versie is echter nog in ontwikkeling en bevatte nog fouten. De interface naar het instructie- en datageheugen zijn hierin veranderd, en sloten niet goed aan op de rvex_plb_wrapper, waardoor de geheugens niet goed konden worden geschreven en gelezen. Na een aantal dagen foutzoeken en met hulp van de ρ-VEX-ontwikkelaars, is de geheugenaansluiting uiteindelijk werkend gekregen met een groter instructie- en datageheugen. Tussendoor zijn ook enkele andere fouten in de ρ-VEX gevonden en verbeterd, zoals fouten in de configureerbaarheid van de issue width.

Testen van de applicatie

Een kleine testapplicatie leek te werken op deze nieuwe configuratie. Bij het testen van de JPEG-decoder-applicatie bleef de ρ-VEX echter in een lus steken. Doordat tijdens het uitvoeren op de ρ-VEX geen communicatie ermee mogelijk is, werd het foutzoeken erg belemmerd. Om de fout te vinden is de ρ-VEX gesimuleerd in *ModelSim*. Door daarin naar de waarde van de program counter te kijken kon worden opgespoord waar in het programma de processor vast bleef zitten. Dit bleek in een simpele lus te gebeuren in de functie *JpegToBmp*, bij het combineren van de drie pixel-matrices na de decodeerstappen. Door nog onbekende fout leek de processor verkeerde waardes uit het geheugen te lezen, waardoor de lus veel te vaak bleef herhalen. Aangezien deze verkeerd gelezen variabele de breedte van het plaatje was, is deze in de code vervangen door constante waarde 32, wat mogelijk was omdat steeds één plaatje werd gebruikt van 32 pixels breed. De simulatie in ModelSim bleef nu niet meer steken. Frappant genoeg bleef de ρ-VEX in de FPGA wel nog steeds steken, mogelijk op een andere plek in het programma. Ook bleek de simulatie in modelsim in ruim vijf keer minder cycli te eindigen dan volgens de VEX-simulatie zou moeten, dus ging er nog steeds iets mis. Ter controle is de uitvoer van het programma bekeken. In de array *bmp_file* wel de BMP-header ingevuld te zijn, maar was de inhoud verder een compleet zwart plaatje.

Al met al bleek dat ergens in de processor een fout zit, of eventueel ergens in de ρ-VEX toolchain, waardoor het programma niet helemaal correct wordt uitgevoerd. De subtiliteit van de fout en de inconsistentie tussen testen in FPGA en in ModelSim, maken het erg lastig om de fout te vinden. Opvallend is dat de kleinere testapplicatie, die differentiële pulscodemodulatie berekende, wel werkte. Waar de fout precies zit is op moment van schrijven nog niet achterhaald. Voor het onderzoek is daarom verder de VEX-simulator gebruikt. Met deze simulator is onderzocht hoe de applicatie kan worden geparallelliseerd. Dit wordt in de volgende paragraaf beschreven.

6.2 Software-optimalisatie

Zoals in subparagraaf 5.2.2 beschreven is het de taak van de compiler om de operaties in te roosteren, en daarbij de meerdere issue slots zo goed mogelijk te benutten. De assembly-code die de VEX compiler genereert is bestudeerd, waarbij duidelijk werd dat deze code niet optimaal was. Op een aantal plekken had de compiler onafhankelijke operaties sequentieel ingeroosterd, wat resulteert in suboptimaal gebruik van de VLIW-processor. Hieruit is geconcludeerd dat de optimalisaties van de compiler efficiënter kunnen worden uitgevoerd. Verschillende optimalisatietechnieken die de compiler uitvoert zijn onderzocht. Hierbij is gezocht naar de technieken die parallellisatie van de code verhogen, en door de compiler niet optimaal worden uitgevoerd. Onderzocht is hoe middels het aanpassen van de C-code kan worden gezorgd dat de compiler efficiëntere code genereereert. Deze paragraaf bevat resultaten van de optimalisaties. Hiervoor is de VEX-simulator gebruikt, geconfigureerd als de ρ-VEX, zoals beschreven in 5.3.2.

Een aantal optimalisatietechnieken doet de compiler zelf al goed. De technieken *expressie-vereenvoudiging* en *dead code elimination* worden, bleek bij het onderzoeken van de assembly code, goed uitgevoerd.

Een techniek die vooral voor een VLIW-processor erg nuttig kan zijn is *loop unrolling*. De mate van loop unrolling die de compiler toepast is met flags in te stellen. Hierbij is de ideale instelling gekozen. Echter bleek dit niet overal optimaal uitgevoerd. In de C-code is vervolgens een aantal loops handmatig unrolled.

Een andere vorm van optimalisatie waarvan gebruik is gemaakt, is *procedure inlining*. Hiervoor zijn verschillende methoden die de compiler zelf uitvoert. Uiteraard is het ook mogelijk om deze techniek zelf in de C-code toe te passen.

Deze paragraaf geeft antwoord op de vraag of het parallellisme van de ρ-VEX beter uitgebuit kan worden, door de applicatie specifiek voor de ρ-VEX aan te passen. Hiervoor wordt eerst ingegaan op loop unrolling in paragraaf 6.2.1. Vervolgens wordt in paragraaf 6.2.2 onderzocht wat de invloed van procedure inlining op de executietijd is.

6.2.1 Loop unrolling

Loop unrolling is een optimalisatietechniek die helpt het parallellisme verder uit te buiten [12]. Loops of lussen zijn veelgebruikte structuren, inherent aan de loop is de branch die aan het eind genomen moet worden om al dan niet nog een keer de inhoud van de loop uit te voeren. Daarnaast bevat een loop per definitie een sequentiële component. Immers wordt de loop meerdere keren achter elkaar uitgevoerd.

Vaak is tijdens het compileren al bekend hoe vaak de loop uitgevoerd dient te worden. Wanneer dit het geval is kunnen de berekeningen die binnen de loop stonden simpelweg onder elkaar gezet worden. Hierdoor is het niet meer nodig om steeds opnieuw de uitkomst van de branch te bepalen. Als de operaties binnen de loop onafhankelijk zijn van de voorgaande iteratie is er nog een tweede voordeel. In plaats van een loop, die altijd sequentieel uitgevoerd wordt, is er nu sprake van simpelweg een aantal onafhankelijke berekeningen die parallel ingeroosterd kunnen worden, een situatie waar de meerdere issue-slots van de ρ-VEX goed gebruikt kunnen worden.

In de JPEG-decoder komt dit soort loops regelmatig voor. Vaak gaat het hierbij om een loop die pixel voor pixel, of MCU voor MCU een bepaalde berekening uitvoert. Deze berekeningen vertonen data-level parallellisme, want een operatie op een pixel of MCU is meestal onafhankelijk van de pixels of MCU's ernaast. Door middel van loop unrolling is dit data-level parallellisme om te zetten in instructie-level parallellisme. Dit maakt deze optimalisatie erg interessant voor het uitvoeren van deze applicatie op een ρ-VEX.

Uiteraard kleven er ook nadelen aan loop-unrolling. Het voornaamste nadeel is dat de hoeveelheid instructies groeit. Immers wordt er, in plaats van dezelfde code telkens opnieuw aan te roepen, voor elke iteratie een extra stuk code gegenereerd. Hierdoor is meer instructiegeheugen nodig. Als er instructie-caching wordt gebruikt kan loop unrolling zelfs een vertraging opleveren omdat er meer instructies uit het geheugen moeten worden ingeladen. Caching wordt in dit onderzoek niet gebruikt.

Compiler instellingen

Om de invloed van de loop unrolling van de compiler te testen is de applicatie getest met een aantal verschillende gradaties van loop unrolling. Om de andere optimalisaties niet te beïnvloeden is hiervoor de flag -H*n* gebruikt. Met deze flag kan de loop unrolling onafhankelijk van andere optimalisaties ingesteld, door *n* een waarde te geven tussen 0 en 4 [19]. In tabel 6.1 staat het aantal benodigde cycli uitgezet tegen de verschillende niveaus van loop unrolling. Het beste resultaat werd bereikt door de flag -H2 te gebruiken. De metingen zijn verkregen door een afbeelding van 32 bij 24 pixels te decoderen.

Wanneer de unrolling-instelling op hoger dan H2 gezet werd, wogen de nadelen die gepaard gaan met urolling niet op tegen de voordelen. De relatieve speedup die totaal bereikt is met behulp van de loop-

Tabel 6.1: Invloed loop unrolling van compiler op executietijd

	-H0	-H1	-H2	-H3	-H4
# Cycli	$3,59 \cdot 10^5$	$3,26 \cdot 10^5$	$3,21 \cdot 10^5$	$3,22 \cdot 10^5$	$3,25 \cdot 10^5$

unrolling van de compiler is hiermee $\frac{3,59 \cdot 10^5}{3,21 \cdot 10^5} = 1,12$. Verder wordt deze instelling gebruikt als uitgangspunt voor de optimalisatie.

Handmatig optimaliseren

Om de code verder te optimaliseren is de assembly bestudeerd. Hierbij viel op een aantal plekken op dat er weinig parallellisatie aanwezig was. Deze stukken code zijn verder bekeken. Er zijn drie verschillende oorzaken gevonden:

- Veel load/store-operaties
- Suboptimale inroostering
- Operaties waren afhankelijk van elkaar

Aan de eerste en de laatste oorzaak was niet veel te doen, dit zijn beperkingen die in de aard van de applicatie zitten. De code op een andere manier schrijven kan hier niet op een structurele manier verbetering in brengen. De tweede oorzaak ligt bij imperfectie van de compiler. Door de C-code aan te passen moest het mogelijk zijn om de compiler deze operaties beter te laten inroosteren.

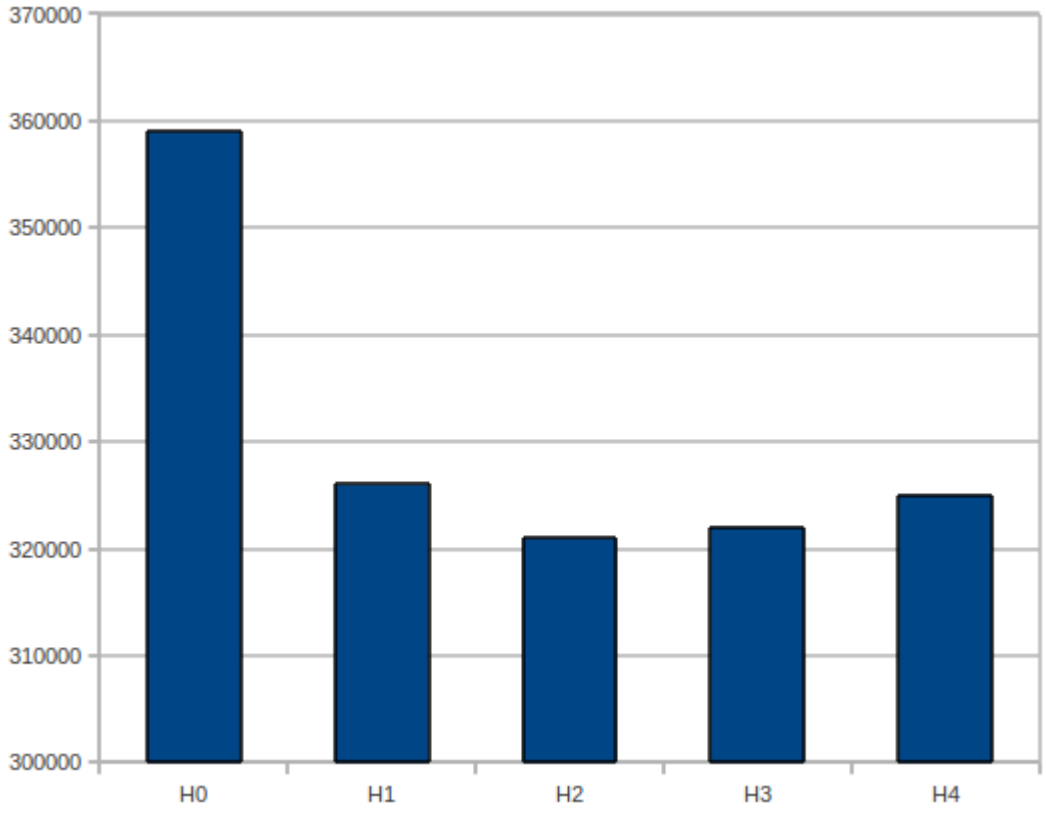
Aanpassingen

Er is begonnen met een stuk assembly-code waar de roostering duidelijk niet optimaal was. Dit stuk is gevonden in de functie *IDCT*. Deze assembly-code is te vinden in appendix B. In de assembly die te vinden is in paragraaf B.3 is goed te zien dat er een behoorlijke hoeveelheid code sequentieel wordt uitgevoerd. Hier zit een aantal load/store-operaties bij. Deze zijn uiteraard niet tegelijk uit te voeren, doordat de ρ-VEX slechts ondersteuning biedt voor één load- of store-operatie in één instructie. Er staan echter ook shift-left-instructies in dit stuk code. Deze kunnen wel tegelijk met een load- of store-operatie worden uitgevoerd.

Wanneer de bijbehorende C-code (paragraaf B.1) bekeken wordt, wordt duidelijk dat het hier om een loop gaat die acht keer uitgevoerd wordt. Hoewel de compiler deze loop uitrolt, wordt elke iteratie sequentieel uitgevoerd. De operatie binnen de loop is niet afhankelijk van de uitkomst van de vorige iteratie van de loop, dus het is wel mogelijk om de iteraties gedeeltelijk parallel uit te voeren, waardoor meer operaties in één instructie gecombineerd kunnen worden. Dit concept heet *software pipelining*.

De loop bestaat uit 3 operaties: inladen, bit-shiften en wegschrijven. De compiler genereert hiervoor de volgende code:

```
        c0    ldw $r0.6 = 0[$r0.6]
;;                                                          ## 7
;;                                                          ## 8
        c0    shl $r0.6 = $r0.6, 3
;;                                                          ## 9
        c0    stw 0[$r0.4] = $r0.6
```



Figuur 6.1: Invloed loop unrolling door compiler, getoond is executietijd in cycli

In VEX-assembly staat een dubbele puntkomma voor het eind van een instructie. Na de load-operatie in deze code wordt een cyclus gewacht (een *bubbel* in de pipeline geïnjecteerd) omdat de load-operatie nog niet klaar is.

Wanneer het woord dat voor de eerste loop gebruikt wordt is ingeladen, kan er geshift worden. Tijdens dit shiften kan het inladen voor de tweede loop al beginnen. Op deze manier kunnen alle shift-operaties samen met een load- of store-operatie in één instructie worden uitgevoerd. In de code van paragraaf B.4 is te zien dat de shift-instructies allemaal gecombineerd worden met andere operaties. De resulterende speedup binnen dit deel van de functie is $\frac{1}{1-\frac{1}{3}} = 1,5$. Eenderde van de operaties heeft immers geen eigen instructie meer nodig.

De loop is geoptimaliseerd door met de hand te unrollen, zoals te zien is in de appendix, in paragraaf B.2. De loop unrolling wordt hierdoor gedaan vóór het compileren. Bij het unrollen bleek dat de loop uit de C-code halen niet genoeg was. Naast het unrollen is de array die gebruikt wordt eerst in tussenvariabelen opgeslagen. Dit was nodig omdat anders het laden, shiften en opslaan alsnog sequentieel ingeroosterd werd. De precieze oorzaak hiervoor is onbekend, daarvoor is meer kennis nodig van de interne werking van de compiler.

De verkregen speedup is niet volledig toe te schrijven aan de parallellisatie van de shift-operaties. De

unrolling die de compiler toepast wordt niet erg efficiënt gedaan. De compiler berekent erg omslachtig de adressen van de elementen die ingeladen moeten worden. Hierdoor worden veel onnodige instructies toegevoegd, wat bij handmatig unrollen niet gebeurt.

De bovenstaande methode is naast de loop voor het verwerken van de rijen, ook voor het verwerken van de kolommen binnen de IDCT toegepast. De C-codes hiervan (origineel en aangepast) zijn ook te vinden in Appendix B.

Resultaten

Door deze loops efficiënter te implementeren is een meetbare speedup bereikt. De verwachting is dat vooral bij een VEX met vier issue-slots speedup te zien is. Om de speedup die veroorzaakt wordt door de extra parallellisatie, en de speedup die door andere optimalisatie bereikt wordt van elkaar te kunnen scheiden, zijn meerdere simulaties gedaan. Een simulatie op de 4-issue-slot VEX, geconfigureerd zoals beschreven in hoofdstuk 5 geeft de totaal bereikte speedup weer. Echter komt een gedeelte van deze speedup niet van parallellisatie.

Door ook een VEX met één issue-slot te simuleren wordt de speedup geïsoleerd die niet door extra parallellisme komt. Wanneer de relatieve speedup van deze twee simulaties vergeleken wordt, kan een conclusie getrokken worden over de speedup die bereikt wordt door de extra parallellisme. De meetgegevens zijn te vinden in tabel 6.2.

Tabel 6.2: Speedup totale applicatie door handmatige loop-unrolling

	Eén issue-slot	Vier issue-slots
# Cycli voor handmatig unrollen	$4,95 \cdot 10^5$	$3,21 \cdot 10^5$
# Cycli na handmatig unrollen	$4,80 \cdot 10^5$	$3,00 \cdot 10^5$
Relatieve speedup	1,031	1,070

Voor de een goede meting van de relatieve speedup van de functie zelf is *gprof* gebruikt. Eerst is de simulatie van de onaangepaste code gemaakt. De applicatie bevond zich 13,7% van de tijd binnen IDCT, de functie waarop de optimalisatie is toegepast. Bij de geoptimaliseerde code nam deze functie 6,78% van de tijd in. Met behulp van deze percentages is berekend hoeveel cycli de functie in beslag nam. Vervolgens is met deze twee getallen, aan de hand van vergelijking 3.2, de relatieve speedup berekend van de functie:

$$3,21 \cdot 10^5 \cdot 13,7\% = 4,40 \cdot 10^4$$

$$3,00 \cdot 10^5 \cdot 6,78\% = 2,03 \cdot 10^4$$

$$S_r = \frac{4,40 \cdot 10^4}{2,03 \cdot 10^4} = 2,17$$

De speedup binnen deze functie is dus aanzienlijk. Deze werkwijze zal op meer plekken in de code toegepast kunnen worden. Het nadeel is wel dat de instructiecode groter wordt. Wanneer dit geen probleem is, kan er vermoedelijk nog een behoorlijke snelheidswinst behaald worden. De aangepaste functie was echter wel de functie waarbinnen het duidelijkst de imperfectie van de compiler te herkennen is. Waarschijnlijk was ook juist op deze functies veel snelheidswinst te behalen.

6.2.2 Procedure inlining

Bij procedure inlining wordt een aanroep naar een functie vervangen door de inhoud van de functie zelf. Een simpel voorbeeld hiervan is de volgende code:

```
int sample(a,b) {return a + b;}     // Functie wordt gedefinieerd
z = sample(x,y);                  // Functie wordt aangeroepen
```

Dit wordt vervangen door deze code:

```
z = a + b;                          // Code wordt direct uitgevoerd
```

Wanneer de originele wordt uitgevoerd, moet in de pipeline bij het aanroepen van de functie eerst een *instruction fetch* worden uitgevoerd. Na de fetch moet er nog een instructie worden ge*flush*t. De call wordt pas in de decode stage bekend.

```
z = sample(x,y) | FE | DE | E0 | E1 | WB |            // Call wordt aangeroepen
                      |NOP |NOP |NOP |NOP |NOP |       // Flush, adres voor call
                                                        nog niet bekend
z = a + b                  | FE | DE | E0 | E1 | WB |  // Functie wordt uitgevoerd
```

Iedere keer dat de functie wordt aangeroepen, kost dit dus twee cycli. Ook bij het terugkeren uit de functie worden cycli verbruikt. Door de aanroep te vervangen door de inhoud van de functie worden deze cycli gespaard. Wanneer een functie vaak vanuit deze plek wordt aangeroepen kan dit een significante versnelling opleveren. Er kleeft uiteraard ook een nadeel aan deze methode. Wanneer een functie van verschillende plekken aangeroepen wordt, moet de code ook op deze verschillende plekken in het instructiegeheugen worden gezet. Wanneer het om een grote functie gaat, of als de functie van veel verschillende plekken aangeroepen wordt, gaat dit veel geheugenruimte kosten.

Compiler-instellingen

De compiler past niet standaard procedure inlining toe. De compiler heeft een aantal mogelijkheden voor inlinen. De simpelste is *auto inline.* Hierbij hoeft alleen de flag *-autoinline* aan de compiler te worden meegegeven. De compiler zal zelf naar eigen inzicht functies inlinen. Het criterium dat gebruikt wordt om te beslissen of een functie inlined wordt, kan niet aangepast worden. Deze optie is getest, maar leek geen verandering te brengen.

Een andere optie is geschikt om als programmeur functies te kiezen die inlined moeten worden. Hiervoor wordt de flag *-c99inline* gebruikt. Door voor de declaratie van de functie `inline` neer te zetten wordt de inhoud van de functie ingevuld op de plaatsen waar deze wordt aangeroepen.

Keuze voor functies

Bij de keuze voor het inlinen van een functie moet op drie dingen gelet worden:

- De functie moet vaak worden aangeroepen. Inlinen geeft per call vier cycli winst, twee bij het aanroepen en twee wanneer de return wordt uitgevoerd. Als de functie vaak wordt aangeroepen, kan dit behoorlijk oplopen. Het is echter zinloos om te doen voor een functie die slechts een paar keer aangeroepen wordt.
- De functie moet op niet te veel verschillende plekken worden aangeroepen. Wanneer de functie op veel verschillende plekken wordt aangeroepen moet de code, die in de originele code slechts één keer aanwezig was, op veel verschillende plekken in de code worden neergezet. Dit heeft een negatief effect op de ruimte die in het instructiegeheugen nodig is.

- De fuctie mag niet te groot zijn. Wanneer de functie erg groot is en op meer dan één plek in de code staat, moet een grote hoeveelheid code meerdere keren in het instructiegeheugen worden geplaatst. Hierbij kan de benodigde hoeveelheid instructiegeheugen snel groeien.

Aan de hand van deze eisen zijn de meest geschikte functies voor inlining gekozen, om het resultaat van inlining te meten. Er is eerst gezocht naar functies die vaak worden aangeroepen. Hiervoor is het eerder beschreven profiel gebruikt (figuur 4.3). De functie *get_one_bit* bleek hierbij de meest voor de hand liggende kandidaat. Deze functie wordt het vaakst en slechts vanaf één plek aangeroepen. Verder zijn ook de functies *reformat*, *get_symbol* en *get_bits* interessant voor inlinen. Deze functies werden beiden vanuit twee plekken, relatief vaak, aangeroepen.

Door voor de functie zelf `inline` te schrijven zal de compiler de functie inlinen. Hierbij is wel een voorwaarde dat de functie in hetzelfde bestand staat als waarin hij aangeroepen wordt, want om de functie in te linen moet de broncode beschikbaar zijn tijdens het compileren. De functies die inlined zijn, werden allemaal slechts vanuit één bestand aangeroepen en konden dus simpelweg verplaatst worden. Een andere oplossing zou kunnen zijn om alle bestanden tezamen te compileren, in plaats van los te compileren en vervolgens te linken.

Resultaten inlining

Het inlinen heeft een aantal resultaten opgeleverd. De eerste test was een simulatie met de *-autoinline*-flag aan. Dit leverde geen versnelling op. De oorzaak hiervoor is dat de meeste functies in deze code nog werden aangeroepen in een ander bestand dan ze gedefinieerd zijn. Hierdoor is het dus niet mogelijk ze daadwerkelijk in te linen. Bij controle van het profiel van de nieuwe software bleek inderdaad dat de functies nog als functie werden aangeroepen, en dus niet inlined waren.

De *autoinline* alleen aanzetten was dus niet voldoende. Omdat de functies die interessant waren toch verplaatst moesten worden, is ervoor gekozen om *-c99inline* te gebruiken. Zo was het direct mogelijk om volledige controle over het inlinen te behouden. Het was immers mogelijk dat er een functie was die wel inlined werd door *autoinline*, terwijl dit een negatief effect had op de executietijd.

De functie *-c99inline* leverde meer resultaten op. Er is begonnen met de functie *get_one_bit*. Hier is een versnelling van 6,4% mee behaald. De andere twee voorgestelde functies zijn vervolgens ook inlined. Hierdoor is de totale versnelling veroorzaakt door inlinen opgelopen tot 11,9%. De resultaten zijn weergegeven in tabel 6.3.

Naast deze functie van de compiler, is ook handmatig inlinen getest. Er is geen significant verschil gemeten tussen automatisch en handmatig inlinen. Dit is verder dus niet gebruikt.

Tabel 6.3: Speedup totale applicatie door inlinen

	Eén issue-slot	Vier issue-slots
# Cycli voor inlinen	$4,80 \cdot 10^5$	$3,00 \cdot 10^5$
# Cycli na inlinen	$4,61 \cdot 10^5$	$2,68 \cdot 10^5$
Relatieve speedup	1,041	1,119

6.2.3 Conclusie optimalisaties

De vraag of de applicatie te versnellen is door deze specifiek voor de ρ-VEX aan te passen, is aan de hand van de vorige paragrafen goed te beantwoorden. In figuur 6.3 is de executietijd per optimalisatie weergegeven.

Uiteraard moet er uit worden gegaan van zo goed mogelijke compiler-instellingen. Dit was de eerste stap van het optimaliseren en leverde behoorlijke winst op (12%). Hierbij waren de instellingen van de compiler geoptimaliseerd wat betreft loop unrolling.

Maar ook wanneer de optimale instellingen voor de compiler worden gebruikt is er nog een behoorlijke winst te behalen door middel van handmatige loop unrolling, zonder al te veel aanpassingen aan de code en zonder dat er erg diep gegraven hoeft te worden in de assembly. De compiler levert verre van optimale code voor de ρ-VEX. Door loop unrolling zelf toe te passen, in plaats van de compiler dit te laten doen, is één functie een factor 2,17 versneld. Dit was een functie waar in de assambly-code duidelijk zichtbaar was dat de compiler loop unrolling niet optimaal toepast. Er kan worden aangenomen dat er meer speedup bereikt kan worden als ook andere functies unrolled worden. De speedup is gedeeltelijk toe te schrijven aan het data-level parallellisme dat beter wordt uitgebuit. Een ander gedeelte is toe te schrijven aan een vermindering in het aantal branch-instructies. Deze laatste versnelling is uiteraard ook te behalen op een gewone RISC-processor.

Hiernaast heeft ook procedure inlining een behoorlijke versnelling veroorzaakt. Deze optimalisatietechniek doet de VEX-compiler wel goed, mits de functies bij elkaar in een bestand staan. Ook bij deze optimalisatietechniek is een behoorlijk deel toe te schrijven aan extra parallellisatie. In figuur 6.2 is het aantal instructies per cyclus weergegeven. Hieruit blijkt dat de optimalisaties niet alleen het aantal benodigde instructies verkleinen, maar ook extra parallellisme toevoegen. De optimalisaties zouden dus juist bij de gebruikte ρ-VEX veel versnelling opleveren. Dit blijkt ook wanneer de totale relatieve speedup van de gebruikte ρ-VEX vergeleken wordt met de ρ-VEX met één issue-slot, uitgegaan van optimale compiler-instelling voor loop-unrolling. Deze vergelijking is te vinden in tabel 6.4.

Tabel 6.4: Totale speedup

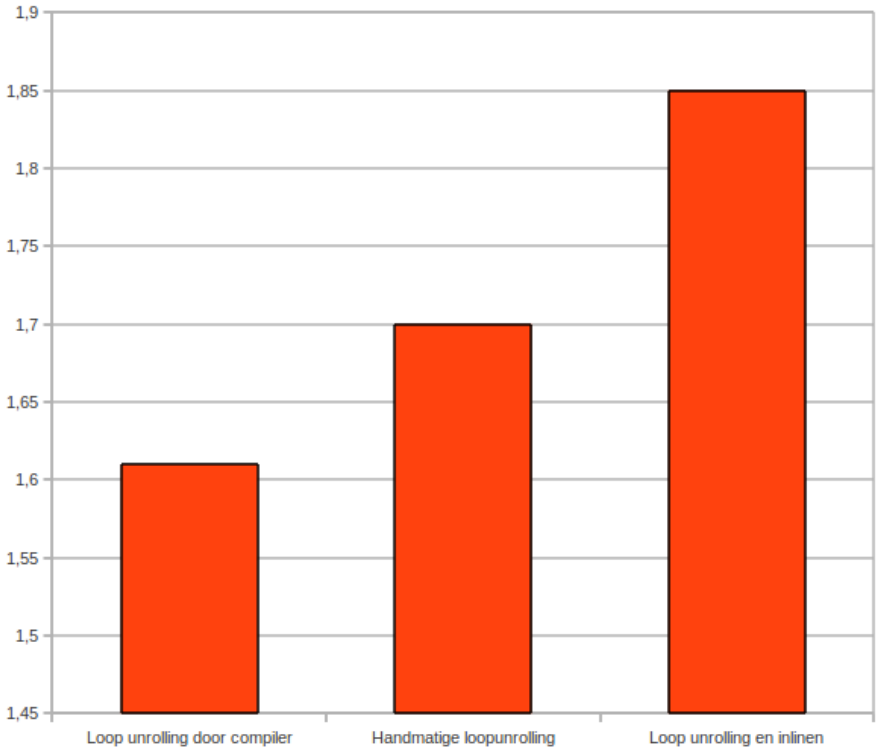
	één issue-slot	vier issue-slots
# Cycli voor optimalisatie	$4,95 \cdot 10^5$	$3,21 \cdot 10^5$
# Cycli na optimalisatie	$4,61 \cdot 10^5$	$2,68 \cdot 10^5$
Totale relatieve speedup	1,074	1,198

Belangrijk om te bedenken bij deze resultaten is dat dit niet de best mogelijke resultaten zijn. De software kan nog verder geoptimaliseerd worden. Er is slechts één functie per techniek onder handen genomen. Er kan vanuit worden gegaan dat verder optimaliseren vergelijkbaar positief effect zal hebben.

6.3 Resultaten acceleratie

De laatste stap in dit onderzoek is het vergelijken van de prestaties van het platform, met en zonder ρ-VEX als accelerator. Hiervoor wordt op beide platformen de applicatie uitgevoerd, waarbij de executietijd wordt gemeten. Op beide platforms wordt de geoptimaliseerde versie van de applicatie gebruikt. Omdat beide processoren op 100MHz werken, kan de uitvoertijd in klokcycli worden gemeten.

Omdat door de fout in de ρ-VEX de applicatie niet werkte op het platform met accelerator, is geprobeerd via indirecte methodes resultaten te verkrijgen. De tijd die het platform met enkel een MicroBlaze nodig heeft, is wel te meten. Om de executietijd bij gebruik van accelerator te bepalen, is gemeten hoeveel tijd de VEX-simulatie gebruikt, en hoeveel tijd de communicatie tussen MicroBlaze en ρ-VEX kost. Omdat de applicatie in VEX-simulatie, ingesteld met de configuratie van de ρ-VEX, in principe evenveel cycli moet gebruiken als op ρ-VEX zelf, is deze berekening een adequate oplossing om een direct meetresultaat te vervangen.



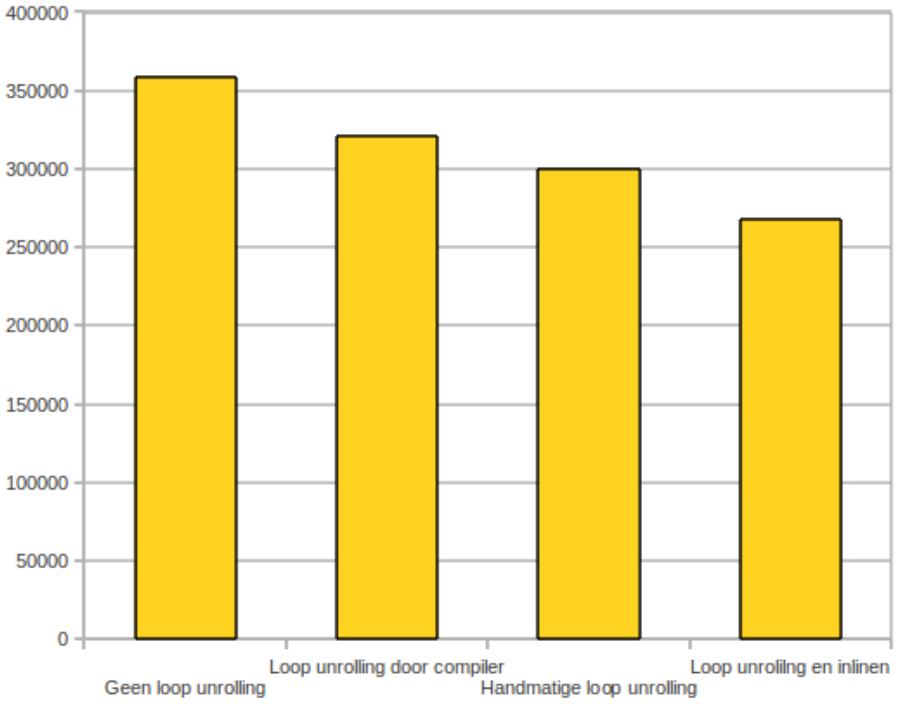
Figuur 6.2: Gemiddeld aantal instructies per cyclus bij uitvoeren van de applicatie

Voor het meten van de tijd op enkel de MicroBlaze, is de methode zoals beschreven in 5.4.3 gebruikt. De meting bevat precies het aanroepen van *JpegToBmp*:

```
StartTimer();
JpegToBmp();
StopTimer();
ReadTime();
```

Bij de eerste meting was het resultaat $1,35 \cdot 10^7$ klokcycli. Deze executietijd was een orde grootte hoger dan verwacht. De oorzaak van deze traagheid is gezocht in de traagheid van de geheugentoegang van de MicroBlaze. Voor zowel instructiegeheugen als datageheugen worden de geheugenchips naast de FPGA gebruikt. Ter vergelijking is het instructiegeheugen van de applicatie in het interne geheugen van de FPGA gezet. Gekozen is om het instructiegeheugen hier te zetten, aangezien deze vaker wordt gelezen dan het datageheugen, en tezamen pasten ze niet. Bij het uitvoeren van de applicatie is vervolgens een tijd van $2,42 \cdot 10^6$ cycli gemeten, ruim een factor vijf sneller dus. De toegangstijd van het instructiegeheugen bleek in dit systeem een groot knelpunt te zijn. Vermoedelijk is het gebruik van het datageheugen ook een knelpunt. Ter vergelijking zou een platform met cache-geheugen kunnen worden gegenereerd, hier is in dit onderzoek niet aan toegekomen. Het resultaat van $2,42 \cdot 10^6$ cycli zal verder gebruikt worden.

De tijd die de VEX-simulator nodig heeft voor het uitvoeren van de applicatie is in de vorige paragraaf on-



Figuur 6.3: Resultaten software-optimalisatie, executietijd in klokcycli

derzocht. Het resultaat uit tabel 6.4 wordt gebruikt, $2,68 \cdot 10^5$ cycli. Om te weten hoe lang de communicatie duurt is de tijd gemeten die het programmeren van het instructiegeheugen kost, de tijd die het programmeren van het datageheugen kost, en de tijd die het uitlezen van het plaatje uit het datageheugen kost. De volgende code toont de meting van het schrijven van het datageheugen:

```
StartTimer();
for (i=0; i < data_count; i++) {
        rvex_write_mem( data_mem_addr, i, data[i];
}
StopTimer();
ReadTime();
```

De code van het schrijven van het instructiegeheugen, en het lezen van het datageheugen zijn vergelijkbaar. De gemeten tijden zijn weergegeven in tabel 6.5. Hier is goed zichtbaar dat het lezen van de data meer tijd kost dan het schrijven, een logisch gevolg van het decomprimeren.

Om de totale acceleratie te berekenen is vergelijking 3.1 gebruikt. Bij het decoderen van één afbeelding volgt hieruit een speedup van:

$$\begin{aligned} S_a &= t_{CPU} - \left(t_{in,totaal} + t_x + t_{out}\right) = 2,42 \cdot 10^6 - \left(\left(1,04 \cdot 10^5 + 6,90 \cdot 10^3\right) + 2,68 \cdot 10^5 + 8,87 \cdot 10^3\right) \quad (6.1) \\ &= 2,03 \cdot 10^6 \text{ cycli} \end{aligned}$$

Als meerdere afbeeldingen worden gedecodeerd, hoeft het instructiegeheugen slechts één keer geladen

Tabel 6.5: Gemeten communicatietijden tussen MicroBlaze en ρ-VEX

	aantal cycli
schrijven instructiegeheugen	$1,04 \cdot 10^5$
schrijven datageheugen	$6,90 \cdot 10^3$
uitlezen bmp_file uit datageheugen	$8,87 \cdot 10^3$

te worden, waardoor de speedup groter wordt. De speedup bij decoderen van n afbeeldingen is:

$$\begin{aligned} S_a &= n\left(t_{CPU} - \left(t_{in,data} + t_x + t_{out}\right)\right) - t_{in,instructies} \\ &= n\left(2,42 \cdot 10^6 - \left(6,90 \cdot 10^3 + 2,68 \cdot 10^5 + 8,87 \cdot 10^3\right)\right) - 1,04 \cdot 10^5 \\ &= n \cdot 2,14 \cdot 10^6 - 1,04 \cdot 10^5 \text{ cycli} \end{aligned} \tag{6.2}$$

Bij het decoderen van grotere afbeeldingen zal de invloed van het schrijven van de instructies minder significant zijn. De versnelling bij het decoderen van meerdere afbeeldingen, zal dan niet veel afwijken van de versnelling bij het decoderen van één afbeelding. Tot slot wordt een relatieve speedup berekend, bij het decoderen van één afbeelding:

$$S_r = \frac{t_{oud}}{t_{nieuw}} = \frac{t_{CPU}}{t_{in,totaal} + t_x + t_{out}} = \frac{2,42 \cdot 10^6}{\left(1,04 \cdot 10^5 + 6,90 \cdot 10^3\right) + 2,68 \cdot 10^5 + 8,87 \cdot 10^3} = 6,24 \tag{6.3}$$

Hoofdstuk 7

Conclusie

De hoofdvraag van dit onderzoek was hoe en hoeveel versnelling te behalen valt bij gebruik van de ρ-VEX als accelerator. Dit onderzoek is gedaan als onderdeel van de realisatie van een demonstrator voor de ρ-VEX. De applicatie die in dit onderzoek is versneld is een JPEG-decoder. Voor een demonstrator zou het interessant zijn om meerdere applicaties te gebruiken. Beeldbewerkings-applicaties zoals kleurenfilters en gammacorrectie zullen waarschijnlijk beter parallelliseerbaar zijn dan de JPEG-decoder, en dus beter de kracht van de ρ-VEX demonstreren.

Om de decoder-applicatie te versnellen is eerst de structuur geanalyseerd, en zijn de libc-aanroepen eruit gestript. Na deze aanpassingen was de applicatie geschikt om op een embedded platform te draaien. Twee platformen zijn gegenereerd, één met enkel een MicroBlaze, en één met een MicroBlaze en een ρ-VEX. De applicatie werkte goed op de MicroBlaze, maar de ρ-VEX bleek nog wat verdere ontwikkeling nodig te hebben om foutloos te kunnen werken. Gelukkig was het met de VEX-simulator mogelijk om toch resultaten te verkrijgen.

Resultaten

Bij het simuleren van de code op verschillende VEX-configuraties is gebleken dat er een winst te behalen valt door het gebruik van de VLIW-eigenschap. Er bleek echter ook duidelijk een limiet te zitten aan de haalbare versnelling. Bij meer dan vier issue slots is de extra rekenkracht nauwelijks meer te benutten, doordat de beperkte ILP van het programma een limiet veroorzaakt, zoals zichtbaar in figuur 5.4. Binnen de applicatie bleken verschillende functies ook verschillend te profiteren van de grotere issue width, zoals te zien in de profielen in figuur 4.3.

Bij het partitioneren van de applicatie is gekozen om het hele decodeerproces op de ρ-VEX te accelereren. De reden hiervoor was dat de benodigde communicatie niet minder zou worden als een kleiner deel van de applicatie zou worden versneld. Een belangrijke factor die deze keuze mogelijk maakte, was dat de ρ-VEX groot genoeg was om dit alles er in te zetten. Achteraf gezien had misschien beter een limiet van 512 instructies kunnen worden aangehouden, zodat de ρ-VEX niet zou hoeven worden aangepast, wat een hoop fouten zou kunnen hebben gescheeld. Van tevoren was echter nog niet bekend dat het vergroten van het geheugen problemen zou opleveren. Ook is natuurlijk niet zeker of een kleiner deel van de applicatie wel foutloos zou werken op de ρ-VEX.

Bij het optimaliseren van de code is gebleken dat aardige winst te behalen valt door loop unrolling en procedure inlining toe te passen. Er is gebleken dat de VEX-compiler nog veel beter zou kunnen optimaliseren. Door handmatig de broncode aan te passen valt een programma toch goed te optimaliseren. De

optimalisatie geeft niet alleen een algemene versnelling, maar verhoogt ook de ILP zodat de applicatie op een VLIW er extra van profiteert. Bij het uitvoeren op de MicroBlaze processor is 7% winst behaald met de optimalisaties, tegenover 20% op de VEX.

Om te berekenen hoeveel versnelling de ρ-VEX oplevert is gemeten hoeveel cycli worden gebruikt om de instructies en data naar de ρ-VEX te schrijven, en om de afbeelding weer uit te lezen. Deze cycli worden opgeteld bij de cycli die de VEX-simulator nodig heeft voor het uitvoeren van de applicatie, wat gelijk hoort te zijn aan hoe lang de ρ-VEX hiermee bezig zou zijn. Deze som wordt vergeleken met het aantal cycli dat de MicroBlaze nodig heeft voor het uitvoeren van de applicatie. De totale versnelling die hieruit volgt is 6,24, bij het decoderen van één afbeelding. Hieruit valt te concluderen dat de ρ-VEX als accelerator een nuttige toevoeging is aan het platform.

Aanbevelingen

Het is nog te betwisten of de vergelijking tussen de twee platformen eerlijk is, aangezien de MicroBlaze een veel trager geheugen gebruikt dan de ρ-VEX. Om deze factor te verkleinen is het instructiegeheugen van de MicroBlaze in het interne geheugen van de FPGA gezet, zodat alleen bij het datageheugen vertragingen kunnen ontstaan. Het gebruik van caching van het datageheugen is in dit onderzoek niet geprobeerd, mogelijk is het uitvoeren op de MicroBlaze hier een stuk sneller mee te maken. Als de wens is om puur de architecturen van de processoren te vergelijken, dan zouden ze in dezelfde situatie moeten worden gezet, dus met even snel geheugen. Het doel van dit onderzoek was echter niet om de processoren te vergelijken, maar om de invloed van een accelerator in het platform te bekijken. Dat de accelerator in dit onderzoek veel sneller is dan de CPU, geeft aan dat het interessant is om voor intensieve berekeningen een extra processor te gebruiken, die een met klein en snel geheugen verbonden is.

In dit onderzoek is nog niet gekeken naar de mogelijkheid om de twee processoren tegelijkertijd berekeningen uit te laten voeren. Door terwijl de accelerator bezig is ook de CPU te benutten zou nog meer parallellisme kunnen worden bereikt.

Om van het in dit onderzoek gemaakte systeem een interessante demonstrator te maken, moet het natuurlijk ook met grotere afbeeldingen werken. Dit moet gewoon kunnen, zolang de ρ-VEX genoeg geheugenruimte heeft. Bij een grotere afbeelding is natuurlijk meer tijd nodig om de data naar de accelerator te sturen, maar zal ook de tijd die wordt gewonnen evenveel groter zijn.

Uit dit onderzoek zijn twee belangrijke conclusies getrokken. De eerste is dat de JPEG-decoder applicatie ongeveer 6 keer sneller uitgevoerd kan worden wanneer de ρ-VEX als accelerator wordt gebruikt. De tweede conclusie is dat de compiler een sleutelrol speelt bij het benutten van een VLIW-processor. Waar de compiler inefficiënt optimaliseert, kan de programmeur de code aanpassen om het parallellisme van de ρ-VEX maximaal te benutten.

Bibliografie

[1] R. Seedorf, F. Anjam, A. Brandon, and S. Wong, "ρ-vex: A parametrized vliw processor," 2011.

[2] R. Seedorf, "Fingerprint verification on the vex processor," Master's thesis, 2010.

[3] P. Tröger, "The multi-core era - trends and challenges," *CoRR*, vol. 0810.5439, 2008.

[4] S. Borkar and A. A. Chien, "The future of microprocessors," *ACM transactions on embedded computing systems*, vol. 54, 2011.

[5] Patterson and Hennessy, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2009.

[6] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *IEEE Computer*, vol. 38, 2005.

[7] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, pp. 10–24, 2006.

[8] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," *Proceedings of the IEEE*, vol. 83, 1995.

[9] F. Anjam, S. Wong, and F. Nadeem, "A shared reconfigurable vliw multiprocessor system," *Parallel and Distributed Processing*, 2010.

[10] W. Wolf, "A decade of hardware-software codesign," *Computer*, vol. 36, 2003.

[11] A. Kalavade, "A hardware-software codesign methodology for dsp applications," *Design & Test of Computers, IEEE*, vol. 10, 1993.

[12] W. Wolf, *Computers as Components*. Morgan Kaufmann, 2001.

[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantative Approach*. Morgan Kaufmann, 2003.

[14] S. L. Graham, "gprof: a call graph execution profiler," 1982.

[15] Wikipedia, "Jpeg — Wikipedia, the free encyclopedia," 2011.

[16] "Jpeg decoder case study," tech. rep., Critical Blue, 2007.

[17] M. Andersson and P. Karlström, "Parallel jpeg processing with a hardware accelerated dsp processor," Master's thesis, Linköpings Universitet, 2004.

[18] "Independent jpeg group."

[19] HP, *The VEX System*, 2003.

Appendix A

Code File I/O

Deze appendix bevat de C-code die geschreven is om het lezen en schrijven van bestanden zelf te implementeren. De volgende vier functies zijn geïmplementeerd:

- fgetc2
- fseek2
- ftell2
- putc2

Om te zorgen dat deze functies worden aangeroepen zijn de volgende pre-processor definities gebruikt. Deze definities zijn ingevuld in *jpeg.h*, welke in elk C-bestand wordt opgenomen. Ook staan hier de definities om de printfuncties te weg te gooien.

```
1  #include "jpegfile.h"
2
3  // Use our own filesystem implementation
4  #define fgetc fgetc2
5  #define fseek fseek2
6  #define ftell ftell2
7  #undef putc
8  #define putc putc2
9
10 #ifdef RUN_ON_VEX
11       // Use no printf
12       #define printf(arg...) 0
13       #define fprintf(arg...) 0
14 #endif
15
16 #ifdef RUN_ON_MICROBLAZE
17       // Use xilinx' simple printf
18       #define fprintf(f, arg...) xil_printf(arg)
19       #define printf(arg...) xil_printf(arg)
20 #endif
```

fileio.c

```
1  /* Simple implementation of file I/O, using static arrays.
2   * Reading is always done from jpeg_file, writing is done to bmp_file.
3   */
4  #include "fileio.h"
5
6  extern unsigned char jpeg_file[];
7  extern unsigned char bmp_file[];
8  extern unsigned int jpeg_file_size;
9  extern unsigned int bmp_file_size;
10
11 long jpeg_pointer = 0;
12 long bmp_pointer = 0;
13
14 /**** Functions for jpeg_file ****/
15
16 int fgetc2(FILE *fp) {
17       if (jpeg_pointer >= jpeg_file_size) {
18             return EOF;
19       }
20       return (int) jpeg_file[jpeg_pointer++];
21 }
22
23 int fseek2(FILE * stream, long int offset, int origin ) {
24       if (origin == SEEK_CUR) {
25             jpeg_pointer += offset;
26       }
27       else if (origin == SEEK_SET) {
28             jpeg_pointer = offset;
29       }
30       else if (origin == SEEK_END) {
31             jpeg_pointer = jpeg_file_size+offset;
32       }
33       else {
34             return -1;
35       }
36       return 0;
37 }
38
39 long ftell2( FILE * stream ) {
40       return jpeg_pointer;
41 }
42
43 /**** Functions for bmp_file ****/
44
45 int putc2(int c, FILE *fp) {
46       if (bmp_pointer >= bmp_file_size) {
47             return EOF;
48       }
```

```
49          bmp_file[bmp_pointer++] = (unsigned char) c;
50
51          return c;
52  }
```

Appendix B

Code software optimalisatie

Deze appendix bevat de codes die gebruikt, geschreven en gegenereerd zijn voor de software-optimalisatie door middel van loop unrolling (paragraaf 6.2). De verwerking van rijen worden gebruikt in het besproken voorbeeld. De andere aanpassing is gedaan binnen de loop voor de verwerking van kolommen. De codes komen uit *fast_int_idct.c* en *fast_int_idct.s*.

B.1 Originele C-code voor verwerking van rijen

```
1   /* Pass 1: process rows. */
2   for (k = 0; k < 8; k++) {
3     /* Prescale k-th row: */
4     for (l = 0; l < 8; l++){
5       Y(k,l) = SCALE(input->block[k][l], S_BITS);
6     }
7     /* 1-D IDCT on k-th row: */
8     idct_1d(&Y(k,0));
9     /* Result Y is scaled up by factor sqrt(8)*2^S_BITS. */
10  }
```

B.2 Aangepaste C-code voor verwerking van rijen

```
1   /* Pass 1: process rows. */
2   for (k = 0; k < 8; k++) {
3       int m1,m2,m3,m4,m5,m6,m7,m8;
4       m1 = input->block[k][0];
5       m2 = input->block[k][1];
6       m3 = input->block[k][2];
7       m4 = input->block[k][3];
8       m5 = input->block[k][4];
9       m6 = input->block[k][5];
10      m7 = input->block[k][6];
11      m8 = input->block[k][7];
12
13      Y(k,0) = SCALE(m1, S_BITS);
14      Y(k,1) = SCALE(m2, S_BITS);
15      Y(k,2) = SCALE(m3, S_BITS);
16      Y(k,3) = SCALE(m4, S_BITS);
17      Y(k,4) = SCALE(m5, S_BITS);
18      Y(k,5) = SCALE(m6, S_BITS);
19      Y(k,6) = SCALE(m7, S_BITS);
20      Y(k,7) = SCALE(m8, S_BITS);
21    /* 1-D IDCT on k-th row: */
22    idct_1d(&Y(k,0));
23    /* Result Y is scaled up by factor sqrt(8)*2^S_BITS. */
24  }
```

B.3 Originele Assemly-code voor verwerken van rijen

```
1  .trace 1
2  L0?3:
3        c0    cmplt $b0.0 = $r0.58, $r0.0
4        c0    add $r0.4 = $r0.60, 1
5        c0    ldw.d $r0.2 = 0[$r0.57]
6        c0    add $r0.5 = $r0.60, 2
7  ;;                                                  ## 0
8        c0    sh2add $r0.6 = $r0.4, $r0.61
9        c0    sh2add $r0.4 = $r0.4, $r0.62
10       c0    sh2add $r0.7 = $r0.5, $r0.61
11       c0    sh2add $r0.5 = $r0.5, $r0.62
12 ;;                                                  ## 1
13       c0    shl $r0.2 = $r0.2, 3
14       c0    add $r0.8 = $r0.60, 3
15       c0    add $r0.9 = $r0.60, 4
16       c0    brf $b0.0, L1?3
17 ;;                                                  ## 2
18       c0    sh2add $r0.10 = $r0.8, $r0.61
19       c0    sh2add $r0.8 = $r0.8, $r0.62
20       c0    sh2add $r0.11 = $r0.9, $r0.61
21       c0    sh2add $r0.9 = $r0.9, $r0.62
22 ;;                                                  ## 3
23       c0    add $r0.12 = $r0.60, 5
24       c0    add $r0.13 = $r0.60, 6
25       c0    add $r0.14 = $r0.60, 7
26       c0    mov $r0.3 = $r0.59
27 ;;                                                  ## 4
28       c0    sh2add $r0.15 = $r0.12, $r0.61
29       c0    sh2add $r0.12 = $r0.12, $r0.62
30       c0    sh2add $r0.16 = $r0.13, $r0.61
31       c0    sh2add $r0.13 = $r0.13, $r0.62
32 ;;                                                  ## 5
33       c0    stw 0[$r0.59] = $r0.2
34       c0    sh2add $r0.17 = $r0.14, $r0.61
35       c0    sh2add $r0.14 = $r0.14, $r0.62
36 ;;                                                  ## 6
37       c0    ldw $r0.6 = 0[$r0.6]
38 ;;                                                  ## 7
39 ;;                                                  ## 8
40       c0    shl $r0.6 = $r0.6, 3
41 ;;                                                  ## 9
42       c0    stw 0[$r0.4] = $r0.6
43 ;;                                                  ## 10
44       c0    ldw $r0.7 = 0[$r0.7]
45 ;;                                                  ## 11
46 ;;                                                  ## 12
47       c0    shl $r0.7 = $r0.7, 3
48 ;;                                                  ## 13
```

```
49        c0    stw 0[$r0.5] = $r0.7
50  ;;                                                  ## 14
51        c0    ldw $r0.10 = 0[$r0.10]
52  ;;                                                  ## 15
53  ;;                                                  ## 16
54        c0    shl $r0.10 = $r0.10, 3
55  ;;                                                  ## 17
56        c0    stw 0[$r0.8] = $r0.10
57  ;;                                                  ## 18
58        c0    ldw $r0.11 = 0[$r0.11]
59  ;;                                                  ## 19
60  ;;                                                  ## 20
61        c0    shl $r0.11 = $r0.11, 3
62  ;;                                                  ## 21
63        c0    stw 0[$r0.9] = $r0.11
64  ;;                                                  ## 22
65        c0    ldw $r0.15 = 0[$r0.15]
66  ;;                                                  ## 23
67  ;;                                                  ## 24
68        c0    shl $r0.15 = $r0.15, 3
69  ;;                                                  ## 25
70        c0    stw 0[$r0.12] = $r0.15
71  ;;                                                  ## 26
72        c0    ldw $r0.16 = 0[$r0.16]
73  ;;                                                  ## 27
74  ;;                                                  ## 28
75        c0    shl $r0.16 = $r0.16, 3
76  ;;                                                  ## 29
77        c0    stw 0[$r0.13] = $r0.16
78  ;;                                                  ## 30
79        c0    ldw $r0.17 = 0[$r0.17]
80  ;;                                                  ## 31
81  ;;                                                  ## 32
82        c0    shl $r0.17 = $r0.17, 3
83  ;;                                                  ## 33
84  .call idct_1d, caller, arg($r0.3:u32), ret()
```

B.4 Assembly-code voor verwerking van rijen na aanpassing

```
1  .trace 1
2  L0?3:
3        c0    cmplt $b0.0 = $r0.58, $r0.0
4        c0    ldw.d $r0.2 = 0[$r0.57]
5        c0    mov $r0.3 = $r0.59
6  ;;                                                  ## 0
7        c0    ldw.d $r0.4 = 4[$r0.57]
8  ;;                                                  ## 1
9        c0    ldw.d $r0.5 = 8[$r0.57]
10       c0    shl $r0.2 = $r0.2, 3
11       c0    brf $b0.0, L1?3
12 ;;                                                  ## 2
13       c0    ldw $r0.6 = 12[$r0.57]
14       c0    shl $r0.4 = $r0.4, 3
15 ;;                                                  ## 3
16       c0    shl $r0.5 = $r0.5, 3
17       c0    ldw $r0.7 = 16[$r0.57]
18 ;;                                                  ## 4
19       c0    shl $r0.6 = $r0.6, 3
20       c0    ldw $r0.8 = 20[$r0.57]
21 ;;                                                  ## 5
22       c0    shl $r0.7 = $r0.7, 3
23       c0    ldw $r0.9 = 24[$r0.57]
24 ;;                                                  ## 6
25       c0    shl $r0.8 = $r0.8, 3
26       c0    ldw $r0.10 = 28[$r0.57]
27 ;;                                                  ## 7
28       c0    shl $r0.9 = $r0.9, 3
29       c0    stw 0[$r0.59] = $r0.2
30 ;;                                                  ## 8
31       c0    shl $r0.10 = $r0.10, 3
32       c0    stw 4[$r0.59] = $r0.4
33 ;;                                                  ## 9
34       c0    stw 8[$r0.59] = $r0.5
35 ;;                                                  ## 10
36       c0    stw 12[$r0.59] = $r0.6
37 ;;                                                  ## 11
38       c0    stw 16[$r0.59] = $r0.7
39 ;;                                                  ## 12
40       c0    stw 20[$r0.59] = $r0.8
41 ;;                                                  ## 13
42       c0    stw 24[$r0.59] = $r0.9
43 ;;                                                  ## 14
44 .call idct_1d, caller, arg($r0.3:u32), ret()
```

B.5 Originele C-code voor verwerking van kolommen

```
1   /* Pass 2: process columns. */
2   for (l = 0; l < 8; l++) {
3     int Yc[8];
4
5     for (k = 0; k < 8; k++) Yc[k] = Y(k,l);
6     /* 1-D IDCT on l-th column: */
7     idct_1d(Yc);
8     /* Result is once more scaled up by a factor sqrt(8). */
9     for (k = 0; k < 8; k++) {
10      int r = 128 + DESCALE(Yc[k], S_BITS+3); /* includes level shift */
11
12      /* Clip to 8 bits unsigned: */
13      r = r > 0 ? (r < 255 ? r : 255) : 0;
14      X(k,l) = r;
15    }
16  }
```

B.6 Aangepaste C-code voor verwerking van kolommen

```
1   /* Pass 2: process columns. */
2   for (l = 0; l < 8; l++) {
3     int Yc[8];
4     int r1,r2,r3,r4,r5,r6,r7,r8;
5
6     for (k = 0; k < 8; k++) Yc[k] = Y(k,l);
7     /* 1-D IDCT on l-th column: */
8     idct_1d(Yc);
9     /* Result is once more scaled up by a factor sqrt(8). */
10
11    r1 = 128 + DESCALE(Yc[0], S_BITS+3); /* includes level shift */
12    r2 = 128 + DESCALE(Yc[1], S_BITS+3); /* includes level shift */
13    r3 = 128 + DESCALE(Yc[2], S_BITS+3); /* includes level shift */
14    r4 = 128 + DESCALE(Yc[3], S_BITS+3); /* includes level shift */
15    r5 = 128 + DESCALE(Yc[4], S_BITS+3); /* includes level shift */
16    r6 = 128 + DESCALE(Yc[5], S_BITS+3); /* includes level shift */
17    r7 = 128 + DESCALE(Yc[6], S_BITS+3); /* includes level shift */
18    r8 = 128 + DESCALE(Yc[7], S_BITS+3); /* includes level shift */
19
20    /* Clip to 8 bits unsigned: */
21    r1 = r1 > 0 ? (r1 < 255 ? r1 : 255) : 0;
22    r2 = r2 > 0 ? (r2 < 255 ? r2 : 255) : 0;
23    r3 = r3 > 0 ? (r3 < 255 ? r3 : 255) : 0;
24    r4 = r4 > 0 ? (r4 < 255 ? r4 : 255) : 0;
25    r5 = r5 > 0 ? (r5 < 255 ? r5 : 255) : 0;
26    r6 = r6 > 0 ? (r6 < 255 ? r6 : 255) : 0;
27    r7 = r7 > 0 ? (r7 < 255 ? r7 : 255) : 0;
28    r8 = r8 > 0 ? (r8 < 255 ? r8 : 255) : 0;
29
30    X(0,l) = r1;
31    X(1,l) = r2;
32    X(2,l) = r3;
33    X(3,l) = r4;
34    X(4,l) = r5;
35    X(5,l) = r6;
36    X(6,l) = r7;
37    X(7,l) = r8;
38
39  }
```