

Prestatiemetingen voor systeemsoftware m.b.v. FPGA

Jens Van den Broeck

Promotoren: prof. dr. ir. Bjorn De Sutter, prof. dr. ir. Dirk Stroobandt
Begeleiders: ir. Niels Penneman, ir. Wim Meeus

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: elektrotechniek

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2012-2013



Prestatiemetingen voor systeemsoftware m.b.v. FPGA

Jens Van den Broeck

Masterproef ingediend tot het behalen van de academische graad van
Master in de ingenieurswetenschappen: elektrotechniek

Academiejaar 2012–2013

Universiteit Gent
Faculteit Ingenieurswetenschappen en Architectuur

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. Jan Van Campenhout

Promotoren

prof. dr. ir. Bjorn De Sutter
prof. dr. ir. Dirk Stroobandt

Begeleiders

ir. Niels Penneman
ir. Wim Meeus

Samenvatting

Bij het ontwikkelen van software gaat de ontwikkelaar op zoek naar de traagste blokken code door een profiel op te stellen van de applicatie. Toepassingen die dergelijke profielen opstellen, noemt men profilers. Het profileren van gebruikersapplicaties is relatief eenvoudig, maar software die draait op systeenniveau, bijvoorbeeld een besturingssysteem, is minder makkelijk te profileren. In deze thesis ontwikkelen we een oplossing die het mogelijk maakt systeemsoftware op een efficiënte te profileren.

Trefwoorden: FPGA, prestatiemeting, profiler, systeemsoftware

Voorwoord

Deze thesis betekent het einde van vijf onvergetelijke jaren aan de Universiteit Gent. Dit is dan ook het gepaste moment voor een dankwoord aan de mensen waaraan ik de laatste jaren veel gehad heb.

In de eerste plaats zou ik graag mijn promotoren, prof. Bjorn De Sutter en prof. Dirk Stroobandt bedanken. Zij hebben de nodige tijd en middelen vrij gemaakt zodat ik deze thesis kon uitvoeren. Ook prof. Pieter Rombouts verdient een plaatsje in dit dankwoord. Hij leende me een oscilloscoop die onmisbaar bleek voor het debuggen, en stelde zijn soldeerstation ter beschikking wanneer ik het nodig had.

Daarnaast komen uiteraard mijn begeleiders, ir. Niels Penneman en ir. Wim Meeus. Zij hebben het risico genomen met mij in zee te gaan, waar ik hen heel dankbaar voor ben. Niels heeft me met engelengeduld geleerd hoe je echt programmeert, maar wat ik vooral apprecieer, is dat hij er altijd was wanneer ik peptalk nodig had. Wim en Niels, bedankt!

Goede begeleiding is één ding, maar sfeer is op zijn minst even belangrijk. Voor dit laatste zou ik in de eerste plaats graag Tim, Ronald, Jonas, Panagiotis en Hadi bedanken. Ook Bart verdient een eervolle vermelding! Samen met Niels hebben ze me geleerd dat ik mijn laptop met een goed wachtwoord moet beveiligen.

Iemand die zeker weten ook onmisbaar was bij het uitvoeren van dit project, is ing. Marnix Vermassen. Wanneer ik elektronica-onderdelen online moest bestellen, zorgde hij ervoor dat dit zo snel mogelijk in orde kwam.

Naast al deze mensen aan de universiteit, is er natuurlijk mijn familie. In het bijzonder zou ik mijn ouders willen bedanken. Ze gaven me alle kansen en hebben me altijd ondersteund in mijn beslissingen. Papa, mama, bedankt voor alles wat jullie gedaan hebben!

Ook mijn broer en zus wil ik bij deze bedanken. De voorbije jaren waren soms heel zwaar, en ze moesten dan ook heel wat gezaag doorstaan.

Vervolgens zou ik nog graag mijn vaste groepsgenoten bedanken. Samen met hen heb ik talloze opdrachten gemaakt, samen stonden we sterk. Marijn, Sander, Tim, Jeroen en Mathias: bedankt voor de goede samenwerking!

De volgende persoon op mijn lijstje is Evy, mijn vriendin. Het afgelopen jaar had ik bijzonder weinig tijd, maar ik zal het goed maken door een paar keer lekker te koken! Ook mag mijn kotmaatje, Evelien niet ontbreken. Samen hebben we vele onvergetelijke en vroege uren doorgebracht in de Overpoort.

Toelating tot bruikleen

De auteur geeft de toelating deze masterproef voor consultatie beschikbaar te stellen en delen van de masterproef te kopiëren voor persoonlijk gebruik.

Elk ander gebruik valt onder de beperkingen van het auteursrecht, in het bijzonder met betrekking tot de verplichting de bron uitdrukkelijk te vermelden bij het aanhalen van resultaten uit deze masterproef.

Jens Van den Broeck
15 juni 2013

Measuring the performance of system software using an FPGA

Jens Van den Broeck

Supervisor(s): prof. dr. ir. Bjorn De Sutter, prof. dr. ir. Dirk Stroobandt, ir. Niels Penneman, ir. Wim Meeus

Abstract—This article presents a solution for long-term profiling of system software. This was achieved by using an FPGA to observe the behaviour of the system software in real-time. The observations are sent to a host computer where they are stored in a file. By parsing the observations correctly, the timing behaviour of the system software can be reconstructed and analysed.

Keywords—FPGA, performance, profiling, system software

I. INTRODUCTION

When a software developer wants to speed up his application, he uses a profiler application to measure which parts of the code are taking most of the execution time. Once this bottleneck has been found, the performance of the application can be increased by taking appropriate actions. A regular application can be profiled by running a profiler in parallel with the application. The profiler gathers information concerning the execution of the application by communicating with the underlying operating system. One cannot simply extend this reasoning to the profiling process of system software, e.g. an OS, because running two OS in parallel on the same hardware is way too intrusive.

II. SYSTEM SOFTWARE

The goal of this thesis is to measure the performance of system software. By using the term *system software*, we take every software package into account that runs directly on top of the hardware. Note that operating systems are not the only example of system software; hypervisors also belong in this category. A hypervisor is used to create a virtualised hardware layer on which multiple OS can run in parallel without influencing one another.

III. PROBLEMS

Commercial tools to profile system software exist. However, these tools suffer some disadvantages. Firstly, they are often very expensive. Secondly, the connection with the hardware on which the system software under test runs is often made via a proprietary, exotic connector. As such, the number of hardware solutions on which system software can be profiled is limited. And thirdly, most tools can't profile over a long time span. This means that profiling can only be done over short sections of the system software code. Profiling the Linux boot process, for example, can be problematic.

IV. GOALS

The main goal of this thesis is the development of a profiler for system software which satisfies the following conditions:

1. We want to have accurate measurements.
2. We want to be able to profile during long periods.

3. The profiling process has to be as unintrusive as possible.

V. PROPOSED SOLUTION

The solution we developed in this thesis, consists of three major components: the system under test (SUT), an observer platform, and a measurement platform.

A. The system under test

The SUT consists of a hardware platform and the system software which we want to test. The hardware platforms we used were a BeagleBoard and a PandaBoard. Both platforms use a Texas Instruments OMAP processor, which is ARM based. The system software we used as a practical test case for the proposed solution, is the hypervisor which is in development at Computer Systems Lab. However, due to the limited time of this thesis, we weren't able to extensively test the hypervisor. Instead, we wrote small test programs to simulate certain actions performed by the hypervisor.

The OMAP processors have a number of General Purpose I/O (GPIO) pins available that are free for use. The signal of some of these pins can be routed to the expansion connectors present on both boards. The fact that GPIO pins don't need a certain protocol to be controlled makes them ideally suited for our application. Only 4 to 7 instructions are needed to bring a GPIO pin in a new state. These additional instructions don't pose a threat to the timing behaviour of the software. The GPIO pins also change states almost immediately, which makes the development of an accurate system possible.

The system software can be profiled by driving a GPIO pin high when a code block is entered and driving it low when the code block is exited, for example.

B. The observer platform

A Xilinx FPGA board is used to observe the behaviour of the system software. It contains a Virtex-5 FPGA on which we implemented an event based sampler. This samples observes the GPIO pins of the hardware platform previously spoken of. In contrast to a continuous sampler, an event based sampler only generates data when the input changes. Using an event-based sampler, a minimal amount of data is generated to be able to reconstruct the timing behaviour of the system software.

The FPGA board provides two expansion connectors, one of which we use to connect to the BeagleBoard or PandaBoard. Care must be taken when connecting two electronic systems to one another. To prevent that one of the two components would break, an interface board was developed to isolate the electronics of the FPGA board and the test hardware. Level shifting functionality is provided to allow voltage translation.

Several communication interfaces are available to connect the FPGA board with the measurement platform, which is a PC in our case. The choice we made is based on two criteria: the data rate has to be high enough and the FPGA should be able to send data without an integrated software stack. Implementing a soft core processor on the FPGA requires quite a lot of area, so the latter condition is important when we want to be able to optimise the design for speed. Eventually we have chosen for PCI Express to connect the FPGA-board with the measurement platform. The data is sent to the PC using Direct Memory Access (DMA).

C. The measurement platform

A Linux PC is used as the measurement platform. As described above, a PCI Express interface is used to connect with the FPGA-board. To be able to actually communicate with the board, a device driver was developed. Also, a streaming application was written to efficiently store the generated data to a file on the hard drive of the computer. Additionally, a control application was developed to enable user control of the sampler.

VI. CONCLUSIONS

During this thesis, a solution was developed to execute long-term profiling of system software. The final solution can observe GPIO lines which keep their state for a minimum of 77 ns. This means that GPIO signals with a frequency of up to 13 MHz can be observed.

The time span over which profiling can be done is limited by the amount of free space on the hard disk of the measurement computer.

Inhoudsopgave

Voorwoord	ii
Lijst van afkortingen	ix
1 Inleiding	1
1.1 Probleemstelling	1
1.2 Doelstelling	2
1.3 Structuur van dit document	2
2 Literatuurstudie	3
2.1 Processorspecifieke registers	3
2.2 Logic analyzers	4
2.3 Gespecialiseerde traceerhardware	5
2.3.1 ARM DSTREAM en DS-5	5
2.3.2 Lauterbach TRACE32	6
2.4 Besluit	6
3 Top-level ontwerp	8
3.1 Het testplatform	8
3.1.1 Verbinding met de FPGA	10
3.1.2 Informatie uit de systeemsoftware halen	10
3.2 Het observatieplatform	10
3.2.1 De sampler	11
3.2.2 Het geheugen	12
3.2.3 Verbinding met de PC	13
3.2.4 Communicatie op de FPGA	14
3.3 Het meetplatform	14
3.3.1 Het stuurprogramma	14
3.3.2 Het meetprogramma	15
3.4 Samenvatting	15
4 Interface testplatform-FPGA	17
4.1 Elektrische compatibiliteit	17
4.2 Het interfacebord	18
4.2.1 De connectoren	18
4.2.2 De level shifter	18
4.2.3 In- en uitschakelen van de level shifter	19

4.2.4	Het interfacebord	19
4.3	GPIO-pinnen van de testbordjes	21
4.3.1	Het BeagleBoard	21
4.3.2	Het PandaBoard	24
4.4	GPIO-pinnen van het FPGA-bord	25
5	Interface FPGA-PC	27
5.1	Communicatie tussen twee PCIe-nodes	28
5.2	Communicatie tussen een PCIe-kaart en de CPU	28
5.3	Interrupts	29
5.4	Detectie en configuratie van een PCIe-kaart	29
6	FPGA-implementatie	32
6.1	Schema	32
6.2	Samplen	33
6.2.1	Top-level ontwerp	33
6.2.2	FPGA-ontwerp	34
6.3	SRAM-controller	36
6.3.1	Top-level ontwerp	36
6.3.2	FPGA-ontwerp	37
6.3.3	Implementatie van een bidirectionele bus	41
6.4	Arbiter	44
6.4.1	Top-level ontwerp	44
6.4.2	FPGA-ontwerp	45
6.4.3	Aanmaken van de FIFO's	52
6.5	Communicatie FPGA-PC m.b.v. PCIe	53
6.5.1	Instellingen van de PCIe-core	53
6.5.2	Inschakelen van de PCIe-vertaalbrug	55
6.5.3	Het instellen van een configuratieregister	56
6.6	Data overbrengen in burst m.b.v. DMA	56
6.7	Optimalisatie van het ontwerp	57
7	PC-implementatie	60
7.1	Ontwerp	60
7.2	Het stuurprogramma	61
7.2.1	Userspace en kernelspace	61
7.2.2	Interface met gebruikersapplicaties	62
7.2.3	Detectie en initialisatie van het FPGA-bord	62
7.2.4	Geheugen beschikbaar stellen voor de gebruiker	63
7.2.5	Leesoperaties afhandelen	64
7.2.6	Gebruikerscommando's afhandelen	64
7.2.7	Interrupts afhandelen	65
7.3	De controle-applicatie	65
7.4	Streamen van data naar de harde schijf	66
7.4.1	Werkingsprincipe	66
7.4.2	Alloceren van grote bestandsbuffers	67
7.4.3	Multithreading	68

7.4.4	Het streamen beëindigen	68
7.4.5	Implementatiedetails	69
7.5	Verwerken van de gestreamde samples	70
7.6	De visualisatie	70
8	Verificatie	74
8.1	Verificatie van het ontwerp	74
8.2	De traagste component van het systeem	75
8.2.1	De FPGA	75
8.2.2	De PC	76
8.2.3	Besluit	77
8.3	Gebruik van de FPGA-bronnen	77
9	De praktijk	79
9.1	De meetopstelling	79
9.1.1	Assemblage van de hardware	79
9.1.2	Opstartprocedure	81
9.2	Experiment: uitvoeringstijd van een programma	82
9.3	Experimenteren met het PandaBoard	82
10	Conclusie	84
10.1	Toekomstig werk	84
10.2	Tijdsbesteding	85
A	Datasheet van het interfacebord	87
A.1	Bedradingsschema	87
A.2	Verificatie en testen	87
A.3	Gebruik	89
B	Maximale GPIO-togglefrequentie	90
Bibliografie		91
Lijst van figuren		95
Lijst van tabellen		97

Lijst van afkortingen

BAR	Base Address Register
BCR	Bridge Control Register
BRAM	Block RAM
CE	Chip Enable
CLB	Configurebaar Logisch Blok
CSL	Computer Systems Lab
DMA	Direct Memory Access
FWFT	First Word Fall Through
GPIO	General Purpose Input/Output
LMB	Local Memory Bus
MMU	Memory Management Unit
MSI	Message Signaled Interrupt
OE	Output Enable
PAR	Place and Route
PCIe	PCI Express
PLB	Processor Local Bus
PSF	Platform Specification Format
RC	Root Complex
SOC	System on chip
SUT	System Under Test

Lijst van afkortingen

x

SWD	Serial Wire Debug
VCD	Value Change Dump
WLF	Wave Log Format
XGI	Xilinx Generic Interface
XUPV5	Xilinx University Program Virtex 5
ZBT	Zero-Bus Turnaround

Hoofdstuk 1

Inleiding

1.1 Probleemstelling

Elk modern digitaal toestel bevat een al dan niet beperkte hoeveelheid software die de verschillende onderdelen ervan aanstuurt en die interactie met de gebruiker mogelijk maakt. In het ideale geval ondervindt men geen hinder van deze *systeemsoftware*: gebruikersapplicaties interageren rechtstreeks met de gebruiker en reageren onmiddellijk. De aanwezigheid van die systeemsoftware gaat echter niet altijd ongemerkt: soms duurt het lang vooraleer het toestel is opgestart, of het duurt een zekere tijd vooraleer het systeem de actie van de gebruiker merkt. De oorzaak van deze kwaaltjes kan achterhaald worden door een analyse te doen van de software en vervolgens de prestaties ervan op te voeren door de oorzaak te elimineren.

Het analyseren en verbeteren van de prestaties van software hangt af van het niveau waarop die software draait. Hebben we te maken met een gebruikersapplicatie, dan kan een zogenaamde *profiler* informatie in verband met de uitvoering van die applicatie opvragen aan het besturingssysteem. Er bestaan tientallen softwarepakketten die applicaties kunnen profileren [22]. Enkele voorbeelden van dergelijke toepassingen zijn *gprof* en *perf tools* voor Linux, *GlowCode* en *Windows Performance Analysis Toolkit* voor Windows [19, 25, 18, 30].

Willen we daarentegen de software gaan analyseren *waarop* de applicaties draaien, dan bevinden we ons op het niveau van de *systeemsoftware*. Merk op dat we hier met de term *systeemsoftware* meer bedoelen dan alleen besturingssystemen. Een hypervisor is een voorbeeld van systeemsoftware dat geen besturingssysteem is. Met een hypervisor is het mogelijk hardware te emuleren en meerdere besturingssystemen op hetzelfde hardwareplatform te draaien zonder dat ze het van elkaar weten. Het Xen Project is hier een voorbeeld van [28].

Voor het profileren van systeemsoftware kunnen we geen gebruik maken van twee parallelle besturingssystemen omdat de elkaar teveel zouden beïnvloeden. In de plaats daarvan moeten we op hardwareniveau informatie doorgeven aan een extern platform dat deze informatie verwerkt.

Er bestaan enkele oplossingen voor het analyseren en profileren van systeemsoftware, maar deze werken vaak slechts op of met bepaalde platformen, kunnen niet over lange perioden profileren en zijn zeer duur [26, 14, 16, 21, 10].

1.2 Doelstelling

Een van de projecten die ontwikkeld worden in het Computer Systems Lab (CSL) is een hypervisor. Deze software heeft twee hoofddoelen: langs de ene kant moet het de mogelijkheid bieden verschillende besturingssystemen parallel te draaien, langs de andere kant virtualiseert het legacy hardware. In deze thesis willen we een oplossing ontwikkelen voor het vinden van flessenhalzen in de broncode van dergelijke systeemsoftware, met deze hypervisor als testprogramma.

Het doel van deze thesis is dus een profiler voor systeemsoftware te ontwikkelen die de voorziet in de volgende functionaliteit:

- We willen kunnen profileren met een hoge nauwkeurigheid.
- Het moet mogelijk zijn over langdurige perioden te kunnen meten, in theorie oneindig lang.
- Het analyseproces moet zo weinig mogelijk intrusief zijn, zodat het tijdsgedrag van de systeemsoftware minimaal wordt beïnvloed door het profileren.

1.3 Structuur van dit document

Het vervolg van dit document geeft eerst en vooral een top-down beschrijving van de ontworpen oplossing, waarna een bespreking volgt van de meetopstelling, gevolgd door een evaluatie van de gehele oplossing en mogelijk toekomstig werk.

Hoofdstuk 2 bespreekt de state-of-the-art. We behandelen drie methoden waarop systeemsoftware geprofileerd kan worden, waarbij het nut van onze oplossing duidelijk wordt.

Hoofdstuk 3 bespreekt het top-level ontwerp. We bespreken de grote bouwblokken van onze oplossing, gaande van de systeemsoftware die informatie over de uitvoering doorgeeft tot de opslag van deze informatie.

Hoofdstukken 4 t.e.m. 7 bespreken elk onderdeel van de ontwikkelde oplossing in meer detail. Hierbij hanteren we een top-down benadering: we starten met een hogenniveau beschrijving van de vereiste functionaliteit, en gaan geleidelijk aan in meer detail.

In **hoofdstuk 8** bespreken we hoe we de correcte werking van de ontwikkelde oplossing geverifieerd hebben en wat de beperkingen ervan zijn. We onderzoeken eveneens wat de traagste component van het systeem is.

Hoofdstuk 9 beginnen we met uit te leggen hoe de meetopstelling in de praktijk opgebouwd is, en welke stappen ondernomen moeten worden vooraleer een meting uitgevoerd kan worden. Vervolgens bespreken we enkele uitgevoerde metingen.

Hoofdstuk 10 begint met een algemene conclusie. We sluiten af met enkele suggesties voor toekomstig werk.

Alle broncode die bij dit project hoort, is te vinden op de volgende website:
https://github.ugent.be/jrvdnbro/thesis_publiek.

Hoofdstuk 2

Literatuurstudie

In dit hoofdstuk bespreken we een aantal bestaande oplossingen die mogelijk gebruikt kunnen worden bij het profileren van systeemsoftware. Bij het uitvoeren van een literatuurstudie vonden we drie commerciële oplossingen waarmee we systeemsoftware kunnen profileren:

1. processorspecifieke registers;
2. logic analyzers;
3. gespecialiseerde traceerhardware.

In de volgende secties bespreken we deze mogelijke oplossingen en hun nadelen, waarna we onze eigen oplossing voorstellen.

2.1 Processorspecifieke registers

Een eerste methode waarop systeemsoftware geprofileerd kan worden is door gebruik te maken van registers die voorzien zijn in de processor. Er bestaan twee manieren om m.b.v. registers aan profiling te doen:

1. samplen van het adres van de huidige instructie;
2. gebruik maken van *performance counters*.

Elke processor bevat een dat het adres van de uitgevoerde instructie bijhoudt. Door het tijdsverloop van de waarde van dit register te bekijken, kan informatie over de uitvoering van de systeemsoftware verkregen worden. Zo weten we hoeveel keer eenzelfde blok code wordt uitgevoerd en hoeveel instructies dit in beslag neemt. Het samplen van het adresregister kan zowel gebeuren met een vaste frequentie als op willekeurige tijdstippen.

Naast het bemonsteren van het adresregister kunnen we ook nog gebruik maken van *performance counters*. Dit zijn speciale registers waarvan de waarde elke klokcyclus verhoogd wordt. Deze registers kunnen gereset, in- of uitgeschakeld en uitgelezen worden door bepaalde instructies uit te voeren. Door het verschil te nemen tussen twee samples van eenzelfde *performance counter* kan berekend worden hoeveel tijd er verstrekken is tussen de twee samples. Een voorbeeld van een profiler die met performance counters werkt, is OProfile [34].

De eerste methode gaat ervan uit dat een functie lang genoeg duurt zodat het adres zeker bemonsterd wordt tijdens het uitvoeren van de functie. In ons geval, waarbij functies zeer kort kunnen zijn, levert deze methode te weinig informatie op. De tweede methode is minder generiek omdat niet elke processor beschikt over performance counters. Bovendien neemt het uitlezen van een performance counter register veel tijd in beslag. Dit zorgt ervoor dat metingen over een korte periode vrij onnauwkeurig zijn.

2.2 Logic analyzers

Een logic analyzer wordt gebruikt om digitale signalen te observeren van een System Under Test (SUT). De analyzer zet de geobserveerde data om in tijdsdiagramma's en kan zo bijvoorbeeld de werking van toestandsmachines of het spanningsverloop van logische signalen grafisch weergeven [23].

Het SUT kan met de logic analyzer verbonden worden door gebruik te maken van een speciale connector, of door een aantal aparte probes te verbinden. Een logic analyzer kan de te observeren signalen over het algemeen op twee manieren bemonsteren. Ten eerste kan dit gebeuren aan een vaste frequentie die bepaald wordt door een interne of externe klok; we spreken van *timing mode*. Ten tweede kunnen de signalen geobserveerd worden door een van de te observeren signalen als klok te gebruiken; we spreken dan van *state mode*.

Een belangrijk nadeel van logic analyzers is dat we de *state mode* niet kunnen gebruiken omdat we het tijdsverloop van de te observeren signalen niet op voorhand kennen. We zijn dus verplicht te kiezen voor de *timing mode* die dan weer continu bemonstert. Voor onze toepassing, waarin er langdurige perioden kunnen zijn waarin niets gebeurt, zal er onnodig veel data gegenereerd worden. Vermits de geheugenruimte meestal beperkt is, zullen we niet lang kunnen meten. In de volgende alinea's bespreken we een aantal logic analyzers om deze stelling te duiden. Tabel 2.1 geeft een overzicht van de specificaties van de logic analyzers die we hier bespreken.

Een eerste logic analyzer is de Saleae Logic16 [20], een USB-apparaat dat enerzijds verbinding maakt met een PC, en anderzijds via probes verbonden wordt met het SUT. De grootte van het intern geheugen is gegeven in aantal samples, de grootte ervan in bytes was niet terug te vinden op de website [20]. Het geheugen lijkt op het eerste zicht geen probleem te zijn en ook de prijs is laag ten opzichte van de andere logic analyzers die we zullen bespreken. We zien echter dat de sampleresolutie halveert telkens het aantal kanalen verdubbelt, wat een significant nadeel van dit toestel is.

De volgende twee logic analyzers zijn van Tektronix, een bekende fabrikant van high-end test- en meetapparatuur [27]. Naast onder andere oscilloscopen, signaalgeneratoren en spectrum analyzers produceren ze ook logic analyzers. De twee toestellen die op de site vermeld staan, zijn de TLA6400 en de TLA7000 [14]. De grote troef van deze toestellen is de sampleresolutie en het aantal mogelijke kanalen dat gelijktijdig bemonsterd kan worden. De nadelen zijn echter de hoge kostprijs en het beperkte buffergeheugen. Door dit laatste is de periode waarover gesampled kan worden beperkt.

Een derde logic analyzer is de Teledyne LeCroy LogicStudio [26]. Het is een USB-apparaat dat langs de ene kant verbonden moet worden met een PC en langs de andere kant via probes

verbonden wordt met het SUT. Net zoals de vorig besproken logic analyzers stelt dit toestel ons slechts in staat over een beperkte periode te samplen.

Tenslotte bespreken we nog een logic analyzer van Agilent Technologies, een bedrijf dat net zoals Tektronix bekend staat om zijn oscilloscopen, signaalgeneratoren en andere test- en meetapparatuur [10]. De technische specificaties van dit apparaat zijn te zien in Tabel ???. Agilent biedt het toestel aan met een standaardconfiguratie, maar er zijn opties beschikbaar (zoals meer kanalen en meer geheugen) om de mogelijkheden van het toestel uit te breiden. Wij beschouwen hier het basismodel. In tegenstelling tot de vorige logic analyzer, is dit toestel in staat 34 kanalen tegelijkertijd te samplen op 500 MHz. Ook beschikt deze logic analyzer over veel meer geheugen, maar er hangt wel een stevig prijskaartje aan vast.

2.3 Gespecialiseerde traceerhardware

Een alternatief voor logic analyzers is gespecialiseerde traceerhardware. Wij bespreken hier twee oplossingen die het mogelijk maken ARM-gebaseerde computersystemen te profileren.

2.3.1 ARM DSTREAM en DS-5

Voor het debuggen en profileren van ARM-gebaseerde platformen voorziet ARM zelf in een oplossing met de naam “DSTREAM” [12]. Bij CSL stond voor deze thesis zo’n toestel ter beschikking. Dit apparaat maakt via USB of Ethernet verbinding met een hostcomputer. Het SUT wordt verbonden via een JTAG-poort, een Serial Wire Debug (SWD)-poort of een ARM-specifieke trace poort.

De DSTREAM komt samen met een softwarepakket, DS-5. Een van de onderdelen van dit pakket is een *trace debugger* [15]. Bij het uitvoeren van instructies op het SUT worden de adressen en de instructies opgeslagen in een buffergeheugen. Dit buffergeheugen kan ofwel een on-chip tracebuffer zijn, ofwel het interne geheugen van de DSTREAM zelf. De eerste optie maakt gebruik van een on-chip circulaire buffer die, in het geval van het BeagleBoard, de 1000 laatste instructies bijhoudt. Voor onze toepassing is dit onvoldoende. De tweede mogelijkheid vereist een speciale traceconnector op het SUT die op de gangbare en betaalbare “development kits” nooit aanwezig is.

Tijdens de ontwikkeling van onze oplossing hebben we bovendien gemerkt dat de uitvoeringstijd van de software op het SUT merkelijk langer wordt door gebruik te maken van de DSTREAM. Als we de hypervisor rechtstreeks uitvoeren op het SUT, krijgen we een uitvoeringstijd van ongeveer 15 seconden. Voeren we de hypervisor echter uit via de DSTREAM, dan neemt de uitvoeringstijd toe tot 23 seconden, een stijging van 53%. Het feit of we effectief gebruik maken van de trace buffer heeft geen impact op de uitvoeringstijd.

De DSTREAM houdt bij het traceren en samplen echter geen rekening met timing, en is dus voor onze toepassing niet geschikt. Net zoals de meeste logic analyzers is de kostprijs relatief hoog: voor de DSTREAM betaal je € 2800, voor het softwarepakket DS-5 komt daar € 7200 bij [16, 13].

2.3.2 Lauterbach TRACE32

De Lauterbach TRACE32 is een alternatief voor de DSTREAM van ARM. Het maakt verbinding met het SUT via een JTAG- of een SWD-interface en slaat een trace op van de uitgevoerde instructies in een intern buffergeheugen. Dit toestel kan aangeschaft worden voor \$ 4982.00 [21].

2.4 Besluit

De grootste problemen bij de bestaande oplossingen zijn de hoge kostprijs en het feit dat ze over een beperkte periode kunnen samplen. Een ander probleem is de verbinding tussen het meetplatform en het testplatform: meestal is hier een speciale connector voor nodig. Wij ontwikkelden een systeem dat generiek is, geen exotische connectoren nodig heeft en relatief goedkoop is.

	Seleae Logic16	TLA6400	TLA7000	LeCroy LogicStudio	Agilent 16801A
Aantal kanalen	16	34-136	34-680	16	34
Sampleresolutie	2 kan. aan 100 MHz 4 kan. aan 50 MHz 8 kan. aan 25 MHz 16 kan. aan 12.5 MHz	40 ps	20 ps - 125 ps	8 kan. aan 1 GHz 16 kan. aan 500 MHz	500 MHz 1 miljoen samples
Intern geheugen	10 miljard samples	2 Mb - 64 Mb	512 Mb	40000 samples (8 kan.) 20000 samples (16 kan.)	(uitbreidbaar tot 32 miljoen)
Prijs	€ 300	\$ 11800	\$ 14000	\$ 1361.84 [24]	€ 10024

Tabel 2.1: Specificaties van enkele logic analyzers.

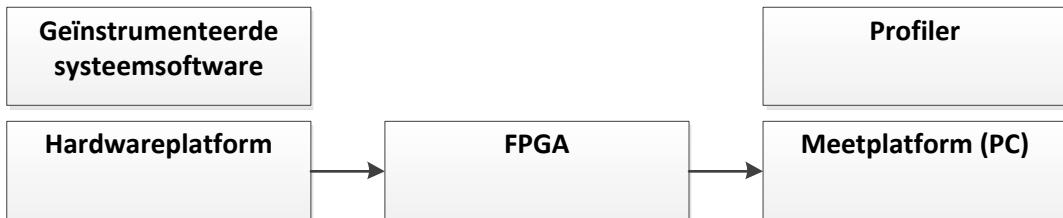
Hoofdstuk 3

Top-level ontwerp

In dit hoofdstuk geven we een globaal beeld van de ontworpen oplossing. Het ontwerp van deze oplossing is gebaseerd op een eerdere implementatie van Niels Penneman [44, 45]. We onderscheiden de volgende basiscomponenten:

1. een testplatform waarop de te profileren systeemsoftware draait;
2. een observatieplatform dat de uitvoering van de systeemsoftware monitort;
3. een meetplatform waarop we de observaties verwerken.

Figuur 3.1 geeft deze verschillende top-level componenten weer. Links staat het testplatform waarop de systeemsoftware draait. Rechts zien we het meetplatform waarop de profiler draait. In het midden bevindt zich de FPGA die het testplatform observeert en deze observaties op een gepaste manier naar het meetplatform zendt. De volgende secties gaan dieper in op elke component van dit ontwerp.

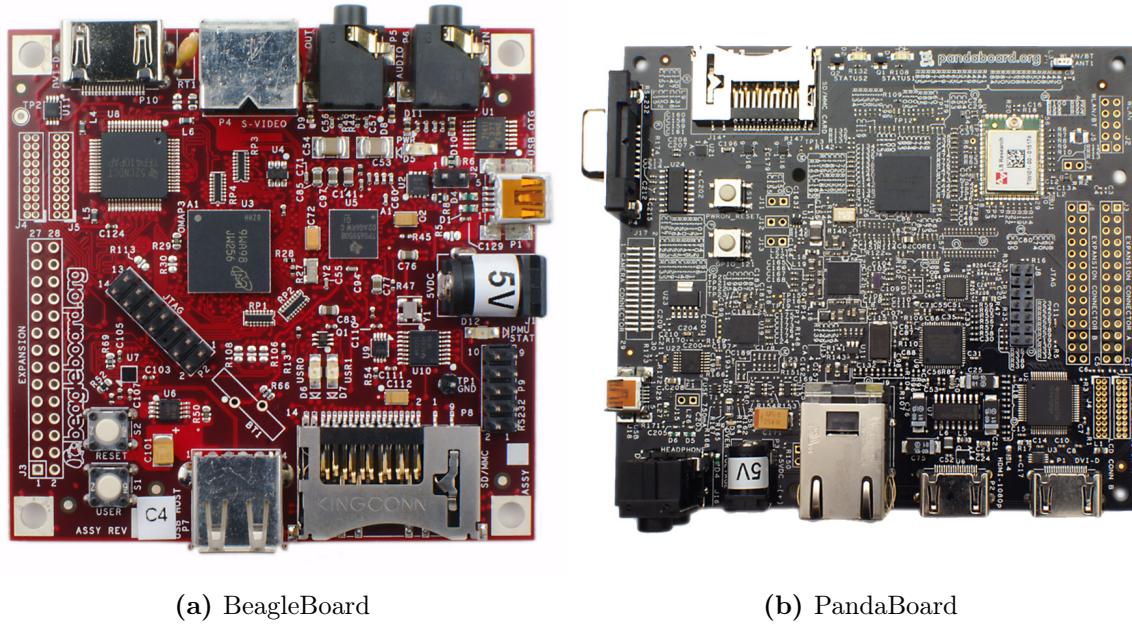


Figuur 3.1: Blokschema van de meetopstelling.

3.1 Het testplatform

Voor het testen van de ontwikkelde oplossing maken we gebruik van enkele testbordjes die ter beschikking stonden. We gebruiken een BeagleBoard en een PandaBoard. Foto's van deze bordjes zijn te zien in Figuur 3.2.

Beide bordjes zijn ontwikkeld door Texas Instruments en werken met OMAP-processoren die gebruik maken van de ARMv7-instructieset. Vermits zowel het BeagleBoard als het



Figuur 3.2: Gebruikte testborden (niet op schaal)

PandaBoard dezelfde instructieset gebruiken is het verschil gezien vanuit de systeemsoftware beperkt.

De softwarestapel van de testbordjes is te zien in Figuur 3.3. De hardware beschikt over een aantal General Purpose Input/Output (GPIO)-pinnen die we gebruiken om informatie over de uitvoering van de systeemsoftware door te geven aan de FPGA. Waarom we voor GPIO gekozen hebben staat beschreven in Sectie 3.1.1. Om de GPIO-pinnen aan te sturen moet de systeemsoftware aangepast worden. We spreken van *geïnstrumenteerde systeemsoftware*.



Figuur 3.3: Blokschema van het testplatform.

Een belangrijke randconditie voor de rest van het ontwerp is de maximale frequentie waarmee de GPIO-pinnen kunnen veranderen. Bij het BeagleBoard is dit ongeveer 2.2 MHz, voor het PandaBoard ongeveer 6.3 MHz. Hoe we aan deze frequentie komen, wordt uitgelegd in Bijlage B.

3.1.1 Verbinding met de FPGA

We willen een oplossing ontwikkelen die voor zowel het testbord als het FPGA-bord zo generiek mogelijk is. Op die manier kunnen we ervoor zorgen dat ook nog andere testborden gebruikt kunnen worden dan diegene die we hier gebruiken. De meest generieke manier om informatie door te geven is door gebruik te maken van GPIO-pinnen. Het gebruik van dit soort pinnen heeft de volgende voordelen:

1. Ze zijn eenvoudig aanstuurbaar: een paar instructies volstaan om de logische toestand van een pin in te stellen.
2. De instelling gebeurt snel: GPIO-pinnen hangen vrijwel altijd rechtstreeks aan de CPU, en er bevinden zich geen componenten tussen de pin en de CPU.
3. Er is geen protocol nodig om ze in te stellen: de gewenste toestand wordt ingesteld door een configuratiebit te schrijven in het geheugen.

Zowel de beschouwde testplatformen als het FPGA-bord beschikken over voldoende GPIO-pinnen om dit project tot een goed einde te brengen.

3.1.2 Informatie uit de systeemsoftware halen

De systeemsoftware die wij profileren is de hypervisor die ontwikkeld wordt binnen CSL. We moeten enkele aanpassingen doorvoeren om deze software de informatie zichtbaar te maken die wij willen observeren. Deze aanpassingen omvatten:

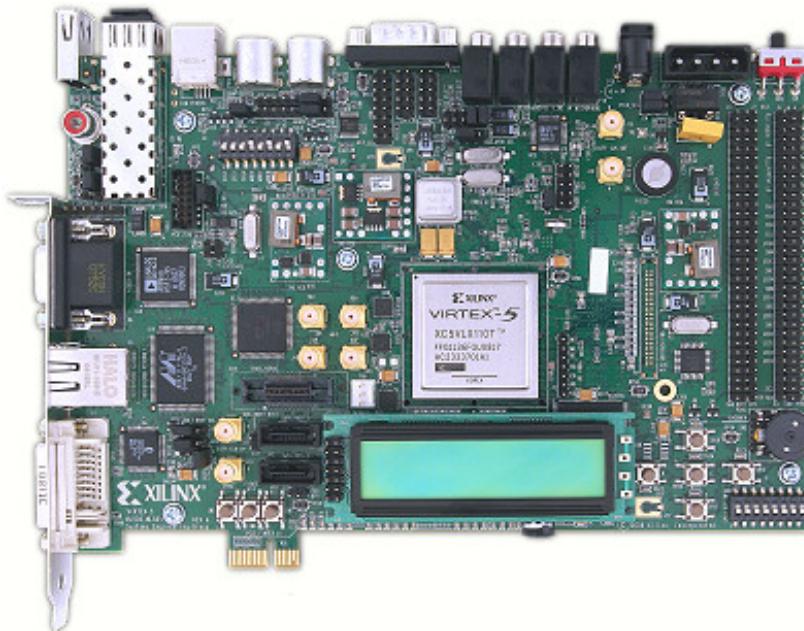
1. een functie die de GPIO functionaliteit initialiseert;
2. code die een bepaalde GPIO-pin hoog brengt;
3. code die een bepaalde GPIO-pin laag brengt.

Vlak voor het opstarten van de hypervisor voeren we een functie uit die de GPIO-functionaliteit initialiseert. Zoals besproken wordt in Sectie 4.3, moet de GPIO-pinnen eerst en vooral op de juiste manier verbonden worden met de CPU. Om zoveel mogelijk implementaties toe te laten, voorziet de CPU immers in multiplexers die het mogelijk maken één pin van de CPU te verbinden met verschillende signaallijnen. Eén van deze signaallijnen is bij een aantal CPU-pinnen een GPIO-lijn. Tenslotte moeten deze pinnen ingesteld worden als uitgang.

Eens de hypervisor opgestart is, sturen we de GPIO-pinnen aan met twee functies: `setGPIO` en `clearGPIO`. Dit zijn functies die door de compiler steeds geïnlined worden en slechts 4-7 instructies omvatten. Deze functies vereisen één argument, namelijk het nummer van de in te stellen GPIO-pin. De GPIO-pinnenummers worden geabstraheerd door het definiëren van preprocessor constanten.

3.2 Het observatieplatform

Om de signalen die de testbordjes uitsturen te observeren, maken we gebruik van een FPGA. Het bord dat voor deze thesis ter beschikking stond is het Xilinx University Program Virtex 5 (XUPV5) bord. Een hiervan is te zien in Figuur 3.4.



Figuur 3.4: Het observatieplatform: een XUPV5 bord.

Het voordeel van een dergelijk FPGA-bord is dat er vele verbindingsmogelijkheden beschikbaar zijn. Zo voorziet het bord onder andere in een Ethernetpoort, een RS-232 seriële poort, een PCI Express (PCIe) connector en een USB-poort. De aanwezigheid van zo'n gamma aan interfaces maakt het voor ons mogelijk de meest geschikte interface te kiezen.

Door gebruik te maken van een FPGA kunnen we een ontwerp maken op hardwareniveau. Dit heeft als voordeel dat het observeren van het testplatform op een snelle manier kan gebeuren. Als we hiervoor gebruik zouden maken van een softwarepakket dat op een computer draait, kunnen er problemen optreden omwille van de volgende redenen:

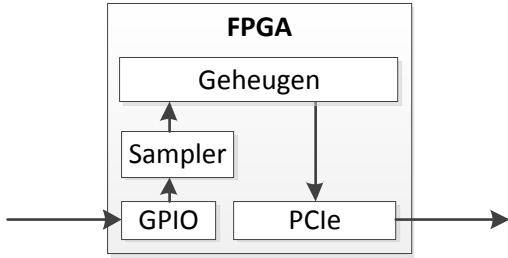
1. Een computer voorziet in het algemeen niet in GPIO-aansluitingen.
2. De observatie-applicatie zal niet de enige toepassing zijn die op de computer draait. Hierdoor kan het zijn dat niet elke verandering van de GPIO-signalen tijdig opgemerkt worden. We hebben dus nood aan een real-time systeem, een FPGA is dit per definitie.

Figuur 3.5 geeft een blokschema weer van het FPGA-ontwerp. De FPGA maakt verbinding met het testplatform via GPIO-pinnen. De signalen op deze pinnen worden geobserveerd door een sampler die zijn observaties opslaat in een buffer. De inhoud van de buffer wordt overgeplaatst naar het geheugen van het meetplatform, via PCIe.

3.2.1 De sampler

In essentie bestaan er twee soorten samplers: continue en gebeurtenis gebaseerde samplers.

Een continue sampler genereert samples aan een vaste frequentie f_s . Hoe hoger de sample-frequentie, hoe hoger de resolutie van de observaties. Het voordeel van dit soort samplers is



Figuur 3.5: Blokschema van het observatieplatform.

dat de verstreken tijd tussen twee samples meteen gekend is: trek de indices van het start- en het eindsample van elkaar af en vermenigvuldig dit met de sampleperiode. Continue samplers hebben echter een significant nadeel: hoe hoger de samplefrequentie, hoe hoger het datadebiet, m.a.w. hoe meer data er gegenereerd wordt. Om te sampelen over lange perioden, wat het doel is van deze thesis, is er dus een gigantische hoeveelheid geheugen nodig. Dit grote geheugen zou een aanzienlijke vertraging van het systeem kunnen impliceren.

Een gebeurtenis gebaseerde sampler genereert enkel een sample op het moment dat er een verandering plaatsvindt aan de ingang. In tegenstelling tot de continue sampler genereert dit type bijgevolg over het algemeen veel minder data. De samples moeten in dit geval uit een tijdswaarde en een observatie bestaan. Deze tijdswaarde houden we bij door een interne timer te voorzien, zoals in meer detail uitgelegd wordt in Sectie 6.2.

We kiezen voor een gebeurtenis gebaseerde sampler omdat we over lange periodes willen meten en omdat de ingang van de sampler voldoende traag varieert.

3.2.2 Het geheugen

De samples worden eerst op de FPGA opgeslagen. Het FPGA-bord voorziet hiervoor in een aantal mogelijke oplossingen [60]:

1. max. 576 kB Block RAM (BRAM) intern aan de FPGA [4];
2. 256 MB DDR2 RAM;
3. 2 GB CompactFlash-kaart;
4. 32 MB lineair flash-geheugen;
5. 1 MB SRAM.

De keuze voor een on-chip oplossing zou de beste zijn qua lees- en schrijfsnelheid, maar de kleine hoeveelheid geheugen beperkt onze mogelijkheden. Bovendien zorgt het gebruik van BRAM voor een grotere bezetting van de FPGA, waardoor we riskeren dat de routering en optimalisatie van de “nuttige” componenten moeilijker wordt. Een off-chip oplossing geniet dus de voorkeur.

Het gebruik van DDR-geheugen vereist een hoop additionele logica en sturing. Deze extra logica neemt plaats in op de FPGA, en maakt het optimaliseren van de implementatie naar snelheid moeilijker [44].

De CompactFlash interface wordt gebruikt om de FPGA te configureren, en is niet beschikbaar als opslagmedium.

Een andere optie zou het lineair flash-geheugen kunnen zijn. Van deze chip kan data geschreven en gelezen worden aan een maximumfrequentie van 52 MHz [38]. Ten opzichte van het SRAM-geheugen is dit echter vier keer trager. Mocht het een snellere geheugenchip zijn, dan zou het flash-geheugen een goede keuze zijn voor onze toepassing.

Het SRAM-geheugen is eenvoudig aan te sturen, en kan, in tegenstelling tot het flash-geheugen, geklokt worden op 200 MHz. Een nadeel ten opzichte van het flash-geheugen is dan weer de grootte: de SRAM-chip heeft slechts een capaciteit van 1 MB. Dit weegt echter niet op tegen de snelheid.

De beste optie is dus het SRAM-geheugen. Om dit aan te sturen voorziet Xilinx in een controller, maar deze vereist een extra bus protocol: de Local Memory Bus (LMB) of de Processor Local Bus (PLB). Omdat deze component een IP core is, wordt het gebruikt als een *black box*. Wij willen volledige controle over de stuursignalen, dus implementeren we een eigen controller.

3.2.3 Verbinding met de PC

Het FPGA-bord dat voor deze thesis ter beschikking stond, voorziet in een aantal verschillende manieren om verbinding te maken met een computer [60]. De vooropgestelde eisen voor deze verbinding zijn:

1. Het datadebit moet minstens zo hoog zijn als de snelheid waarmee het testbord data kan genereren.
2. Dataoverdracht moet mogelijk zijn zonder softwarestapel op de FPGA. Het gebruik van software vereist immers een softcore processor die veel plaats inneemt op de FPGA, waardoor het geheel moeilijker te optimaliseren valt.

Tabel 3.1 geeft een vergelijking weer tussen de verschillende verbindingsmogelijkheden. Uit deze tabel blijkt dat niet elke interface even geschikt is voor onze toepassing.

	Threoretische max. snelheid	Software nodig?
RS-232 Seriële Poort	92 kbps	Nee
Ethernet	1 Gbps	Ja
USB 2.0	480 Mbps[1]	Ja
PCIe 1.0	2 Gbps[61]	Nee
SATA	1.5 Gbps	Nee

Tabel 3.1: Vergelijking van de verbindingsmogelijkheden FPGA-PC

De snelheden die in deze tabel gegeven worden, zijn de theoretische maxima van elke interface. Bij het gebruik van pakketgebaseerde protocollen zal data-overdracht steeds trager verlopen door overhead inherent aan het gebruikte protocol. Voor de seriële poort werd de maximumsnelheid gegeven als 115200 baud. Om dit getal om te rekenen naar een bytesnelheid moeten we het delen door 10 [8].

De Serial ATA verbinding kunnen we niet gebruiken, omdat de controller op het FPGA-bord een *host*-controller is: er kan enkel een harde schijf of iets dergelijks aan gekoppeld worden, geen computer. We besluiten dat PCIe de beste keuze is om te communiceren met de PC.

Om het debiet te maximaliseren maken we gebruik van een Direct Memory Access (DMA)-controller voor dataoverdrachten. Dit ontlast de CPU en zorgt ervoor dat de dataoverdracht zo snel mogelijk gebeurt door blokken data in één keer over te zetten. Het enige dat de CPU hoeft te doen, is de controller instellen: van waar de data komt, naar waar ze moet en hoeveel data er overgedragen moet worden. Tijdens de dataoverdracht is de CPU beschikbaar voor andere taken, zoals eerder ontvangen data verwerken. Wanneer de overdracht voltooid is, wordt de CPU hiervan op de hoogte gesteld met een interrupt die gegenereerd wordt door de DMA-controller.

3.2.4 Communicatie op de FPGA

De FPGA bevat verschillende componenten die met elkaar verbonden moeten worden. Dit zijn er meer dan twee, dus is er een busprotocol nodig om de onderlinge communicatie in goede banen te leiden.

De standaard systeembus die aangeboden wordt door Xilinx is de PLB. Dit is een busprotocol dat speciaal ontworpen werd om blokken in ASIC- en System on chip (SOC)-ontwerpen op een standaardmanier met elkaar te verbinden.

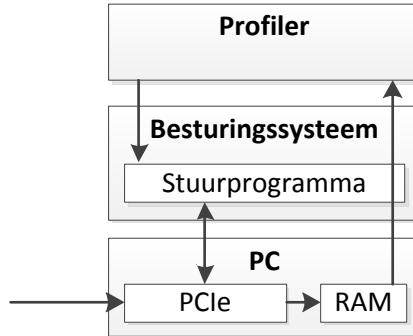
De PLB-bus is gebaseerd op het master-slave principe. Dit impliceert dat een component die een dataoverdracht moet kunnen initiëren, moet beschikken over een master-interface. Omgekeerd, indien een component moet kunnen antwoorden op een verzoek, dient deze te beschikken over een slave-interface. Xilinx levert softcore IPs voor beide interfaces, zodat deze enkel geïnstantieerd hoeven te worden.

3.3 Het meetplatform

Figuur 3.6 geeft het blokschema van het meetplatform weer. We gebruiken een computer waarbij we het FPGA-bord in een vrij PCIe-slot steken. De samples komen de PC binnen via de PCIe-bus, waarna ze in het RAM-geheugen worden opgeslagen. Om dit alles te mogelijk te maken, is een stuurprogramma vereist. Dit stuurprogramma maakt het eveneens mogelijk de sampler op de FPGA te besturen vanuit een gebruikersapplicatie.

3.3.1 Het stuurprogramma

Zonder stuurprogramma kan het besturingssysteem niet weten hoe het met het FPGA-bord moet communiceren. De verschillende taken die het stuurprogramma moet vervullen, zijn:



Figuur 3.6: Blokschema van het meetplatform.

- initialiseren van het FPGA-bord;
- controle van de sampler mogelijk maken;
- DMA-overdrachten configureren en initiëren;
- binnengekomen data doorgeven aan de gebruiker.

3.3.2 Het meetprogramma

We kunnen het meetprogramma implementeren op twee manieren. De eerste manier houdt in dat we de samples verwerken van zodra ze de PC binnenkomen. Dit vereist echter een programma dat bijzonder snel gegevens kan verwerken, en dat in geen geval mag vastlopen. De tweede manier bestaat uit twee stappen: eerst streamen we de samples naar een bestand op de harde schijf, waarna we het bestand verwerken eens alle samples opgeslagen zijn.

Praktisch gezien is de tweede manier de beste oplossing. We ontwikkelden eerst een streaming-applicatie die de binnengekomen data zo snel mogelijk naar de harde schijf wegschrijft. De observaties die we met deze applicatie verkregen hebben, visualiseren we met een ander softwarepakket dat hierin gespecialiseerd is. Voor deze thesis hebben we gebruik gemaakt van ModelSim.

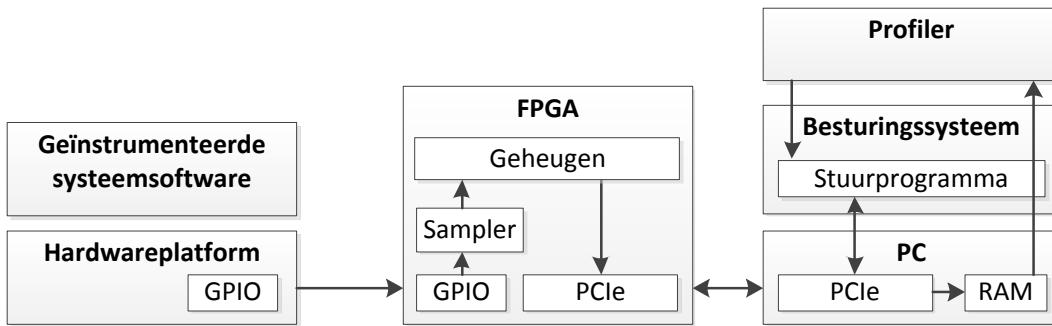
3.4 Samenvatting

Een samenvatting van het gehele systeem is te zien in Figuur 3.7. Van links naar rechts onderscheiden we het testplatform, het observatieplatform en het meetplatform. Een schatting van de totale kostprijs van onze oplossing is te vinden in Tabel 3.2. De prijzen hebben we opgezocht op de website van Farnell, tenzij anders vermeld [17]. Het BeagleBoard dat wij gebruikt hebben, is revisie C4. Het C5-bord kost echter evenveel. De SD-kaart kan gebruikt worden om software op te laden in het testbord. Voor het FPGA-bord hebben we de niet-academische prijs genomen. Academische instellingen kunnen dit bord aankopen voor \$ 750. Het grootste deel van de kosten voor het interfacebord zijn in ons geval gegaan naar adapters voor de level shifter IC's. Deze IC's hebben immers geen DIP-verpakking, d.w.z. ze hebben geen verticale pinnetjes, maar hebben een vlinderverpakking. Om deze vlinderverpakking om

te zetten naar een DIP-verpakking gebruiken we adapters. Deze kosten per stuk een kleine € 25 [11].

Onderdeel	Prijs
BeagleBoard C5	€ 142.66
PandaBoard ES	€ 161.75
4 GB SD-kaart	€ 12.34
XUPV5 FPGA-bord[9]	\$ 2000.00
Interfacebord	€ 65.00
Totaalprijs	€ 2219.00 (BeagleBoard) € 2238.19 (PandaBoard)

Tabel 3.2: Schatting van de kosten voor dit project



Figuur 3.7: Top-level ontwerp van het gehele systeem.

Hoofdstuk 4

Interface testplatform-FPGA

In dit hoofdstuk bespreken we de verbinding tussen de testplatformen en de FPGA. Eerst en vooral bekijken we de elektrische compatibiliteit tussen beide componenten. Er blijkt een level shifter nodig te zijn om spanningen om te zetten van het ene naar het andere platform. Na een gedetailleerde bespreking van deze level shifter geven we een overzicht van het gebruik en de werking van de GPIO-pinnen van elk testbord. Tenslotte geven we nog een overzicht van de GPIO-pinnen die we gebruikt hebben op het FPGA-bord.

4.1 Elektrische compatibiliteit

Het verbinden van twee elektronische systemen moet altijd voorzichtig gebeuren. Er moet speciale aandacht besteed worden aan de spanningsniveaus, maar ook aan de maximaal toegelaten stroom die uit het bordje getrokken mag worden.

Het spanningsniveau van het BeagleBoard is gelijk aan 1.8 V [32]. De maximale stroom die getrokken kan worden uit de 1.8 V-voedingslijn is gelijk aan 30 mA. Hierbij is die 30 mA de *totale* stroom die onttrokken mag worden. Op het BeagleBoard zitten ook nog componenten die aan de deze voedingslijn hangen, waardoor 30 mA een optimistische limiet is [31].

Het PandaBoard is zeer gelijkaardig aan het BeagleBoard: ook hier geldt een spanningsniveau van 1.8 V [39, 42]. De I/O spanning wordt hier geleverd door een 1.2 A 1.8 V bron. Opnieuw dient rekening te worden gehouden met het feit dat er al heel wat elektronica aan deze spanningsbron hangt op het bordje [40, 41]. De beschikbare stroom aan de uitbreidingsconnector is dus zeker niet 1.2 A.

Uit de gebruikershandleiding van het FPGA-bord concluderen we dat de logische spanning ingesteld kan worden op 2.5 V of 3.3 V [60]. Het gewenste spanningsniveau kan gekozen worden door het instellen van jumper J20. Om praktische redenen werd hier gekozen voor een spanningsniveau van 3.3 V, zie ook Sectie 4.4. De 3.3 V-voeding kan een maximale stroom leveren van 10 A, hiervoor verwijzen we naar p.34 van de gebruikershandleiding [60].

Het verschil in spanningsniveaus zorgt ervoor dat er een omzetting moet gebeuren bij het verbinden van de bordjes met de FPGA. Deze omzetting doen we aan de hand van een interfacebordje.

4.2 Het interfacebord

Het interfacebord dat we in deze sectie bespreken, zorgt voor een probleemloze verbinding tussen de testbordjes en het FPGA-bord. We willen dit bordje zo generiek mogelijk maken, zodat we zowel het Beagle- als het PandaBoard via dit bord kunnen verbinden met het FPGA-bord. Het bordje moet de volgende componenten bevatten:

1. connectoren voor het Beagle- en PandaBoard;
2. een connector voor het FPGA-bord;
3. level shifters om de overgang van 1.8 V naar 3.3 V mogelijk te maken;
4. logica om de level shifters uit te schakelen.

4.2.1 De connectoren

De eenvoudigste manier om de testbordjes met ons interfacebord te verbinden, is door gebruik te maken van een flatcable. Vermits de GPIO-signalen voor elk testbordje op een andere plaats in de uitbreidingsconnector zitten, moeten we voorzien in aparte connectoren voor het Beagle- en het PandaBoard.

We maken verbinding met het FPGA-bord door te voorzien in een derde connector die rechtstreeks op de uitbreidingsconnector van het FPGA-bord geplaatst kan worden.

4.2.2 De level shifter

Het basisprincipe van een digitale level shifter is eenvoudig: vertaal een spanning van het ene naar het andere niveau.

Het ontwerp van deze level shifter hebben we gebaseerd op ontwerp van TinCanTools, een organisatie die uitbreidingsbordjes maakt voor o.a. BeagleBoards. Een korte kijk op enkele schema's leert ons dat de component *SN74AVC4T774PW* ook voor onze toepassing geschikt is [7, 6].

De SN74AVC4T774PW, gefabriceerd door Texas Instruments, is een *4-bit dual-supply bus transceiver with configurable voltage translation and 3-state outputs* [48]. Dit betekent dat er met één IC 4 bitlijnen parallel vertaald kunnen worden van het ene naar het andere spanningsniveau. Bovendien kunnen zowel de bron- als de doelspanning ingesteld worden.

Een leuke extra aan dit IC is diens pin om de in- en uitgangen van de level shifter hoogimpedant te maken. Op deze manier kunnen we beide spanningsdomeinen gescheiden houden tot we zeker zijn dat het testbord klaar is om data te verzenden en dat het FPGA-bord klaar is om data te ontvangen. Zo kunnen we voorkomen dat bijvoorbeeld een als uitgang geconfigureerde GPIO-pin op het FPGA-bord verbonden wordt met een als uitgang geconfigureerde GPIO-pin van het BeagleBoard.

4.2.3 In- en uitschakelen van de level shifter

De GPIO-pinnen van het FPGA-bord kunnen zowel als in- of als uitgang gebruikt worden. Om ervoor te zorgen dat ons interfacebord in geen geval signalen opdringt aan de GPIO-pinnen die als uitgang geconfigureerd zijn, maken we gebruik van de isolatiefunctionaliteit van het level shifter IC. De pin van het IC dat dit isolatie mogelijk maakt, is actief-laag. Dit betekent dat de isolatie opgeheven wordt door aan deze pin een lage spanning op te dringen.

Om te bepalen wanneer de level shifters ingeschakeld moeten worden, kiezen we voor drie XOR-poorten waarvan de ingangen verbonden worden met 6 GPIO-pinnen van het FPGA-bord. De uitgangen van de XOR-poorten verbinden we met de ingangen van een NAND-poort. De uitgang van de NAND-poort verbinden we met de isolatiepin van elk level shifter IC.

Met deze logica is het mogelijk de level shifters in of uit te schakelen door het patroon “010101” of “101010” aan de GPIO-pinnen van het FPGA-bord aan te leggen. We zouden dit even goed kunnen bereiken door één GPIO-pin van het FPGA-bord te gebruiken, maar we wilden toch in enige redundantie voorzien. Deze redundantie zorgt er bijvoorbeeld voor dat de kans bijzonder klein is dat de level shifters “per ongeluk” ingeschakeld zouden worden.

De reden waarom we isolatie willen, is dat er in een eerste versie van het interfacebord een BeagleBoard gesneuveld is. De belangrijke les die we hier geleerd hebben is dat dergelijke bordjes niet als stroombron gebruikt kunnen worden.

4.2.4 Het interfacebord

We hebben ervoor gekozen acht GPIO-signalen te observeren van het testplatform. Dit kan relatief eenvoudig uitgebreid worden door enkele parameters in de implementatie te veranderen. Het gebruik van acht GPIO-signalen zorgt ervoor dat we twee level shifter IC's moeten gebruiken. Een blokschema van het interfacebord is te zien op Figuur 4.1. Een foto van het uiteindelijke interfacebord is te zien in Figuur 4.2. Het bedradingsschema is terug te vinden in Bijlage A.

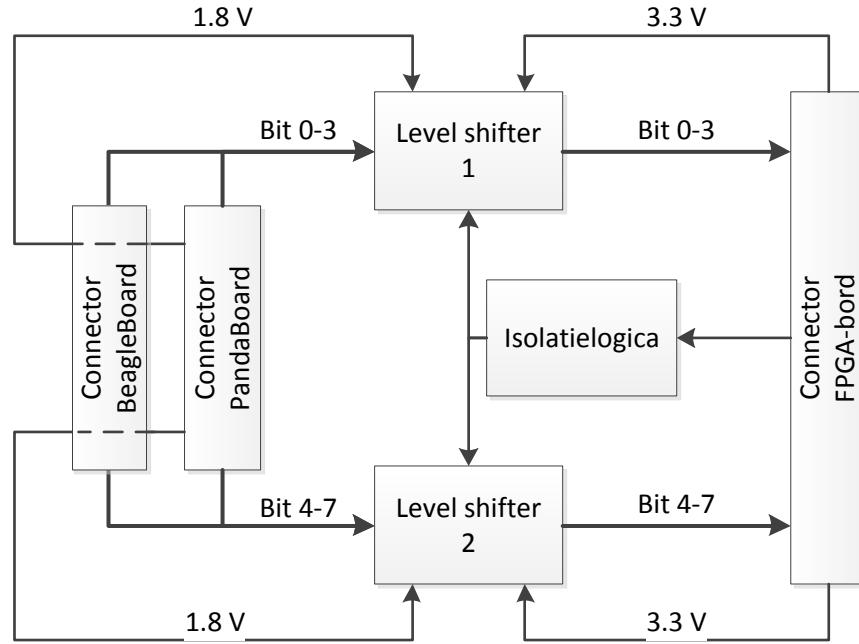
Centraal op Figuur 4.1 zien we de level shifter IC's. Zoals gezegd kan elk IC slechts vier bits vertalen, dus hebben we er twee voorzien. Vermits de level shifter een vertaling uitvoert van het ene naar het andere spanningsniveau, moet deze langs twee kanten gevoed worden.

De uitbreidingsconnectoren van de testbordjes reserveren één van hun pinnen om de I/O voedingslijn te verbinden. We maken hier handig gebruik van door deze voedingslijn te gebruiken voor één kant van de level shifters. Uit de beschikbare pinnen op de uitbreidingsconnectoren kiezen we, per connector, acht GPIO-lijnen die we verbinden met de level shifters.

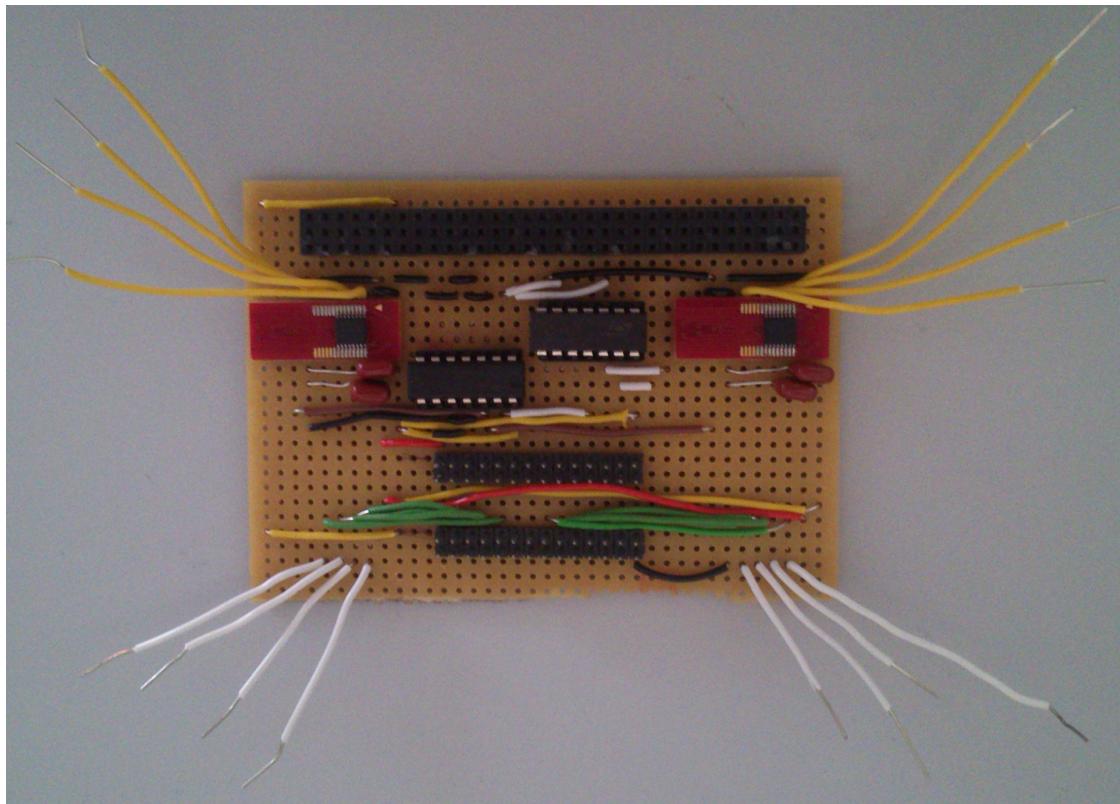
Aan de rechterkant van Figuur 4.1 zien we de connector voor het FPGA-bord. Ook deze connector voorziet in een aantal pinnen waarmee de I/O voedingsspanning verbonden is. We maken gebruik van deze voorziening om de andere kant van de level shifters te voeden.

De uitbreidingsconnector van de FPGA bevat veel GPIO-pinnen. Van deze pinnen kiezen we er acht die we zullen gebruiken als datalijnen, en zes om de isolatielogica te implementeren, zoals besproken in de vorige sectie.

De isolatielogica bestaat uit twee IC's: één IC met vier XOR-poorten en één IC met drie NAND-poorten.



Figuur 4.1: Blokschema van de level shifter.



Figuur 4.2: Foto van het interfacebord.

4.3 GPIO-pinnen van de testbordjes

Zowel het Beagle- als het PandaBoard beschikken over uitbreidingsconnectoren. Een aantal pinnen van deze connectoren zijn GPIO-pinnen, en kunnen dus gebruikt worden om informatie i.v.m. de uitvoering van de systeemsoftware naar buiten te brengen.

Deze IO-pinnen zijn elk verbonden met een eigen multiplexer die ze met de signaallijnen intern in de processor verbindt. Het is dus belangrijk deze multiplexers correct in te stellen zodat de GPIO-signaallijnen beschikbaar zijn op de processorpinnen die doorverbonden zijn met de uitbreidingsconnector van de testbordjes. Bovendien moeten ook nog multiplexers van andere pinnen worden ingesteld, zodat die geen kortsluiting vormen met dezelfde signaallijnen. Het systeem van multiplexers laat immers toe om bepaalde signaallijnen op meer dan één pin naar buiten te brengen. Het is uiteraard nooit de bedoeling om een signaallijn door te verbinden met meer dan één pin tegelijkertijd.

4.3.1 Het BeagleBoard

Een overzicht van de mogelijke signalen op elke uitbreidingspin wordt gegeven in Tabel 20 van de *System Reference Manual* [32]. Om nu te gaan bepalen welke pinnen we zullen gebruiken om data naar de FPGA te sturen, moeten we rekening houden met de volgende voorwaarden:

1. Kan er een GPIO-signalen op de pin geplaatst worden?
2. Wordt de pin van de OMAP-processor waarmee deze pin verbonden is niet gebruikt op het BeagleBoard?

Een bijkomend criterium waarop de keuze gebaseerd werd, is de toegankelijkheid naar de level shifter toe. Door de plaatsing van de componenten op het interfacebordje is het eenvoudiger vier pinnen aan de ene kant van de connector te kiezen, en vier pinnen aan de andere kant. Tabel 4.1 geeft een overzicht van de gekozen GPIO-pinnen. We geven het nummer van de uitbreidingspinnen, samen met de padnaam van de OMAP-processor, de *muxmode* en het GPIO-nummer. De padnaam en de *muxmode* hebben we nodig om de multiplexering in te stellen.

Connector pin nummer	OMAP Padnaam	GPIO nummer	Muxmode
3	MMC2_DAT7	139	4
5	MMC2_DAT6	138	4
7	MMC2_DAT5	137	4
9	MMC2_DAT4	136	4
15	MMC2_DAT1	133	4
17	MMC2_DAT0	132	4
19	MMC2_CMD	131	4
21	MMC2_CLK0	130	4

Tabel 4.1: Overzicht van de gebruikte GPIO-pinnen van het BeagleBoard.

Eens we de pinnen kennen die we wensen aan te sturen, moet de multiplexering goed ingesteld worden. Bij de OMAP-processoren gebeurt dit door een waarde te schrijven naar een *Pad Multiplexing Register*. De bitlayout van zo'n register wordt uitgelegd in Sectie 7.4.4 van de technische handleiding [51]. Het belangrijkste veld voor ons is **MUXMODE**. Dit moet op de waarde ingesteld worden uit Tabel 4.1 om het GPIO-signalen naar buiten te brengen. De velden **PULL**, **INPUTENABLE**, **OFFENABLE**, **WAKEUPENABLE** en **WAKEUPEVENT** kunnen voor onze toepassing veilig op 0 ingesteld worden.

Bij het instellen van een *Pad Multiplexing Register* dient rekening gehouden te worden met het feit dat een 32-bit register twee pads configueert, zoals te zien is in Figuur 7-7 van de OMAP-handleiding [51]. Onze GPIO-pinnen worden bijgevolg correct geconfigureerd bij het instellen van het configuratieregister voor die pin op de waarde **0x0004**.

Door de informatie uit Tabel 4.1 en Tabel 7-4 van de handleiding te combineren kunnen we de geheugenadressen van de multiplexersregisters vinden. Deze adressen worden weergegeven in Tabel 4.2. Hierbij staat "H" voor de bovenste 16 bits van het 32-bit register en "L" voor de onderste 16 bits ervan.

GPIO nummer	Geheugenadres	Positie in register
139	0x48002168	H
138	0x48002168	L
137	0x48002164	H
136	0x48002164	L
133	0x4800215C	H
132	0x4800215C	L
131	0x48002158	H
130	0x48002158	L

Tabel 4.2: Geheugenadressen van de pad configuratie registers voor het BeagleBoard.

De GPIO-pinnen worden 2-aan-2 geconfigureerd in hetzelfde register, waardoor het schrijven van de waarde **0x00040004** naar elk van de vier adressen volstaat.

De OMAP-processor voorziet in 192 GPIO-pinnen. De aansturing van deze GPIO-signalen gebeurt aan de hand van zes modules die elk over een eigen registerbank beschikken. Gezien een register uit 32 bits bestaat, kan elke module 32 GPIO-lijnen aansturen. Volgens Tabel 24-5 van de technische handleiding worden onze gekozen pinnen aangestuurd door module 5 [51].

Hoe de aansturing in zijn werk gaat, wordt uitgelegd in Sectie 24.5.4 van de handleiding [51]. De configuratieregisters voor elke GPIO-module zijn terug te vinden in Tabel 24-7 van de handleiding. Tabel 4.3 geeft een overzicht van de adressen van deze configuratieregisters voor de gebruikte GPIO-pinnen.

Register	Geheugenadres
GPIO_OE	0x49056034
GPIO_DATAOUT	0x4905603C
GPIO_SETDATAOUT	0x49056094
GPIO_CLEARDATAOUT	0x49056090

Tabel 4.3: Geheugenadressen GPIO-configuratieregisters voor het BeagleBoard.

De volgende registers zijn betrokken bij de aansturing van een GPIO-pin:

1. **GPIO_OE** - Een 1-bit in dit register stelt de overeenkomende GPIO-lijn in als input, een 0 stelt deze in als output. Zie ook Tabel 24-27 van de handleiding.
2. **GPIO_DATAOUT** - Elke bit in dit register stelt de waarde van een GPIO-pin in, zie ook Tabel 24-31 van de handleiding.
3. **GPIO_SETDATAOUT** - Een 1-bit in dit register brengt de bijhorende GPIO-pin hoog. Elke bit stuurt 1 GPIO-pin aan, zie ook Tabel 24-59 in de handleiding.
4. **GPIO_CLEARDATAOUT** - Een 1-bit in dit register brengt de bijhorende GPIO-pin laag. Elke bit stuurt 1 GPIO-pin aan, zie ook Tabel 24-57 in de handleiding.

In deze registers wordt GPIO-pin 128 aangestuurd door bit 0 en GPIO-pin 159 door bit 31. Het bitmasker dat onze pinnen selecteert, heeft bijgevolg de volgende binaire voorstelling:

`0b0..0111100111100`, waarbij “`0..0`” de restbits representeert. Als hexadecimale voorstelling krijgen we de waarde `0x00000F3C`. Hier staat een 1-bit voor een gebruikt GPIO-kanaal.

Om de GPIO-kanalen als output in te stellen, schrijven we de waarde `0xFFFFF0C3` naar het `GPIO_0E`-register. Deze waarde heeft een nulbit op elke positie die we wensen te gebruiken als output en is het binaire complement van `0x00000F3C`.

Wanneer dit alles is gebeurd, kunnen we de GPIO-pinnen vanuit de systeemsoftware aansluiten door de gewenste waarden te schrijven naar het `GPIO_DATAOUT`-, `GPIO_SETDATAOUT`- of `GPIO_CLEARDATAOUT`-register.

4.3.2 Het PandaBoard

De keuze van de GPIO-pinnen op het PandaBoard kan op een zeer gelijkaardige manier gebeuren als bij het BeagleBoard. Daarom zullen we hier niet de volledige uitleg herhalen, maar enkel de essentiële verschillen geven.

Een fundamenteel verschil tussen het PandaBoard en het BeagleBoard is dat het PandaBoard over twee uitbreidingsconnectoren beschikt. Details over deze connectoren zijn te vinden in Sectie 2.16 van de *System Reference Manual* van het PandaBoard [39]. Een van de twee uitbreidingsconnectoren kunnen wij niet gebruiken omdat die niet voorziet in een voedingslijn voor de level shifters. We kiezen dus voor connector A [39].

De gekozen pinnen van de uitbreidingsconnector van het PandaBoard staan vermeld in Tabel 4.4. De *muxmode* is deze keer niet te vinden in de *System Reference Manual*, maar staat vermeld in Tabel 18-9 van de technische handleiding [52].

Connector pin nummer	OMAP Padnaam	GPIO nummer	Muxmode
10	MCSPI1_CS1	138	3
12	MCSPI1_SIM0	136	3
14	MCSPI1_CS2	139	3
16	MCSPI1_CS0	137	3
18	MCSPI1_SOMI	135	3
20	MCSPI1_CLK	134	3
23	I2C4_SDA	133	3
24	I2C4_SCL	132	3

Tabel 4.4: Overzicht van de gebruikte GPIO-pinnen van het PandaBoard.

De bitlayout van een *Pad Multiplexing Register* van het PandaBoard vinden we terug in Figuur 18-6 van de technische handleiding [52]. We zien dat de inhoud van dit register gelijk is aan dat van het BeagleBoard, dus het `MUXMODE`-veld is het enige veld dat verschillend is van nul. Uit Tabel 4.4 zien we dat elk *Pad Multiplexing Register* de waarde `0x0003` moeten krijgen.

Door de informatie uit Tabel 4.4 en Tabel 18-9 van de handleiding te combineren kunnen we de geheugenadressen van de multiplexeringsregisters vinden. Deze adressen worden weergegeven in Tabel 4.5. Hierbij staat “H” voor de bovenste 16 bits van het 32-bit register en “L” voor de onderste 16 bits ervan.

GPIO nummer	Geheugenadres	Positie in register
139	0x4A10013C	L
138	0x4A100138	H
137	0x4A100138	L
136	0x4A100134	H
135	0x4A100134	L
134	0x4A100130	H
133	0x4A100130	L
132	0x4A10012C	H

Tabel 4.5: Geheugenadressen van de pad configuratie registers voor het PandaBoard.

Alle pinnen, behalve pinnen 139 en 132, worden 2-aan-2 geconfigureerd. Voor deze pinnen kan het configuratieregister ingesteld worden op `0x00030003`. Bij het configureren van pinnen 132 en 139 moeten we opletten dat we de configuratie van de andere pin die ingesteld wordt door dit register niet wijzigen.

Net zoals het BeagleBoard voorziet het PandaBoard in 192 GPIO-pinnen die aangestuurd worden door zes modules. Zoals uit Tabel 25-10 van de technische handleiding volgt, worden onze GPIO-pinnen aangestuurd vanuit de vijfde module [52]. Deze geheugenadressen van de registers die betrokken zijn bij het aansturen van de GPIO-pinnen zijn te zien in Tabel 4.6.

Register	Geheugenadres
GPIO_OE	0x4805B134
GPIO_DATAOUT	0x4805B13C
GPIO_SETDATAOUT	0x4805B194
GPIO_CLEARDATAOUT	0x4805B190

Tabel 4.6: Geheugenadressen GPIO-configurateregisters voor het PandaBoard.

Het bitmasker dat we moeten gebruiken om onze GPIO-pinnen te selecteren als output, ziet er als volgt uit: `0b0..011111110000`, waarbij “0..0” opnieuw de restbits representeren en “1” duidt op een uitgangspin. De hexadecimale voorstelling is als volgt: `0x00000FF0`.

We stellen de GPIO-kanalen in als output door de waarde `0xFFFFF00F` te schrijven naar het `GPIO_OE`-register. Als dit gebeurd is, kunnen we de GPIO-kanalen instellen aan de hand van de andere drie registers uit Tabel 4.6.

4.4 GPIO-pinnen van het FPGA-bord

Het FPGA-bord voorziet in twee 3x32 uitbreidingsconnectoren. Elke connector bestaat fysisch uit een 2x32 header, en een 1x32 header. De beschrijving van deze connectoren is te vinden op p.22 en volgende in de gebruikershandleiding [60]. Deze connectoren heten Xilinx Generic Interface (XGI) Expansion Headers.

Een van de uitbreidingsconnectoren bezit 32 pinnen die 2-aan-2 differentieel worden aangestuurd. De andere uitbreidingsconnector bezit 32 GPIO pinnen die single-ended aangestuurd worden.

Vermits de GPIO-signalen die de testbordjes genereren single-ended zijn, gaat de voorkeur uit naar het gebruik van de tweede uitbreidingsconnector.

De elektrische schema's van het FPGA-bord geven aan wat er met elkaar verbonden is [54], in het bijzonder de verbindingen met de uitbreidingsconnectoren. Wij gebruiken de connectoren J_4 en J_5 . Deze zijn te zien op p.11 van de schematic [54]. Header J_4 bevat enerzijds een rij massasignalen, en anderzijds een rij GPIO-signalen. Header J_5 bevat o.a. een 3.3 V-voedingsspanning. Deze gebruiken we om de level shifter te voeden.

Om te bepalen welke GPIO-pinnen we gebruiken van connector J_4 , kijken we naar de plaatsing van de componenten op het interfacebordje, zie ook Figuur 4.2. De basisgegevens van de gebruikte GPIO-pinnen staan neergeschreven in Tabel 4.7. We onderscheiden twee soorten signalen: data- en controlesignalen. De datalijnen vormen een verbinding met de GPIO-pinnen van het testbordje, de controlelijnen worden gebruikt om de level shifters in of uit te schakelen. Dit laatste is de isolatielogica uit Figuur 4.1.

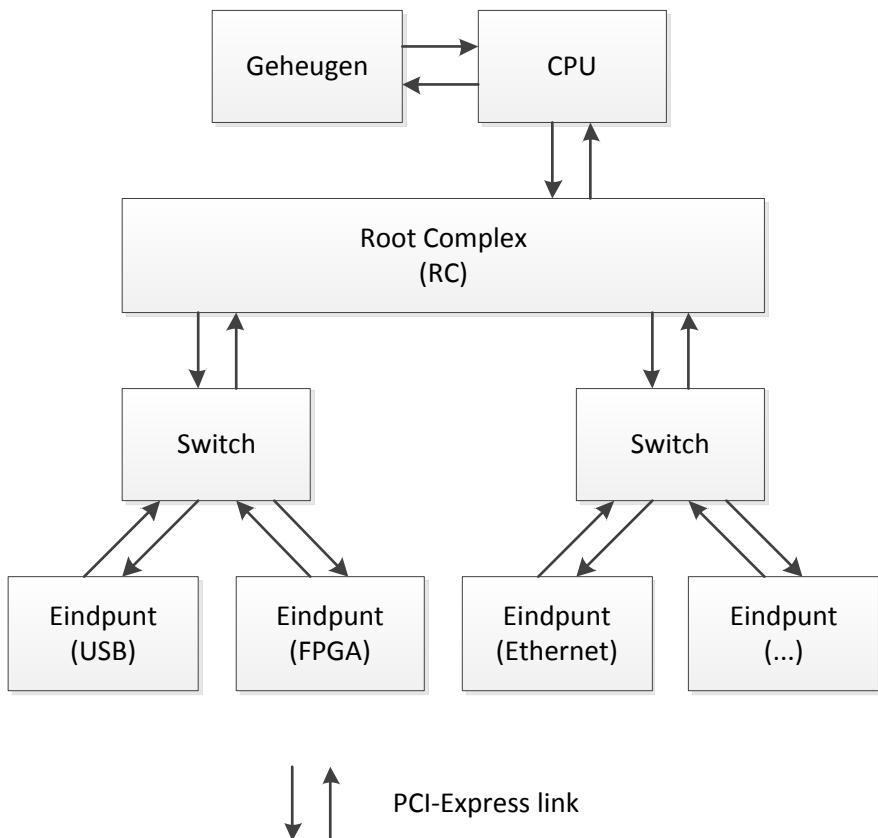
Connector pin nummer	FPGA pin	Beschrijving
<i>Datalijnen</i>		
2	H33	Bit 0
4	F34	Bit 1
6	H34	Bit 2
8	G33	Bit 3
58	AJ34	Bit 4
60	AM32	Bit 5
62	AN34	Bit 6
64	AN33	Bit 7
<i>Controlelijnen</i>		
36	AH34	Bit 0
38	AE32	Bit 1
42	AH32	Bit 2
44	AK34	Bit 3
46	AK33	Bit 4
48	AJ32	Bit 5

Tabel 4.7: Overzicht van de gebruikte uitbreidingspinnen van het FPGA-bord.

Hoofdstuk 5

Interface FPGA-PC

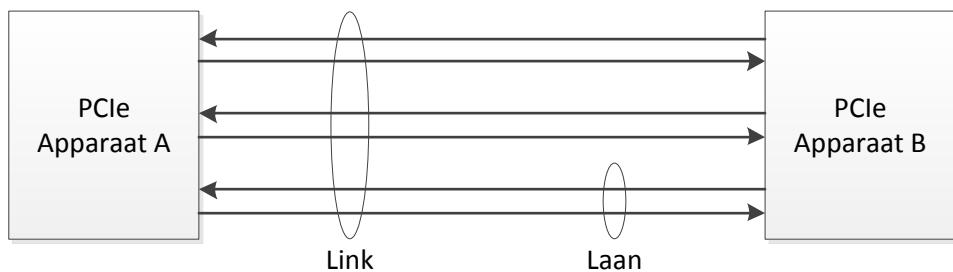
In dit hoofdstuk gaan we dieper in op de gekozen communicatie-interface voor het verbinden van het FPGA-bord met de PC. Figuur 5.1 geeft een vereenvoudigde voorstelling van de PCIe-architectuur. In essentie bestaat deze uit een Root Complex (RC), waaraan een boom met apparaten hangt.



Figuur 5.1: Grafische voorstelling van de PCIe-architectuur.

5.1 Communicatie tussen twee PCIe-nodes

Communicatie met een PCIe-kaart gebeurt aan de hand van een *link* gebaseerd op het point-to-point principe, zie ook Figuur 5.2 [46]. De link bestaat uit één of meer sets signaallijnen, *lanen* genoemd, waarbij elke set signaallijnen een full-duplexverbinding is. Op hardwareniveau is elke signaallijn differentieel uitgevoerd om een hoge bitrate mogelijk te maken. Uitbreidingskaarten met een PCIe-connector worden ingedeeld op basis van de linkbreedte, wat overeen komt met het aantal lanen. Er bestaan connectoren met diverse breedtes: men spreekt van x1, x2, x4, x8, x16 en x32 verbindingen. Over dezelfde lane wordt de data serieel verzonden, maar indien er meerdere lanen beschikbaar zijn, kan de data-overdracht geparalleliseerd worden. Het gebruikte FPGA-bord beschikt over een x1-link.



Figuur 5.2: Schematische voorstelling van een PCIe-verbinding.

De PCIe-standaard legt het maximum datadebiet van één lane op. Versie 1.0 kan een datadebiet aan van 250 MB/s (2 Gbps), versie 2.0 een debiet van 500 MB/s (4 Gbps). De laatste versie van de standaard, versie 3.0, voorziet een datadebiet van 1 GB/s (8 Gbps). Op het FPGA-bord is een PCIe 1.0 interface beschikbaar, waardoor een maximale datarate van 250 MB/s haalbaar is.

Tenslotte moet gezegd dat het PCIe-protocol pakketgebaseerd is. Dit betekent dat er onder andere gewerkt wordt met *memory read packets* en *memory write packets*. Het gebruik van een pakketgebaseerd protocol impliceert dat het theoretisch maximale datadebiet niet gehaald kan worden. Dit komt door de overhead die ontstaat door *framing*: elk pakket krijgt een *header* en eventueel een *footer*, waarin informatie over het pakket zit.

5.2 Communicatie tussen een PCIe-kaart en de CPU

De CPU kan met een PCIe-apparaat communiceren door stuurcommando's te sturen naar het RC.

Opdat de host-CPU data van een PCIe-apparaat zou kunnen uitlezen, krijgt elke PCIe-kaart een plaats in de adresruimte van de Memory Management Unit (MMU): het apparaat wordt *gemapt* in het geheugen van de computer, we spreken van *memory mapped I/O*. Er kan data verzonden worden naar een apparaat door naar zijn mapping in het geheugen te schrijven. Hetzelfde geldt voor het ontvangen van data.

Om controle van de hardware op de PCIe-kaart mogelijk te maken door de PC, en om ervoor te zorgen dat de PC statusinformatie kan opvragen van het PCIe-apparaat, maakt deze laatste

een aantal registers zichtbaar. Door deze registers te mappen in zijn adresruimte kan de processor de nodige status- en controle-informatie opvragen. Het mappen zelf gebeurt aan de hand van een zogenaamd Base Address Register (BAR), zie Sectie 5.4.

5.3 Interrupts

Wanneer een apparaat een taak uitvoert die enige tijd duurt, zoals het overzetten van data, kan de hostprocessor op twee manieren te weten komen wanneer de actie voltooid is. Ofwel doet de CPU aan polling, ofwel laat het randapparaat aan de hand van een interrupt zelf weten aan de processor dat het klaar is.

In deze thesis worden er blokken data overgedragen van het FPGA-bord naar het RAM-geheugen van de computer. Om de host CPU te laten weten dat een dataoverdracht afgelopen is, genereert de FPGA een interrupt.

De klassieke manier om een interrupt te genereren, bestaat erin een bitlijn even hoog te brengen. Deze bitlijn is een rechtstreekse verbinding van het apparaat met een interrupt controller. Voor elke interrupt die kan binnengaan in de interrupt controller, is er dus een aparte pin voorzien. Dit betekent dat het aantal apparaten dat interrupts kan genereren fysisch beperkt is door het aantal pinnen aan de interrupt controller. Het principe van de Message Signaled Interrupt (MSI) biedt hiervoor een oplossing.

In tegenstelling tot normale interrupts kunnen MSI's als virtuele interrupts worden aanzien: het zijn eigenlijk boodschappen die naar een geheugenlocatie worden geschreven. Dit geheugenadres wordt in de gaten gehouden door de interrupt controller, en als er een vastgelegde waarde op het adres geschreven wordt, genereert de controller een interrupt voor de CPU. Op basis van de waarde die naar het geheugen geschreven werd, kan de oorsprong van de interrupt achterhaald worden.

Het is duidelijk dat het aantal benodigde fysische interruptlijnen drastisch daalt bij gebruik van MSI's, aangezien de werking steunt op het schrijven van een zekere waarde naar een bepaald geheugenadres. Oudere PCIe-kaarten hadden tot 4 interruptlijnen ter beschikking, die bovendien allemaal gedeeld moesten worden door alle apparaten op de PCIe-bus.

Hoewel de PCIe-implementatie op het FPGA-bord klassieke interrupts ondersteunt, maakt onze implementatie gebruik van MSI's. Door dit te doen, kunnen er tot 32 verschillende interrupts gegenereerd worden. Met "verschillend" wordt hier bedoeld dat er 32 componenten kunnen zijn met elk hun eigen interrupt. In deze implementatie wordt er slechts 1 interrupt gebruikt, maar uitbreiden naar meerdere MSI's is geen grote stap. De keuze voor MSI's is dus gemaakt eerder met het oog op eventuele uitbreiding dan uit gemakzucht.

5.4 Detectie en configuratie van een PCIe-kaart

Elke PCIe-kaart presenteert zichzelf aan de computer door middel van een informatiestructuur, de *Configuration Space*. In deze datastructuur is alle informatie terug te vinden die nodig is om te kunnen communiceren met het apparaat. Tabel 5.1 geeft de informatie weer die vervat

zit in de configuration space. De parameters die gebruikt werden bij het uitvoeren van deze thesis worden *vet-cursief* weergegeven.

Offset (bytes)	Parameter (+0 bytes)	Parameter (+2 bytes)
0x00	<i>Vendor ID</i>	<i>Device ID</i>
0x04	<i>Command Register</i>	Status register
0x08	Class Code + Revision ID	
0x0C	Cache Line + Latency Timer	Header Type + BIST
0x10	<i>BAR 0</i>	
0x14	<i>BAR 1</i>	
0x18	<i>BAR 2</i>	
0x1C	<i>BAR 3</i>	
0x20	<i>BAR 4</i>	
0x24	<i>BAR 5</i>	
0x28	CardBus CIS pointer	
0x2C	Subsystem Vendor ID	Subsystem Device ID
0x30	Expansion ROM BAR	
0x34	Reserved	
0x38	Reserved	
0x3C	IRQ Line + IRQ Pin	Min. granularity + Max. latency

Tabel 5.1: Configuration space van een PCIe-kaart [35].

Elk moederbord dat PCIe ondersteunt, bevat firmware in het BIOS die configuratie van en communicatie met elk apparaat mogelijk maakt. Hetgeen de CPU te zien krijgt van de PCIe-architectuur, is het RC. Door commando's te sturen naar deze controller is er interactie mogelijk tussen de CPU en de PCIe-kaarten.

Bij het opstarten van de computer enumereert het OS de PCIe-bus, en komt het te weten welke kaarten er allemaal aanwezig zijn. Elke kaart krijgt een bepaald adres toegekend. Vanaf dan kan de configuration space uitgelezen en beschreven worden door van dat adres (+ eventuele offset) te lezen, of ernaar te schrijven.

De eerste twee velden in de configuration space, *Vendor ID* en *Device ID*, vertellen welk apparaat er aanwezig is. Zo staat een Vendor ID van 0x10EE voor *Xilinx*. Samen met een Device ID van 0x0007 weet het besturingssysteem dat er zich een *Virtex V FPGA* van *Xilinx* op dat adres bevindt.

Het derde veld, *Command Register*, wordt door het besturingssysteem gebruikt om de PCIe-kaart te besturen. Zo kan het OS het apparaat al dan niet toestemming geven busmaster te spelen. Dit laatste is nodig als er data moet worden overgedragen van het apparaat naar het RAM-geheugen van de PC. De structuur van het commandoregister is afhankelijk van de fabrikant van het PCIe-apparaat. Tabel 4-6 in de handleiding van het Xilinx PCIe-blok geeft een overzicht van de verschillende bits van dit register [56].

Een PCIe-kaart kan voorzien zijn van een eigen soort geheugen. Nemen we als voorbeeld een

videokaart die voorziet in een eigen blok grafisch geheugen. Om beelden te kunnen weergeven, moet de hostcomputer naar dit geheugen kunnen schrijven. Een PCIe-apparaat laat aan de hostcomputer weten dat het apparaat voorziet in geheugen door middel van zogenaamde BARs. Elk BAR geeft informatie door over één blok adresseerbaar geheugen op de PCIe-kaart. Volgens de PCIe-standaard kunnen tot 6 van deze BARs gedefinieerd worden, maar het FPGA-bord is beperkt tot 3.

Bij het opstarten van de hostcomputer vraagt het BIOS aan elk PCIe-apparaat over hoeveel adresseerbare geheugenzones, dus over hoeveel BARs, het beschikt. Het BIOS vraagt voor elk BAR aan het PCIe-apparaat hoe groot het adresseerbare geheugen is en reserveert een stuk van het fysieke RAM-geheugen in de hostcomputer met dezelfde grootte. Dit stuk RAM-geheugen is niet bruikbaar als normaal RAM-geheugen voor het OS, maar wordt *gemapt* op het geheugen van de PCIe-kaart. Dit wil zeggen dat een schrijfoperatie naar het gemapte RAM-geheugen eigenlijk een schrijfoperatie naar het geheugen op de PCIe-kaart is.

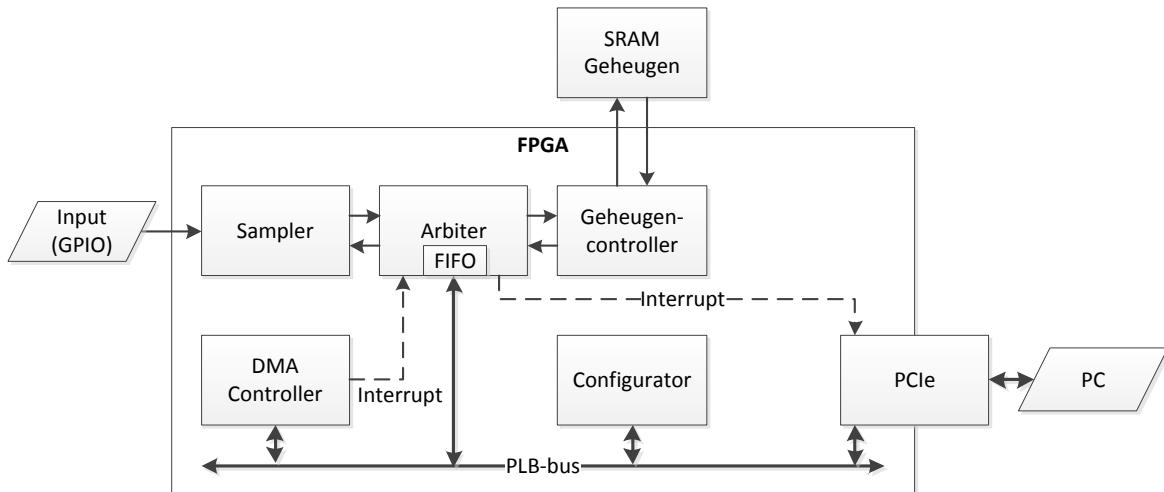
Hoofdstuk 6

FPGA-implementatie

Dit hoofdstuk bespreekt het ontwikkelde FPGA-ontwerp. We starten met een schematisch overzicht; vervolgens bespreken we elke component in meer detail.

6.1 Schema

Figuur 6.1 geeft de geïmplementeerde blokken op de FPGA weer. We onderscheiden een sampler, een arbiter, een geheugencontroller, het off-chip SRAM-geheugen, een DMA-controller, een configurator en een PCIe-core.



Figuur 6.1: Blokschema van de FPGA-implementatie.

De samples die gegenereerd worden door de sampler, worden opgeslagen in een snel geheugen (SRAM). De geheugenchip moet aangestuurd worden d.m.v. een bepaald protocol, dat wordt gecontroleerd door de geheugencontroller.

Samples die opgeslagen worden in het geheugen, moet op een bepaald moment terug uitgelezen worden om ze naar de PC te zenden. Om dit mogelijk te maken, werd een arbiter geïmplementeerd. Op momenten waarop relatief weinig samples gegenereerd worden, leest de arbiter

het geheugen uit, en vult het geleidelijk aan een FIFO-buffer op. Op het moment dat de FIFO een bepaald aantal samples bevat, genereert de arbiter een interrupt om de PC te laten weten dat er data klaar staat.

Om de samples op een efficiënte manier te kunnen overzetten naar de host-PC, maken we gebruik van DMA en een DMA-controller. Deze leest een aantal samples uit de FIFO-buffer van de arbiter, en schrijft deze data vervolgens naar het RAM-geheugen van de PC.

Een PCIe-module verzorgt het PCIe-bus protocol. Deze component bestaat in essentie uit een controller en een brug. De controller genereert de nodige pakketten om te communiceren met de host, en de brug vertaalt adressen van de ene kant (FPGA) naar de andere kant (PC).

De laatste component, de Configurator, staat in voor het configureren van de PCIe-controller bij het initialiseren van de FPGA, en maakt een verbinding met de PC mogelijk.

De onderste vier modules op Figuur 6.1 zijn allemaal met elkaar verbonden met een PLB. Belangrijk op te merken is dat ook de arbiter eraan hangt, en dit om de sampler te kunnen controleren (in- of uitschakelen) en statusinformatie (buffer overrun, geheugen vol, ...) door te kunnen geven van de sampler naar de host-computer. De PLB-bus die gebruikt werd voor deze thesis, heeft een adresruimte van 4 GB [57]. Elk apparaat dat aan deze bus hangt, kan een stuk van die adresruimte toegekend krijgen. Wanneer we met de grafische interface van Xilinx een component toevoegen aan ons ontwerp, wordt er automatisch een stuk van de geheugenruimte toegekend. Deze standaardwaarden volstaan om een werkend systeem te krijgen. Tabel 6.1 geeft een overzicht van deze opdeling.

Component	Begin	Einde	Grootte
DMA-controller	0x80200000	0x8020FFFF	64 KB
PCIe	0x85C00000	0x85C0FFFF	64 KB
PCIe-venster	0xA0000000	0xBFFFFFFF	512 MB
Arbiter	0xCEE00000	0xCEE0FFFF	64 KB
Configurator	0xCD000000	0xCD00FFFF	64 KB

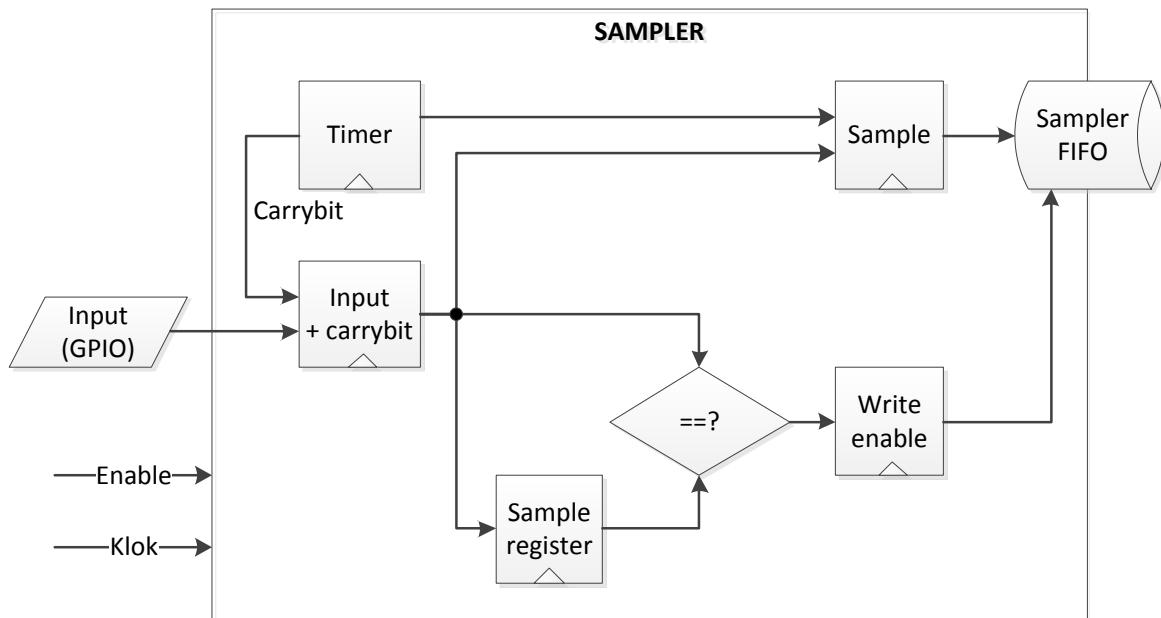
Tabel 6.1: Adresruimte van de PLB-bus.

Het PCIe-venster geeft de FPGA toegang tot een stuk van het RAM-geheugen van de hostcomputer. Als we in de FPGA bijvoorbeeld een waarde schrijven naar het adres `0xA0001000`, dan wordt dit adres door de PCIe-vertaalbrug vertaald naar een geheugenadres aan de hostkant, in dit geval `0x00001000`. Dit PCIe-venster maakt het met andere woorden mogelijk voor de DMA-controller om data te schrijven naar het RAM-geheugen van de hostcomputer.

6.2 Samplen

6.2.1 Top-level ontwerp

In Sectie 3.2.1 hebben we de verschillende soorten samplers besproken. Daar bleek een gebeurtenis gebaseerde sampler het beste geschikt te zijn voor deze thesis. Een eenvoudig blokschema van dit soort sampler wordt weergegeven in Figuur 6.2.



Figuur 6.2: Blokschema van de sampler.

Een gebeurtenis gebaseerde sampler genereert enkel data wanneer haar ingangen veranderen. Deze veranderingen kunnen op eender welk tijdstip gebeuren, waardoor er geen tijdsinformatie meer vervat zit in de geobserveerde ingangssignalen. Dit wordt opgelost door een tijdsstempel toe te voegen aan elke observatie.

De tijdsstempel wordt gegenereerd door een interne timer die telt op elke stijgende klokflank. Door het invoeren van een timer introduceren we echter een bijkomend probleem: de timer beschikt slechts over een eindig aantal bits, waardoor deze vroeg of laat zal overlopen. Om deze fout op te vangen, wordt de carry bit van de timer beschouwd als een extra ingang van het gehele systeem.

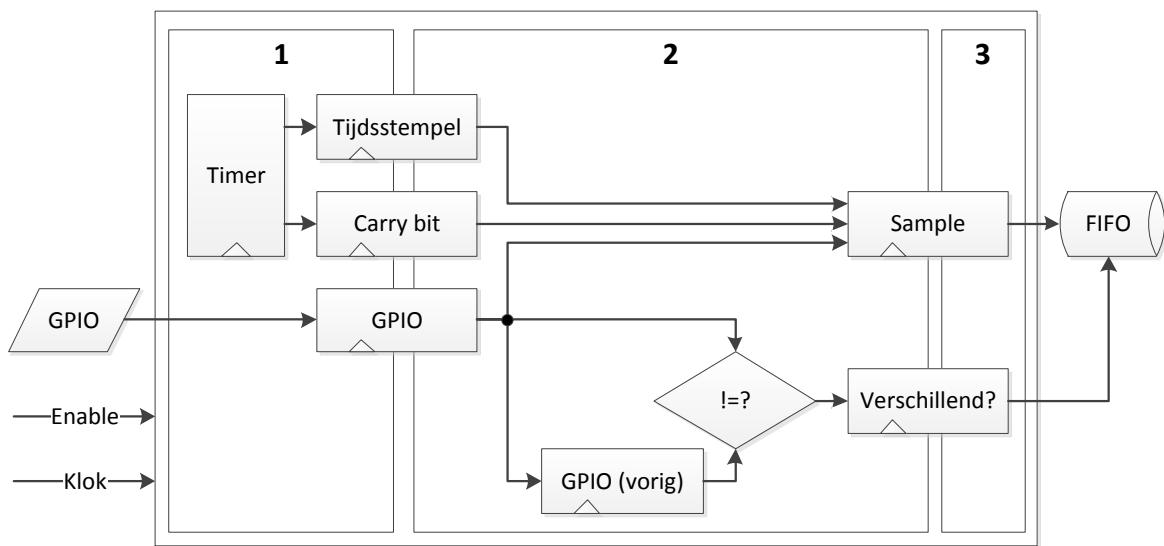
De data die vervat zit in een sample bestaat uit de geobserveerde GPIO-signalen en de carry bit van de timer. Wanneer deze data verschilt met de geobserveerde data op de vorige stijgflank van de klok, wordt een nieuw sample opgeslagen in de FIFO van de sampler. Dit sample is een concatenatie van de data en een tijdsstempel.

6.2.2 FPGA-ontwerp

Het ontwerp van de sampler is gebaseerd op de component die ontworpen werd in de thesis van één van mijn begeleiders, Niels Penneman [44]. Om de samplerresolutie, en dus de maximale samplefrequentie, zo hoog mogelijk te krijgen, werd er een pijplijn geïmplementeerd die uit drie trappen bestaat: observatie, verwerking en opslag. Tabel 6.2 geeft een overzicht van de in- en uitgangssignalen, Figuur 6.3 geeft een grafische voorstelling van de implementatie, waarbij de getallen 1, 2 en 3 de pijplijntrappen voorstellen.

Een sample heeft een vaste bitlayout, die voor een stuk instelbaar is door de VHDL-constanten `DATA_WIDTH` en `TIMESTAMP_WIDTH`. Zoals de benaming al doet vermoeden, stelt de eerste waarde

Signaalnaam	I/O	Breedte	Beschrijving
clk	I	1	Samplerklok
rst	I	1	Sampler reset
data	I	DATA_WIDTH-1	Observatielijnen (GPIO)
enable	I	1	In- of uitschakelen van de sampler
fifo_full	I	1	FIFO-status: (niet) vol
out_write	O	1	FIFO Write Enable
out_data	O	32	FIFO Data Input

Tabel 6.2: Overzicht van de in- en uitgangssignalen van de geïmplementeerde sampler.**Figuur 6.3:** FPGA-ontwerp van de sampler.

het aantal datalijnen in (inclusief een carry bit), en de tweede de bitbreedte van de tijdsstempel. Wij hebben gekozen voor 9 databits (8 GPIO-lijnen en een carrybit) en 23 tijdsbits. De grafische voorstelling van een sample is te zien in Figuur 6.4.

De som van beide variabelen steeds 32 moet zijn. Dit komt omdat de rest van het FPGA-ontwerp ervan uit gaat dat een sample 32 bits breed is. Eventueel kan er overgegaan worden naar bredere samples, mits de nodige aanpassingen doorgevoerd worden in de rest van het ontwerp.

**Figuur 6.4:** Bitlayout van een sample.

De VHDL-implementatie van de sampler bestaat uit een aantal processen. De volgende alinea's bespreken de functionaliteit van deze processen.

De **interne timer** wordt gecontroleerd door een eerste proces. Wanneer de sampler is ingeschakeld (`enable=1`), verhoogt een interne variabele met 1. Wanneer die variabele haar maximale waarde bereikt (d.i. $2^{TICK_WIDTH} - 1$), wordt de carrybit voor 1 klokperiode hoog gebracht.

De **eerste stap** van de datapijplijn voert **data-acquisitie** uit. Hiermee bedoelen we dat de geobserveerde GPIO-signalen in een register worden opgeslagen. Naast de ingangssignalen worden eveneens de carrybit en de tijdsstempel in een pijplijnregister opgeslagen.

De **tweede stap** van de datapijplijn **vergelijkt** het huidige sample met het vorige. Een sample wordt gecreëerd uit de data ontvangen van de eerste trap: GPIO-ingangen, carry bit en tijdsstempel worden geconcateneerd. Verder wordt er een signaal aangestuurd dat aan de volgende trap laat weten of het huidige sample moet opgeslagen worden (d.i. de huidige geobserveerde data is verschillend van de vorige). Tenslotte worden de GPIO-ingangen uit de eerste trap nogmaals opgeslagen, dit keer apart, om ze op de volgende stijgflank te kunnen vergelijken met de observaties op dat moment.

De **derde stap** van de datapijplijn staat in voor de **opslag** van de gegenereerde samples in een FIFO-buffer. Enkel wanneer de vergelijking in de vorige trap aantoont dat de laatst geobserveerde data verschillend is van de voorlaatste data, wordt het nieuwe sample naar de FIFO geschreven. Bovendien voeren we deze schrijfoperatie enkel uit wanneer de FIFO nog niet vol is.

6.3 SRAM-controller

6.3.1 Top-level ontwerp

In Sectie 3.2.2 werden de verschillende mogelijkheden besproken om samples tijdelijk op het FPGA-bord op te slaan. Het SRAM-geheugen kwam als beste optie uit de bus, gezien de lage implementatiecomplexiteit en de hoge werkfrequentie. De geheugenchip die aanwezig is op het FPGA-bord, is bovendien een Zero-Bus Turnaround (ZBT)-chip. Dit betekent dat er tussen een lees- en een schrijfoperatie van en naar het SRAM-geheugen geen klokcycli verloren gaan.

De geheugenchip moet volgens een bepaald protocol aangestuurd worden. Hiervoor hebben we twee opties: ofwel maken we gebruik van een geheugencontroller geleverd door Xilinx, ofwel schrijven we er zelf één. Kiezen we voor de Xilinx geheugencontroller, dan zijn we verplicht een busprotocol te gebruiken. Om volledige controle te hebben over de stuursignalen en omdat een SRAM-geheugen helemaal niet moeilijk aan te sturen is, kiezen we ervoor zelf een controller te schrijven.

Een belangrijk detail is dat de geheugenchip gebruik maakt van een bidirectionele databus: zowel de te schrijven als de gelezen data wordt over deze bus getransporteerd. De implementatie van deze bidirectionele bus leverde wat problemen op. Sectie 6.3.3 bespreekt mijn oplossing.

6.3.2 FPGA-ontwerp

Uit de gebruikershandleiding van het FPGA-bord weten we dat de volgende geheugenchip aanwezig is: ISSI IS61NLP25636A-200TQL [60]. Dit geheugen heeft een grootte van 1 MB met een adresbus van 18 bits, en een databus van 36 bits. Deze laatste bestaat uit 32 databits data en 4 bits optionele paritybits. Wij maken enkel gebruik van de databits.

Het SRAM-geheugen kan, indien gewenst, in burst mode aangesproken worden. Burst mode betekent dat het aangelegde adres binnenin de geheugenchip aangepast wordt om op de volgende kloktik al het volgende datawoord te ontvangen of naar buiten te sturen. De interne teller van de geheugenchip bestaat echter slechts uit 2 bits bestaat, waardoor er in burst mode maar 4 opeenvolgende adressen aangesproken kunnen worden.

In de datasheet zijn, naast de stuursignalen, timingdiagrammen terug te vinden voor de lees- en schrijfoperaties, al dan niet in burstmode [37]. Vermits het SRAM-geheugen in verschillende packages geproduceerd wordt, is enige voorzichtigheid geboden bij het bepalen van de beschikbare pinnen. Uit de *Bill of Materials* van het FPGA-bord waarmee ik werkte is af te leiden dat de gebruikte package “JEDEC 100-pin TQFP” is [53].

Om de geheugencontroller nog te kunnen gebruiken in eventuele andere implementaties is deze zo generiek mogelijk ontworpen. Zo is een signaal voorzien om de burst mode in te schakelen, ook al wordt dit niet gebruikt in de implementatie voor deze thesis. In de volgende secties wordt eerst de interactie met de arbiter besproken, daarna de interactie met de chip zelf. Tenslotte volgt nog een woordje uitleg bij de controller.

Interface met de arbiter

De signalen die betrokken zijn bij de communicatie tussen de geheugencontroller en de arbiter worden weergegeven in Tabel 6.3.

Signaalnaam	I/O	Breedte	Beschrijving
clk	I	1	Klok
rst	I	1	Reset
rstinv	I	1	Inverse reset
address	I	18	(start)adres van operatie
cmd_read	I	1	Gewenste operatie: lezen?
cmd_write	I	1	Gewenste operatie: schrijven?
cmd_burst	I	1	Operatie uitvoeren in burst?
data_in	I	32	Weg te schrijven data
data_out	O	32	Uitgelezen data
data_ready	O	1	Hoog als data op <code>data_out</code> geldig is.

Tabel 6.3: Overzicht van de in- en uitgangssignalen van de SRAM-controller: interface met de arbiter.

De **klok** die hier wordt gedefinieerd, is het kloksignaal van de *controller*, niet dat van de geheugenchip. De controller wordt aangestuurd met hetzelfde kloksignaal als de arbiter. De klok die de geheugenchip toegevoerd krijgt, wordt gegenereerd door een aparte klokgenerator.

Wat opvalt, is de aanwezigheid van **twee resetsignalen**: een normale en een geïnverteerde versie. Het nut van deze signalen wordt uitgelegd in Sectie 6.7.

Voor elke operatie moet de SRAM-chip een **adres** toegevoerd krijgen via een 18-bit bus.

Verder worden er een aantal **controlesignalen** gedefinieerd. We kozen hier voor aparte stuurlijnen om te lezen en te schrijven, om zo ook een rustoperatie te kunnen invoeren in het geval er geen data geschreven of gelezen moet worden. Het burstsignaal wordt in deze implementatie niet gebruikt omdat de arbiter zelf voldoende snel adressen kan genereren. De implementatie is tenslotte in beperkte mate beschermd tegen verkeerde aansturing: wanneer zowel `cmd_read` als `cmd_write` hoog zijn, wordt er geen opdracht gegeven aan de SRAM-chip.

De weg te schrijven data wordt op de **ingangsbus** geplaatst, uitgelezen data komt terecht op de **uitgangsbus**. De arbiter weet dat er geldige data op de uitgangsbus staat als het **ready-signaal** hoog is.

Interface met de SRAM-chip

De SRAM-chip voorziet in meer signalen dan dat de FPGA nodig heeft om de chip te besturen. We overlopen eerst de signalen die door het FPGA-bord op een constante waarde gehouden worden. Daarna bespreken we de signalen die wel aangestuurd kunnen worden, maar die wij niet gebruiken. Tenslotte kijken we naar de signalen die effectief gebruikt worden door de geheugencoontroller.

Eerst en vooral zijn er een aantal pinnen gedefinieerd in de datasheet van de SRAM-chip die op het FPGA-bord ofwel permanent met de voedingsspanning verbonden zijn, ofwel permanent aan massa hangen [54]:

- \overline{CKE} : permanent laag - klok ingeschakeld;
- $\overline{CE2}$: permanent laag - chip ingeschakeld als `ctrl2sram_ce` laag is;
- $CE2$: permanent hoog - chip ingeschakeld als `ctrl2sram_ce` laag is;
- ZZ : permanent laag - chip kan niet in stand-by modus gebracht worden.

Vervolgens zijn er 5 signalen die gedefinieerd zijn in de datasheet, maar niet gebruikt worden in onze VHDL-implementatie:

- **MODE** - Deze lijn wordt via een pulldown weerstand laag getrokken. Een lage waarde betekent dat de interne burststeller lineair incrementeert. Als **MODE** daarentegen hoog is, wordt er interleaving toegepast op de opeenvolgende burst adressen. De laatst vermelde functionaliteit is echter niet nuttig voor deze thesis, en we kozen dan ook voor een permanent lineaire burst mode door **MODE** onaangeroerd te laten.
- **DQP[a-d]** - Dit zijn 4 parity bits (1 per databyte) om eventueel foutdetectie te doen. Om de complexiteit van de implementatie te beperken, maken we geen gebruik van deze bits.

Signaalnaam	I/O	Breedte	Beschrijving
<code>ctrl2sram_a</code>	O	18	Adresbus
<code>ctrl2sram_we</code>	O	1	Operatie (lezen of schrijven)
<code>ctrl2sram_ce</code>	O	1	Chip Enable (CE) (actief-laag)
<code>ctrl2sram_bw[a-d]</code>	O	1	Schrijf data byte (Byte Enable)?
<code>ctrl2sram_oe</code>	O	1	Output Enable
<code>sram_dq[a-d]_I</code>	I	8	Tristate bus signaal: uitgelezen data
<code>sram_dq[a-d]_O</code>	O	8	Tristate bus signaal: weg te schrijven data
<code>sram_dq[a-d]_T</code>	O	1	Tristate bus signaal: tristate signaal

Tabel 6.4: Overzicht van de in- en uitgangssignalen van de SRAM-controller: interface met de geheugenchip.

Tabel 6.4 geeft de signalen weer die betrokken zijn bij de communicatie tussen de geheugencooller en de geheugenchip.

De adresbus definiëren we aan de hand van het signaal `ctrl2sram_a`.

Of we willen lezen van, dan wel schrijven naar het geheugen wordt bepaald door de toestand van het `ctrl2sram_we`-signaal. Een lage waarde betekent schrijven, een hoge waarde betekent lezen.

Veel chips die gebruikt worden in digitale toepassingen delen eenzelfde data- en/of controlebus. De bus van het SRAM-geheugen wordt op het FPGA-bord gedeeld met die van het Flash configuratiegeheugen. Om de controller toe te laten te communiceren met een specifieke chip, wordt er in een CE-signaal voorzien. In ons geval moet het `ctrl2sram_ce` laag gebracht worden om te communiceren met het SRAM-geheugen.

Hoewel het SRAM-geheugen werkt met 32-bit datawoorden, is de data gegroepeerd per 8 bits. Bij het schrijven van een 32-bit waarde naar het geheugen, kunnen de te schrijven bytes geselecteerd worden door de signalen `ctrl2sram_bw[a-d]` gepast aan te sturen. In deze implementatie wordt er niet met afzonderlijke bytes gewerkt, zodat deze signalen op een constante waarde gehouden kunnen worden.

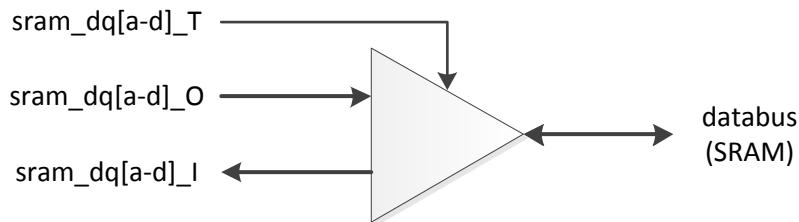
Het Output Enable (OE)-signaal moet volgens de waarheidstabell in de datasheet enkel hoog zijn bij het geven van dummy leesopdrachten [37]. Gezien er geen “Dummy Reads” worden uitgevoerd, is het voldoende `ctrl2sram_oe` laag te houden.

De laatste drie signalen uit Tabel 6.4 horen in principe bij elkaar. Het is immers de combinatie van deze signalen die het implementeren van een bidirectionele bus mogelijk maakt. In theorie biedt VHDL de mogelijkheid dit te doen door signalen van het type `INOUT` te definiëren, maar de Xilinx toolchain kan hier niet goed mee overweg. We werken met deze signalen om een tristate buffer af te dwingen van de toolchain.

De manier waarop deze signalen gedefinieerd worden, zowel het type als de naam, ligt vast [64]. In dit geval, waar de bidirectionele bus de naam `sram_dq[a-d]` draagt, zijn de volgende signalen betrokken bij het implementeren van een tristate buffer:

- **sram_dq[a-d]_T** - Het stuursignaal van de buffer: de toestand van dit signaal bepaalt of er data gelezen, dan wel geschreven wordt van of naar de bidirectionele bus.
- **sram_dq[a-d]_I** - Ingangsbus van de component: hierop komt de uitgelezen data van het SRAM-geheugen terecht.
- **sram_dq[a-d]_O** - Uitgangsbus van de component: hierop komt de weg te schrijven data naar het SRAM-geheugen terecht.

De interactie tussen deze signalen wordt duidelijk in Figuur 6.5. Meer informatie in verband met de implementatie van de tristate buffer is te vinden in Sectie 6.3.3.



Figuur 6.5: Tristate buffer om een bidirectionele bus te implementeren.

De controller

De controllerimplementatie bestaat uit de volgende processen:

1. een proces om de tristate buffers aan te sturen en de bidirectionele bus te schrijven en te lezen;
2. een proces om een signaal te geven aan de arbiter dat er geldige data op de uitgangsbus **data_out** staat;
3. een proces om de weg te schrijven data op de bidirectionele bus te plaatsen wanneer het schrijfcommando gegeven werd;
4. een proces om de stuursignalen in te stellen.

Het eerste proces stuurt de tristate buffer van de bidirectionele bus aan. In eerste instantie wordt het zgn. tristate signaal aangestuurd: laag om data naar de bus te schrijven, hoog om data van de bus te lezen [64]. In dit proces wordt de databus eveneens geschreven of gelezen, naargelang de operatie.

Na het geven van een leesopdracht is de gelezen data niet meteen beschikbaar op de databus. De chip voert deze data slechts de tweede stijgflank na het geven van de leesopdracht naar buiten. Om de implementatie eenvoudiger te maken, wordt de ingang van een schuifregister, geïmplementeerd in het tweede proces, voor één klokperiode hoog gebracht. Het **data_ready**-signaal wordt hoog wanneer een waarde uit het SRAM-geheugen uitgelezen werd.

Voor het schrijven van data naar het SRAM-geheugen moet eerst een schrijfcommando gegeven worden, en pas de kloktik erna moet de data aangelegd worden. Om dit te doen, implementeert ook het derde proces een schuifregister.

Tenslotte gebeurt de aansturing van de chip in het vierde proces. Het eerste wat dit proces doet, is de drie commandosignalen `cmd_read`, `cmd_write` en `cmd_burst` samenbrengen in één variabele. Hierdoor kan een toestandsmachine gedefinieerd worden met deze geconcateneerde waarde als toestand. De tweede stap houdt in dat de stuursignalen op basis van de zonet gedefinieerde toestand aangestuurd worden. Het aansturen van deze signalen gebeurt volgens de waarheidstabell uit de datasheet van de SRAM-chip [37].

Het verschil tussen onze eigen controller met een IP core van Xilinx, is hoogstwaarschijnlijk minimaal. De Xilinx-core werd niet getest, omdat de aansturing ervan toch net iets anders is dan die van onze eigen component. Het voordeel dat we hebben met onze eigen controller is dat we zelf volledige controle hebben over de stuursignalen van het SRAM-geheugen.

6.3.3 Implementatie van een bidirectionele bus

Een eerste implementatie van de geheugencoontroller leverde een lastige foutmelding op bij het synthetiseren van de VHDL-code:

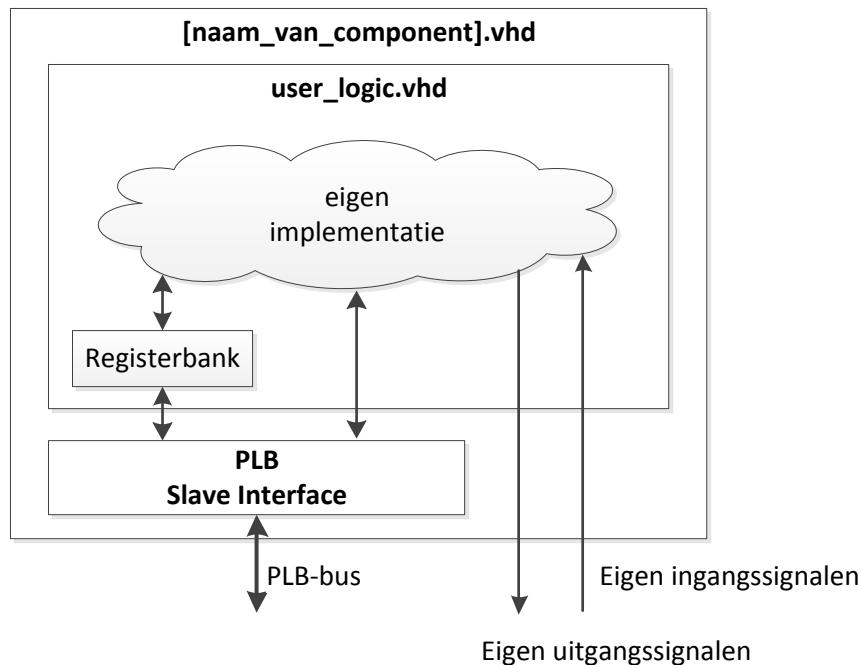
ERROR:MapLib:824 - Tri-state buffers are not supported in Virtex5. [...]

De oorzaak van deze fout is terug te vinden in het gebruik van VHDL-signalen van het type INOUT, die gebruikt worden om de bidirectionele bus van het SRAM-geheugen te definiëren. Op zich kan de Xilinx toolkit dit type signalen wel aan, maar enkel op het hoogste VHDL-niveau. We gaan nu eerst dieper in op wat bedoeld wordt met “het hoogte VHDL-niveau”, waarna we de geïmplementeerde oplossing bespreken.

De grafische interface van de Xilinx toolchain voorziet in een wizard om snel en eenvoudig templates te genereren voor eigen VHDL-componenten die uiteindelijk met IP cores van Xilinx moeten kunnen communiceren. De template die deze wizard genereert, bestaat uit de volgende bestanden:

- `[naam_van_component].vh` - Hierin wordt de interface gedefinieerd die zichtbaar is voor de toolchain. Dingen zoals een controller om te communiceren met de PLB-bus worden hier geïnstancieerd. In dit bestand wordt eveneens een instantie gemaakt van het object dat zich in `user_logic.vhd` bevindt.
- `user_logic.vhd` - Dit bestand bevat de eigenlijke implementatie van de component. Naast de eigen geschreven component bevat deze template ook logica om een eventuele registerbank aan te sturen.

Figuur 6.6 geeft de structuur weer van een zogenaamde *custom peripheral* die met deze wizard gegenereerd werd. Hierop is de hiërarchie van de bestanden zichtbaar: het hoofdbestand, `[naam_van_component].vh`, instantieert een PLB interface en maakt een instantie aan van de geïmplementeerde logica in `user_logic.vhd`. De eigen implementatie in `user_logic.vhd` hoeft niet alle signalen over de PLB-bus te sturen. Er kunnen evengoed eigen signalen, poorten genoemd, gedefinieerd worden zodat andere bouwbllokken zonder tussenkomst van de PLB-bus aan de component gekoppeld kunnen worden.

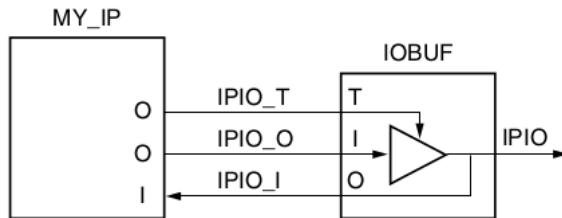


Figuur 6.6: Structuur van een custom peripheral.

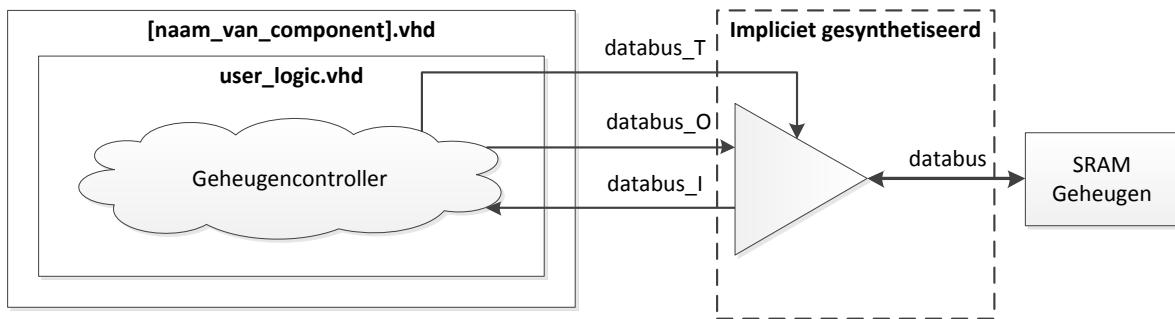
De wolk op de figuur met “eigen implementatie” erin kan in deze context opgevat worden als de geheugencontroller. Het feit dat het SRAM-geheugen één enkele bus gebruikt voor de in- en output van data, zorgt ervoor dat deze geheugencontroller een signaal van het type INOUT moet definiëren. Laat dit nu net een probleem zijn voor de toolchain. Bidirectionele signalen kunnen enkel d.m.v. INOUT gedefinieerd worden op het hoogste niveau van de component, d.w.z. in het bestand `[naam_van_component].vhdl`.

Documentatie in verband met het definiëren van alle soorten poorten, dus ook die van het type INOUT, is terug te vinden in de Xilinx Platform Specification Format (PSF) [64]. Het type poort dat ons aanbelangt heet “tristate buffer” en wordt besproken vanaf p.67.

Figuur 3-1 van de Xilinx PSF, hier te zien in Figuur 6.7, maakt duidelijk op welke manier het toch mogelijk is een bidirectioneel signaal aan te maken op het niveau van `user_logic.vhd` [64]. In plaats van één INOUT signaal, dat we voorlopig “databus” noemen, worden er 2 uitgangen (`databus_T` en `databus_0`) en 1 ingang (`databus_I`) gedefinieerd. Het verband tussen de signalen en de implementatie is te zien in Figuur 6.8. Het doel van deze tristate buffer is het scheiden van in- en uitgangsdata. Hetgeen binnen het omstippelde kader staat wordt impliciet gesynthetiseerd door de gepaste parameters mee te geven aan de toolchain.



Figuur 6.7: Suggestie van Xilinx om een tristate buffer te synthetiseren.



Figuur 6.8: Implementatie van een bidirectionele bus.

Het tristate-singaal *databus_T* bepaalt de richting van de bidirectionele bus. Het tristate signaal is actief-laag [64]. Dit betekent dat het uitgangssignaal *databus_O* wordt doorgevoerd naar de bidirectionele bus als het tristate-signaal laag is, en dat de bidirectionele bus uitgelezen kan worden als het hoog is.

Om nu de synthesetool te laten weten dat we van dit koppel signalen een tristate buffer willen maken, passen we het poortdefinitiebestand aan. Dit bestand bevindt zich op de volgende locatie: `[projectmap]/pcores/[naam_van_component]/data/[naam_van_component].mpd`

```
[...]
PORT databus="", DIR=IO, VEC=[0:7], THREE_STATE=TRUE, \
    TRI_I=databus_I, TRI_O=databus_O, TRI_T=databus_T, \
    ENABLE=SINGLE
PORT databus_I="", DIR=I, VEC=[0:7]
PORT databus_O="", DIR=O, VEC=[0:7]
PORT databus_T="", DIR=O
[...]
```

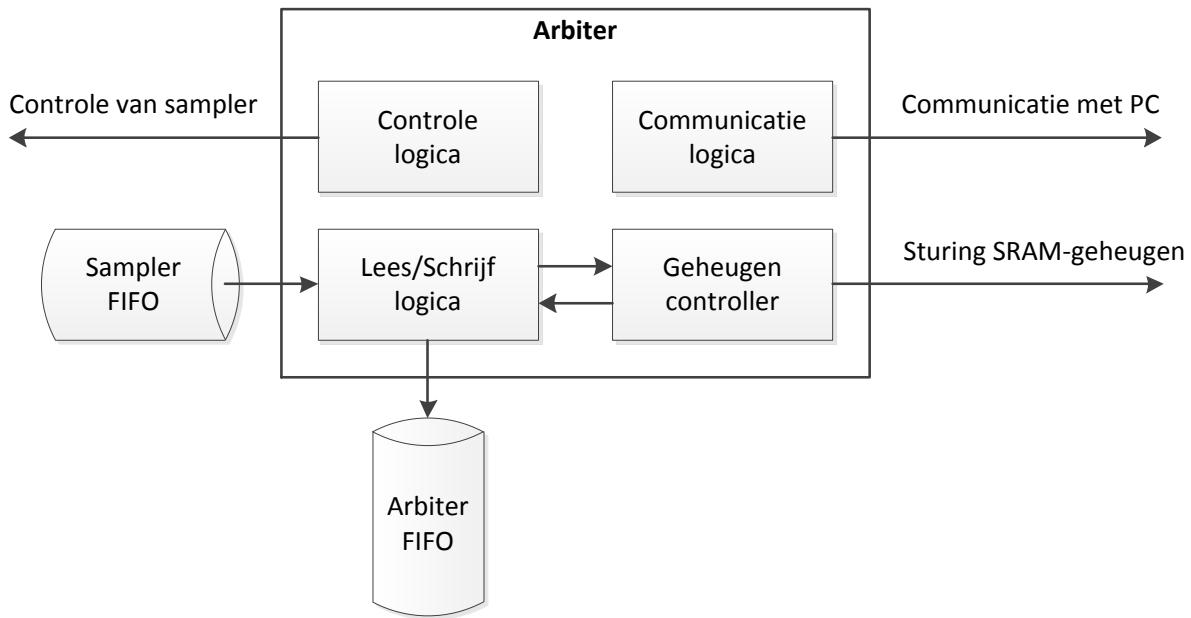
Hierbij werd uitgegaan van een 8-bit bus. De poortparameter `ENABLE` stelt in of elke bit van de bus een eigen tristate signaal moet krijgen (`ENABLE=SINGLE`) of niet (`ENABLE=MULTI`). Door de definitie van de tristate poort (*databus*) laten we de toolchain weten dat het een wrapper moet maken rond de signalen *databus_I*, *databus_O* en *databus_T*.

Na het definiëren van deze poorten, kunnen andere componenten rechtstreeks met de bidirectionele bus verbonden worden. De *_I*-, *_O*- en *_T*-poorten dienen open gelaten te worden.

6.4 Arbiter

6.4.1 Top-level ontwerp

In Figuur 6.9 is de top-level structuur van de arbiter te zien. Deze component moet in essentie drie taken vervullen: beheer van het SRAM-geheugen (lezen/schrijven), controle van de sampler en data-overdracht naar de PC.



Figuur 6.9: Top-level schema van de arbiter.

De eerste taak is het **beheer van het SRAM-geheugen**. De samples worden gelezen uit de FIFO van de sampler, en moeten tijdelijk opgeslagen worden in het SRAM. Wanneer er plaats is in de FIFO van de arbiter, worden samples uit het SRAM gelezen en in deze FIFO geplaatst. Dit alles gebeurt zoals beschreven in Sectie 6.1.

De tweede taak is het **controleren van de sampler**. De arbiter voorziet in een PLB slave interface. Dit betekent dat er vanaf de PLB-bus (en dus ook via PCIe) registers kunnen geschreven en gelezen worden. Enerzijds worden die registers gebruikt om de toestand van de arbiter te weten te komen, anderzijds om de sampler in of uit te schakelen. Er kan eveneens informatie opgevraagd worden over de toestand van de FIFO's.

De derde en laatste taak houdt het **coördineren van de dataoverdracht** in. De arbiter laat de PC weten dat er een bepaald aantal samples in de FIFO van de arbiter geplaatst zijn met een interrupt. De PC programmeert dan de DMA-controller en initieert een dataoverdracht. Wanneer een datablok is overgezet naar het RAM-geheugen van de PC, laat de DMA-controller aan de arbiter weten dat ze klaar is d.m.v. een interrupt. Vervolgens geeft de arbiter dit door aan de PC door opnieuw een interrupt te genereren. De PC voert tenslotte een reset uit van de DMA-controller en het proces kan van vooraf aan beginnen.

6.4.2 FPGA-ontwerp

De arbiter is de centrale component van de FPGA-implementatie die de verschillende submodules met elkaar verbindt en deze controleert. Figuur 6.10 geeft een blokschema weer van de relaties tussen die submodules en de processen die eraan verbonden zijn. Ook de verschillende klokdomen worden op dit schema vermeld. Deze laatste bespreken we op het einde van deze sectie.

Onderdelen

De rechthoeken in Figuur 6.10 geven objectinstanties weer. We geven een kort overzicht van de verschillende blokken.

Sampler Deze component werd al uitvoerig besproken in Sectie 6.2. Vanuit de arbiter is het mogelijk deze in of uit te schakelen.

Geheugencontroller Ook deze component werd al in detail behandeld in Sectie 6.3. De controller vormt een eenvoudige interface met het SRAM-geheugen en kan, indien gewenst, vervangen worden door een andere controller met dezelfde arbiter-interface om een ander type geheugen aan te sturen.

Sampler FIFO Deze FIFO vormt de overgang tussen de sampler en de arbiter. Enerzijds zorgt die voor een bufferwerking wanneer de arbiter tijdelijk geen samples kan schrijven naar het geheugen. Anderzijds is het door gebruik te maken van een tweepoortige FIFO mogelijk te voorzien in een apart klokdomain voor zowel de sampler als de arbiter. Sectie 6.4.2 bespreekt de klokdomen in meer detail.

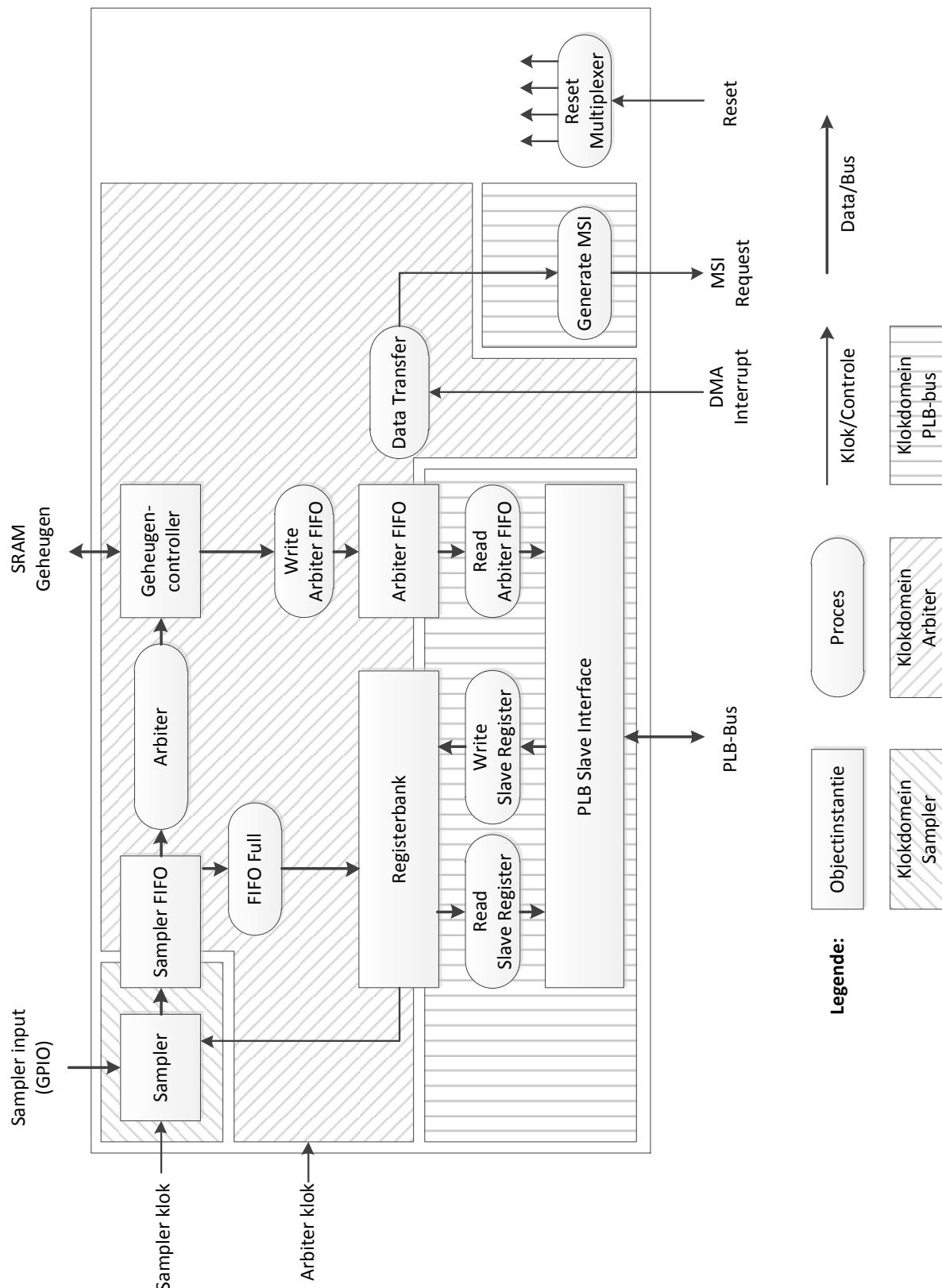
Arbiter FIFO De arbiter FIFO is er om de DMA-controller van een volledig datablok te kunnen voorzien. Deze FIFO wordt op het tempo van de arbiter opgevuld. Wanneer het aantal samples in de FIFO een bepaald niveau bereikt heeft, doet de arbiter een aanvraag bij de PC om de data over te zetten met behulp van DMA.

PLB Slave Interface De arbiter communiceert met de buitenwereld aan de hand van een PLB-bus. Deze interface is een IP core geleverd door Xilinx.

Registerbank De configuratie- en statusvlaggen van de FPGA-implementatie worden opgeslagen in de registers van deze registerbank. In principe is dit geen echte submodule, maar eerder een set van VHDL-signalen. De registers kunnen geschreven en gelezen worden via de PLB-interface.

Processen

De zonet besproken submodules moeten aangestuurd worden. Dit gebeurt door de 9 processen uit Figuur 6.10.



Figuur 6.10: Blokschema van de arbiter.

Reset multiplexering Signalen met een grote fanout verlagen de maximale werkfrequentie van het FPGA-ontwerp. Het resetsignaal is hier een goed voorbeeld van: elk signaal dat conditioneel aangestuurd wordt vanuit het resetsignaal, verhoogt de fanout. Om een te hoge fanout en dus een traag systeem te vermijden, maken we duplicaten van het resetsignaal, waardoor elk dupliaat minder signalen moet aansturen, en waardoor de fanout naar omlaag gaat. Dit wordt gerealiseerd in het proces **Reset Multiplexer**.

Beveiliging tegen verlies van samples Wanneer de FIFO van de sampler vol zit en er een nieuw sample gegenereerd wordt, gaat dit laatste sample verloren. Als de arbiter deze volgelopen FIFO van de sampler uitleest, is het onmogelijk te achterhalen of er samples gegenereerd werden in de periode dat de FIFO vol zat. Daarom is het nodig in een statusvlag te voorzien die weergeeft of de FIFO van de sampler *ooit* is vol gelopen. Deze statusvlag slaan we op in één bit van de registerbank, en kan uitgelezen worden door de hostcomputer. Die weet op zijn beurt dat er mogelijk een periode is waarvoor geen samples opgeslagen werden. De aansturing van deze vlag gebeurt in het proces **Fifo Full**.

Opvullen van de arbiter FIFO Telkens een leesrequest van het SRAM-geheugen is voltooid, slaan we de uitgelezen waarde op met het proces **Write Arbiter FIFO**.

Initiëren van een DMA overdracht Op het moment dat de FIFO van de arbiter een bepaald aantal samples bevat, wordt er een DMA overdracht aangevraagd aan de PC door het proces **Data Transfer**. Naast het initiëren van een overdracht, wordt deze ook gecoördineerd door dit proces. Dat gebeurt met behulp van een toestandsmachine, die later in meer detail wordt uitgelegd. Communicatie met de PC gebeurt aan de hand van interrupts.

Genereren van interrupts Zoals besproken in Sectie 5.3 gebruiken we MSIs om met de PC te communiceren. De PCIe-core heeft een ingangssignaal, `msi_request`, dat het mogelijk maakt een MSI te genereren. Dit signaal moet volgens de handleiding minstens twee klokcycli hoog blijven om een interrupt aan de PC-kant te genereren [59]. Deze timing wordt verzorgd door het proces **Generate MSI**.

Uitlezen van de FIFO van de arbiter Eens de PC weet dat er genoeg data in de FIFO van de arbiter zit om een DMA-blok te vullen, stuurt deze de DMA-controller aan. De DMA-controller leest de arbiter FIFO in burst uit via de PLB-bus. Het proces **Read Arbiter FIFO** handelt deze leesrequests af.

Registerbank voor status en controle Het is belangrijk voor de PC dat deze over enige vorm van informatie beschikt i.v.m. de status van de arbiter: is de FIFO van de sampler ooit vol gelopen, is er een fout gebeurd in de arbiter, debuggen van fouten, ... Statusinformatie kan naar buiten gebracht worden via een statusregister. Het proces **Read Slave Register** reageert op PLB-leesrequests die gericht zijn op de arbiter.

De sampler moet aangestuurd kunnen worden vanuit de PC. Een manier om dit te bereiken, is gebruik maken van een controleregister. Door een bepaalde waarde naar dit register te schrijven, kan de sampler bijvoorbeeld in- of uitgeschakeld worden. Het proces **Write Slave Register** maakt het mogelijk PLB-schrijfrequests naar registers af te handelen.

De geïmplementeerde registerbank van de arbiter is te zien in Tabel 6.5. De eerste kolom geeft de offset van het register t.o.v. het basisadres van de arbiter uit Tabel 6.1, de tweede kolom geeft de naam van het register.

Offset	Beschrijving
+0x00000000	Status en controle
+0x00000004	Arbiter FIFO

Tabel 6.5: Registerbank van de arbiter.

De bitlayout van het controleregister is te zien in Figuur 6.11. Tabel 6.6 geeft een beschrijving van de verschillende bitvelden. Bits 11 en 28 worden niet gebruikt.



Figuur 6.11: Bitlayout van het controleregister van de arbiter.

De bitlayout van het arbiter FIFO register is te zien in Figuur 6.12. We onderscheiden slechts 1 veld, namelijk het uitgelezen sample. Door van dit register te lezen, wordt een nieuw sample klaargezet om uitgelezen te worden bij een volgende leesoperatie. De data kan dus slechts 1 keer uit dit register gelezen worden. Een schrijfrequest naar dit adres wordt genegeerd.



Figuur 6.12: Bitlayout van het FIFO uitgangsregister van de arbiter.

Eén geheugen, twee operaties Ondertussen zou de noodzaak voor enige arbitratie van de geheugentoegangen duidelijk moeten zijn: samples moeten enerzijds kunnen worden opgeslaan in het geheugen, en moeten anderzijds op tijd terug uitgelezen worden om overgebracht te worden naar de PC.

Het proces dat deze geheugentoegang beheert, heet **Arbiter**. Het proces bestaat uit drie stappen:

1. Bepaal of het mogelijk is een sample over te brengen van de FIFO van de sampler naar het geheugen en of het mogelijk is een sample over te brengen van het geheugen naar de FIFO van de arbiter.
2. Bepaal de operatie die de geheugencoontroller moet uitvoeren: lezen, schrijven of niets doen.
3. Stuur de geheugencoontroller aan.

¹L=Bit is leesbaar, S=Bit is schrijfbaar

Bit	Naam	L/S ¹	Beschrijving
0	AF	L	0 = FIFO van de arbiter is niet vol 1 = FIFO van de arbiter is vol
1	AE	L	0 = FIFO van de arbiter is niet leeg 1 = FIFO van de arbiter is leeg
2	OF	L	0 = FIFO van de sampler is minstens 1 keer opgevuld 1 = FIFO van de sampler is nooit opgevuld
3	SF	L	0 = FIFO van de sampler is niet vol 1 = FIFO van de sampler is vol
4	SE	L	0 = FIFO van de sampler is niet leeg 1 = FIFO van de sampler is leeg
5	MF	L	0 = SRAM-geheugen is niet vol 1 = SRAM-geheugen is vol
6	ME	L	0 = SRAM-geheugen is niet leeg 1 = SRAM-geheugen is leeg
8-10	CS	L	Toestand van de communicatie met de PC 0b000 = wacht tot sampler ingeschakeld wordt 0b001 = wacht tot er genoeg data in de FIFO van de arbiter zit 0b010 = wacht op een interrupt van de DMA-controller 0b100 = wacht tot de PC de DMA-controller reset 0b111 = ongeldige toestand
12-27	AFC	L	aantal samples in de FIFO van de arbiter
29	RF	L/S	0 = houd OF vlag vast 1 = wis OF vlag
30	FA	L/S	0 = enkel DMA overdracht wanneer er voldoende data is 1 = forceer een DMA overdracht (flush de FIFO van de arbiter)
31	ES	L/S	0 = sampler is uitgeschakeld 1 = sampler is ingeschakeld

Tabel 6.6: Bitvelden van het controleregister van de arbiter.

Om te bepalen of het mogelijk is een actie te ondernemen, houdt dit proces drie variabelen bij. Een leespointer wijst naar het adres waarvan gelezen zal worden. Een schrijfpointer wijst naar het adres waarnaar geschreven zal worden. Tenslotte is er nog een variabele die het aantal samples bijhoudt dat wel geschreven, maar nog niet uitgelezen werd.

We bepalen de mogelijke operaties als volgt. Als er geldige data aan de uitgang van de FIFO van de sampler staat én het aantal ongelezen samples is kleiner dan het maximum aantal samples dat in het SRAM opgeslagen kan worden, dan is het mogelijk een sample over te dragen van de FIFO van de sampler naar het SRAM-geheugen. Als de FIFO van de arbiter niet vol is én er staan ongelezen samples in het SRAM-geheugen, dan is het mogelijk een sample over te dragen van het SRAM-geheugen naar de FIFO van de arbiter.

Eens het geweten is operaties ondernomen kunnen worden, kan de uit te voeren actie bepaald worden. Wanneer er geen enkele operatie mogelijk is, wordt een rustopdracht ingesteld. Wanneer enkel lezen of enkel schrijven mogelijk is, wordt een leesopdracht, respectievelijk schrijfopdracht ingesteld. Wanneer zowel lezen als schrijven mogelijk is, wordt er rekening gehouden met de vorig uitgevoerde operatie. Was dit een leesoperatie, dan zullen we nu schrijven. Was dit een schrijfoperatie, dan zullen we nu lezen.

Het genereren van interrupts

De arbiter communiceert met de PC aan de hand van interrupts. Om een interrupt via de PCIe-core te genereren, moet het **MSI_request**-signaal gepast aangestuurd worden. Het proces **Generate MSI** verzorgt de juiste timing op deze signaallijn. Aangezien dit proces werkt op de klok van de PLB-bus en interrupts worden gegenereerd vanuit de arbiter (die aangestuurd wordt met een andere klok), dienen we op een gepaste manier van klokk domeinen te wisselen.

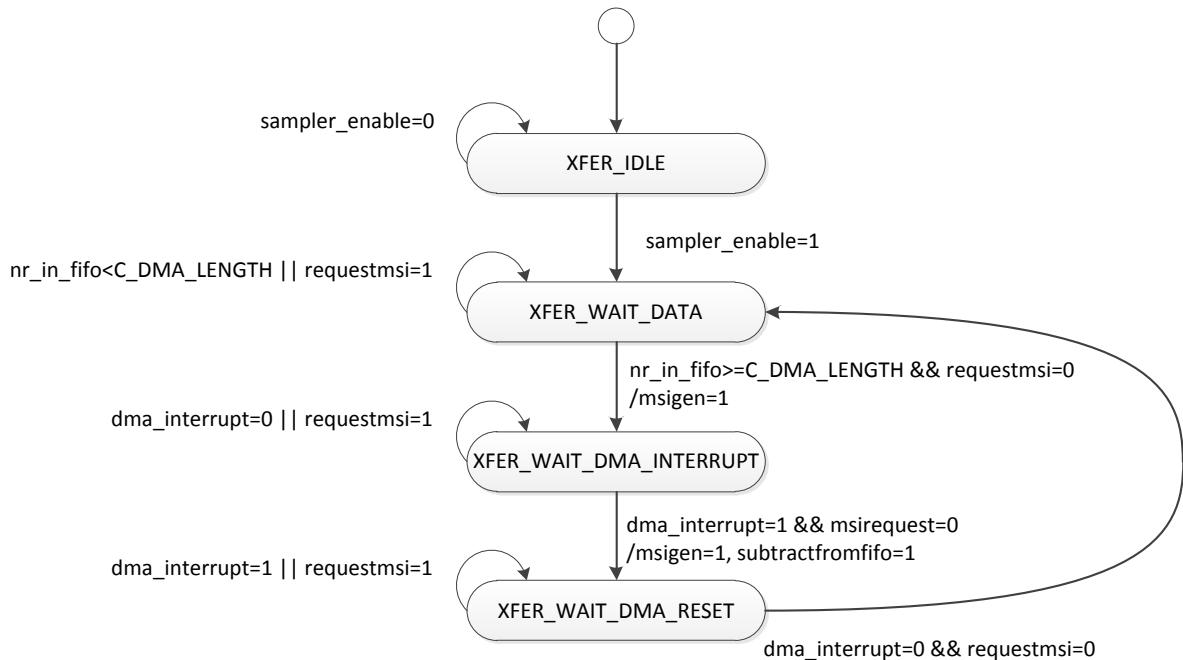
Het proces **Generate MSI** implementeert een schuifregister, dat ervoor zorgt dat een puls op een ingangssignaal een voldoende lange puls levert op de **msi_request**-lijn van de PCIe-core. Voorwaarde is dat de puls op het ingangssignaal één **PLB**-periode lang is. De communicatie met het **Generate MSI** proces verloopt via een two-way handshake.

Communicatie met de PC

Bij elke dataoverdracht naar de PC zijn een aantal acties betrokken. Deze acties worden uitgevoerd het proces **Data Transfer**. Het toestandsdiagram van dit proces is te zien op Figuur 6.13.

Bij het opstarten van de FPGA is de sampler nog niet ingeschakeld. Zolang de sampler uitgeschakeld is, blijft het communicatieproces in de toestand **XFER_IDLE**.

Vanaf het moment dat de gebruiker de sampler inschakelt, begint het proces met wachten tot er voldoende data in de FIFO van de arbiter zit in de toestand **XFER_WAIT_DATA**. De variabele **nr_in_fifo** houdt het aantal samples bij dat in de FIFO zit. Het aantal samples in een DMA blok wordt gegeven door de constante **C_DMA_LENGTH**. Als er zich voldoende samples in de FIFO bevinden om een DMA blok te vullen, wordt er een MSI gegenereerd door de ingang van het **Generate MSI**-proces (**msigen**) hoog te brengen. De PC weet bij het ontvangen van de interrupt dat ze een DMA-overdracht moet initiëren.



Figuur 6.13: Toestandsdiagram voor de communicatie tussen de FPGA en de PC.

De toestand `XFER_WAIT_DMA_INTERRUPT` wacht tot de DMA-overdracht voltooid is. Wanneer de DMA-controller klaar is met data over te zetten, genereert het een interrupt op de signaallijn `dma_interrupt`. De arbiter laat dit aan de PC weten door een nieuwe MSI te genereren. Er zijn `C_DMA_LENGTH` samples overgezet (de grootte van een DMA blok), dus dit getal kan van `nr_in_fifo` getrokken worden. Dit laatste gebeurt in een apart proces, dat het schrijven naar de FIFO van de arbiter beheert. Er wordt overgegaan naar de volgende toestand, `XFER_WAIT_DMA_RESET` op voorwaarde dat er geen MSI meer gegenereerd wordt.

De PC reageert op de laatst ontvangen interrupt door de DMA-controller te resetten. De arbiter weet dat dit gebeurd is wanneer de lijn `dma_interrupt` laag wordt. Op voorwaarde dat er geen MSI gegenereerd wordt, gaat de arbiter opnieuw wachten tot er voldoende samples in de FIFO van de arbiter zitten om een nieuw DMA blok te vullen.

Opdeling in klokdomeinen

Teneinde het FPGA-ontwerp sneller maken, werken we met 3 klokdomeinen. Deze zijn te zien in Figuur 6.10. We onderscheiden de volgende indeling:

1. **Het samplerdomein** - De sampler is de meest kritische component van het systeem: willen we de meetnauwkeurigheid verhogen, dan moet de samplerfrequentie naar omhoog. Om zo weinig mogelijk componenten te hebben die deze maximale samplerfrequentie potentieel kunnen beïnvloeden, krijgt de sampler een eigen klokdomein toegekend.
2. **Het arbiterdomein** - De arbiter zit samen met de geheugencontroller in een eigen klokdomein. Dit doen we omdat het SRAM-geheugen met een maximale frequentie van 200 MHz aangesproken kan worden [37], en de PLB-bus deze frequentie lang niet bereikt.

3. **Het PLB-domein** - De DMA-controller, de PCIe-core, de Configurator en de PLB-interface van de arbiter zijn gegroepeerd in een derde en laatste klokdomain. Vermits de PCIe-core door de PC gebruikt wordt om de DMA-controller in te stellen, en de DMA-controller de arbiter moet kunnen uitlezen om data over te zetten, horen deze componenten samen in hetzelfde klokdomain. De maximale werkfrequentie van dit domein is beperkt door de componenten die met de PLB-bus verbonden zijn. De PCIe-core is hier de beperkende factor, waardoor de maximale frequentie tussen 125 MHz en 150 MHz ligt [59].

6.4.3 Aanmaken van de FIFO's

Wanneer de standaard IP's die geleverd worden door Xilinx voor een bepaalde toepassing niet volstaan, kunnen er geparametriseerde versies van deze IP's gegenereerd worden. Xilinx biedt hiervoor een applicatie aan die *Core Generator* heet. Voor elke parametriseerbare component voorziet deze applicatie in een wizard die het mogelijk maakt op een overzichtelijke manier de verschillende parameters naar wens in te stellen.

Zowel de FIFO van de sampler als de FIFO van de arbiter zijn geparametriseerde versies van de IP core "FIFO Generator" [62]. De belangrijkste eis van beide FIFO's is de mogelijkheid om te kunnen lezen en schrijven in verschillende klokdomainen. Figuur 3-1 in de handleiding van de *FIFO Generator* geeft de mogelijke lees- en schrijfsignalen weer, geordend per klokdomain. De overgang tussen het lees- en het schrijfdomein gebeurt aan de hand van interne logica.

Het geheugen dat gebruikt wordt door de FIFO's bevindt zich, uiteraard, in de FPGA. Er bestaan drie types geheugen: BRAM, distributed RAM en built-in FIFO. De ingebouwde FIFO's zijn het efficiëntst: in de core generator geven we de lees- en schrijf klokfrequentie in, waarmee de core generator een geoptimaliseerde FIFO genereert.

Het uitlezen van de gegenereerde FIFO's kan op twee manieren gebeuren: standaard of First Word Fall Through (FWFT). Bij de eerste manier is de uitgelezen data beschikbaar op de kloktik na het aanvragen van een datawoord. De tweede manier stelt dit datawoord ter beschikking op dezelfde kloktik als wanneer een leesopdracht gegeven wordt. De FIFO van de sampler en FIFO van de arbiter zijn allebei FWFT FIFO's.

De grootte van de FIFO van de arbiter werd gekozen op 16 kB. Rekening houdend met de breedte van een sample (32 bits) kan de FIFO van de arbiter 4096 samples bevatten. De FIFO van de sampler is gedimensioneerd op 1024 samples, wat overeen komt met een grootte van 4096 kB.

De FIFO implementeert een aantal vlaggen om haar toestand naar buiten te brengen, waaronder een "vol"-vlag. Zolang de "vol"-vlag gebruikt wordt om logica aan te sturen aan de schrijfkant van de FIFO is er geen probleem. De problemen komen echter wanneer de FIFO vol zit en er een waarde wordt uitgelezen. Door de synchronisatie tussen het lees en het schrijf klokdomain duurt het enkele kloktikken vooraleer de, in dit geval, "vol"-vlag gereset wordt. Wat de vertraging op deze signalen is, staat neergeschreven in Tabel 3-24 van de handleiding [62]. Om geen invloed te hebben van de vertraging, houdt de arbiter zelf bij hoeveel samples er naar de FIFO van de arbiter geschreven werden. De sampler houdt enkel rekening met de "vol"-vlag van de FIFO van de sampler.

6.5 Communicatie FPGA-PC m.b.v. PCIe

Om tweezijdige communicatie mogelijk te maken tussen een hostcomputer en het FPGA-bord, werd er gekozen voor PCIe als interface. Het PCIe-protocol wordt verzorgd door de component *PCIe-core* op de FPGA. Deze component wordt als softcore geleverd door Xilinx.

De hoofdfunctie van de PCIe-core is het vertalen van adressen binnen de adresruimte van de FPGA naar adressen binnen de adresruimte van de PC, en omgekeerd. Op die manier is het mogelijk om vanaf de PC data te lezen vanuit een register op de FPGA, op voorwaarde dat dit register adresseerbaar is op de PLB-bus. De FPGA kan eveneens data schrijven naar het RAM-geheugen van de PC.

6.5.1 Instellingen van de PCIe-core

Voor elke component die Xilinx aanbiedt, zijn er een aantal instelbare parameters. De belangrijkste parameters voor de PCIe-core staan neergeschreven in Tabel 6.7. Meer informatie over deze parameters is te vinden in Tabel 1 van de datasheet [59].

<i>Algemene instellingen</i>	
C_IPIFBAR_NUM	1
Het aantal geheugenregio's van de PC die we willen aanspreken vanuit de FPGA.	
C_PCIBAR_NUM	3
Het aantal geheugenregio's van de FPGA die we willen aanspreken vanuit de PC. In ons geval zijn dit er 3: de arbiter, de DMA-controller en de PCIe-brug (om te debuggen).	
C_BOARD	ML505
Dit is het FPGA-bord waarmee we werken.	
C_COMP_TIMEOUT	1
De maximale tijd dat een PCIe-pakket kan wachten vooraleer verzonden te worden naar de PC. Als deze tijd verstrekken is, wordt het pakket weggegooid. 0 betekent een timeout van 50 µs, 1 betekent een timeout van 50 ms. Wij kiezen voor de langste levensduur.	
C_NO_OF_LANES	1
Aantal lanes waarover de PCIe-link beschikt. Het FPGA-bord voorziet slechts in een x1-link, dus deze variabele moet 1 zijn.	

<i>Header instellingen</i>	
C_DEVICE_ID	0x0505
Herkenningsnummer waaraan een stuurprogramma op de PC deze PCIe-kaart kan herkennen. Dit mag gelijk zijn aan om het even wat, zolang het stuurprogramma zoekt naar een PCIe-apparaat met een Device ID die gelijk is aan deze waarde.	
C_VENDOR_ID	0x10EE
Herkenningsnummer waaraan een stuurprogramma op de PC deze PCIe-kaart kan herkennen. De waarde 0x10EE staat voor Xilinx [43]. In principe mag ook deze waarde vrij gekozen worden, zolang het stuurprogramma zoekt naar een PCIe-apparaat met een Vendor ID met deze waarde.	

<i>Geheugenzones van de PLB-bus toegankelijk vanuit de PC</i>	
C_PCIBAR_LEN_0	16
Standaardgrootte van 64 KB.	
C_PCIBAR2IPIFBAR_0	0xCEE00000
De eerste component die we vanaf de PC willen aanspreken, is de arbiter. In deze registerbank zijn de status- en controleregisters beschikbaar gesteld.	
C_PCIBAR_LEN_1	16
Standaardgrootte van 64 KB.	
C_PCIBAR2IPIFBAR_1	0x80200000
De tweede component die aanspreekbaar moet zijn vanuit het stuurprogramma, is de DMA-controller.	
C_PCIBAR_LEN_2	16
Standaardgrootte van 64 KB.	
C_PCIBAR2IPIFBAR_2	0x85C00000
De derde en laatste component is meer bedoeld om te debuggen. Op dit adres bevindt zich de PLB-kant van de PCIe-brug. Dit BAR biedt dus de mogelijkheid aan de PC de PCIe configuratieregisters van de FPGA uit te lezen.	

<i>Toegang tot het RAM-geheugen van de PC</i>	
C_IPIFBAR_0	0xA0000000
PLB-bus adres van een stuk RAM-geheugen op de PC. Als een component op de PLB-bus wil schrijven naar adres x in het RAM-geheugen van de PC, dan moet het een schrijfrequest doen naar $0xA0000000 + x$.	
C_IPIFBAR_HIGHADDR_0	0xBFFFFFFF
Dit is het einde van de aperture in de PLB-adresruimte die gemapt is op het RAM-geheugen van de PC.	

Tabel 6.7: Parameters van de PCIe-core.

6.5.2 Inschakelen van de PCIe-vertaalbrug

De PCIe-controller beschikt over een *translation bridge* die adressen vertaalt van het PLB-domein (FPGA) naar het PCIe-domein (PC) en omgekeerd. Deze vertaling gebeurt echter niet wanneer de brug is uitgeschakeld, wat het geval is bij het programmeren van de FPGA. Er moet dus een oplossing bedacht worden die de PCIe-vertaalbrug inschakelt telkens wanneer de FPGA uit haar resettoestand komt.

Na het bestuderen van een referentie-ontwerp [55], en de datasheet van de PCIe-core bekijken te hebben [59], lag de oplossing voor de hand: configureren het Bridge Control Register (BCR). De bitlayout van dit register is te zien in Figuur 6.14.

De eerste nuttige bit, **BME**, staat voor *Bus Master Enable*. Deze bit moet aan staan om de PCIe-core te laten weten dat ze zich kan en mag gedragen als busmaster. Dit is nodig om te kunnen lezen en schrijven, met andere woorden: om enige geheugentoegang te kunnen hebben via PCIe moet deze bit aan staan.

De tweede en laatste set nuttige bits, **E0-E2**, schakelen de BARs in of uit. Uitschakelen van een BAR komt erop neer dat alle lees- of schrijfrequests van de PC voor dat specifieke BAR worden genegeerd. Wanneer we de PC dus toegang willen geven tot het venster van de PLB-adresruimte dat beschikbaar gesteld wordt door een bepaald BAR, moet Ex aan staan. In de huidige implementatie worden de 3 beschikbare BAR's gebruikt, dus moeten de 3 bits aan staan.

Uit bovenstaande redenering volgt dat de waarde die naar het BCR geschreven moet worden gelijk is aan **0x00000107**.

31	9 8 7	3 2 0
Gereserveerd	B M E	Gereserveerd E0-E2

Figuur 6.14: Bitlayout van het BCR.

6.5.3 Het instellen van een configuratieregister

Normaal gezien gebeurt het configureren van de IP cores op de FPGA met behulp van een programma dat op een processor draait. In het geval van deze thesis, waar er maar 1 register ingesteld moet worden, zou dit een significante verhoging in complexiteit vereisen. Complexiteit die bovendien kan vermeden worden.

Er werd dus gekozen voor het schrijven van een eigen hardwarecomponent die enkel en alleen actief is wanneer de FPGA uit haar reset toestand komt, de *Configurator* op figuur 6.1. Deze component genereert een PLB-schrijfrequest naar het BCR van de PCIe-core.

Uit tabel 6.1 volgt dat het basisadres van de PCIe-registerbank gelijk is aan **0x85C00000**. De offset van het BCR in deze registerbank is te vinden in de gebruikershandleiding van de PCIe-core [59]: **0x00000030**. Er moet dus een schrijfrequest gegenereerd worden die **0x00000107** schrijft naar het adres **0x85C00030**.

Eens deze configuratie gebeurd is, doet de *Configurator* niets meer.

6.6 Data overbrengen in burst m.b.v. DMA

Voor het uitvoeren van DMA overdrachten maken we gebruik van een softcore van Xilinx: Central DMA-controller [58]. Uit Tabel 6.1 volgt dat het basisadres van de DMA registerbank gelijk is aan **0x80200000**. We gaan nu dieper in op de waarden die naar de registers van de DMA-controller geschreven moeten worden om een overdracht te initiëren.

Vooraleer de sampler in te schakelen, en dus data te capteren, moet de PC geconfigureerd worden om DMA te ontvangen. Het besturingssysteem op de PC reserveert een stuk van het RAM-geheugen voor DMA, en deelt dit geheugen op in verschillende buffers met eenzelfde grootte. De grootte van deze buffers is gelijk aan 4 keer het aantal samples per DMA blok (1 sample is 32 bits, of 4 bytes).

Eerst moet het **DMA Control Register** ingesteld worden zodat de DMA-controller weet wat het moet doen met het bron- en doeladres. Elk van beide adressen kan zowel als statisch (bv. een FIFO) als incrementeel behandeld worden. In dit geval is het bronadres statisch (de FIFO van de arbiter), het doeladres moet geïncrementeerd worden (RAM-geheugen). Volgens de datasheet moet dus de waarde **0x40000000** naar het adres **0x80200004** geschreven worden.

Het tweede register is het **Source Address Register**. Dit bevat het adres waarvan gelezen moet worden. In dit geval is het een adres binnen de registerbank van de arbiter, namelijk de FIFO van de arbiter. Uit Tabel 6.1 volgt dat het basisadres van de arbiter gelijk is aan **0xCEE00000**. Samen met Tabel 6.5 weten we dat het adres van de FIFO gelijk is aan **0xCEE00004**. De waarde **0xCEE00004** moet dus geschreven worden naar het adres **0x80200008**.

Het volgende register is het **Destination Address Register**. Hierin moet het adres komen waarnaar de data geschreven moet worden. Uit Tabel 6.1 is af te leiden dat het basisadres om naar de PC te schrijven gelijk is aan **0xA0000000**. Het besturingssysteem beschikt over een aantal DMA buffers die elk een bepaald stuk van het RAM-geheugen innemen. Bij het initiëren van een DMA overdracht kiest het besturingssysteem één van deze buffers, en schrijft

`0xA0000000` vermeerderd met het fysische adres van de gekozen buffer naar het DMA register `0x8020000C`.

Uit de bespreking van de werking van de arbiter volgt dat de DMA-controller een interrupt moet genereren wanneer de overdracht voltooid is. Dit wordt ingesteld met behulp van het **Interrupt Enable Register**. Er kunnen twee interrupts ingeschakeld worden: één die aangeeft of de overdracht met succes is voltooid en één die aangeeft dat de overdracht vroegtijdig is afgebroken. In beide gevallen moet de PC actie ondernemen, dus moeten beide interrupts gegenereerd worden. Volgens de datasheet moet de waarde `0x00000003` geschreven worden naar het adres `0x80200030`.

Als laatste moet het **Length Register** ingesteld worden. Hierin moet het aantal *bytes* geschreven worden dat overgezet moet worden. Net zoals bij het bepalen van de grootte van de DMA buffers, is dit hier gelijk aan 4 keer het aantal samples per DMA blok. Schrijven naar dit register begint de DMA overdracht. Het adres van dit register is `0x80200010`.

6.7 Optimalisatie van het ontwerp

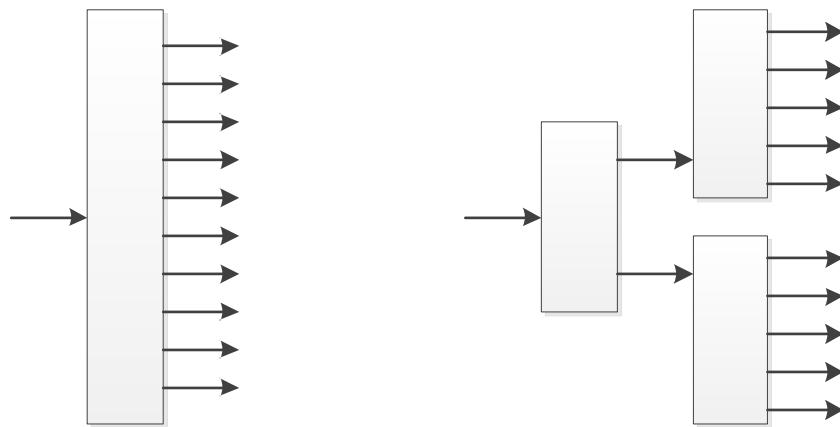
Bij het synthetiseren van de eerste versies van het FPGA-ontwerp kregen we de melding dat het ontwerp te traag was: de klokfrequenties die we opgegeven hadden werden niet gehaald. Om de oorzaak van dit “traag zijn” te onderzoeken, hebben we gebruik gemaakt van *Xilinx Timing Analyzer*.

De *Timing Analyzer* drukt de tijd dat een signaal onderweg is van punt A naar punt B uit in *slack*. Deze slack wordt grofweg berekend door de vertraging in het gesynthetiseerde ontwerp af te trekken van de maximaal toelaatbare vertraging. Deze laatste wordt bepaald door de gewenste klokfrequentie. Een negatieve slackwaarde betekent dat het signaal te lang onderweg is. Een positieve waarde betekent dat het signaal snel genoeg op zijn bestemming geraakt. Uiteindelijk is het beperkende pad datgene met de meest negatieve slackwaarde.

De belangrijkste parameter die de *slack* bepaalt, is de vertraging veroorzaakt door het datapad. Deze vertraging is enerzijds te wijten aan de routering binnenin de FPGA en anderzijds aan het aantal logische niveau’s dat het signaal moet doorlopen. Naast absolute waarden geeft de *Timing Analyzer* procentueel aan wat het aandeel is van beide factoren. Xilinx raadt aan te optimaliseren tot routering het grootste aandeel van de totale propagatietijd inneemt [36].

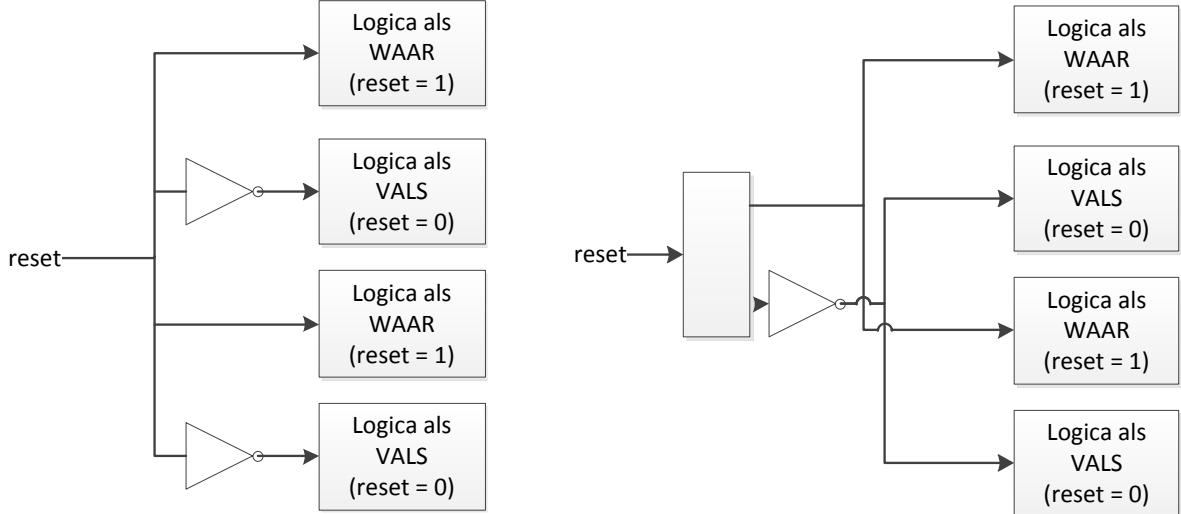
Niet alleen het *aantal* logische niveau’s, maar ook de *fanout* van de verschillende signalen is van belang: een signaal met een grote fanout is inherent traag. Dit probleem treedt vooral op bij resetsignalen. Elke bit van elke variabele die een resetwaarde toegekend krijgt, heeft één verbinding met het resetsignaal. De fanout van het resetsignaal kan verlaagd worden ten koste van een extra logisch niveau. De winst die we maken door de fanout te verlagen is echter superieur ten opzichte van het tijdsverlies door het extra niveau. Figuur 6.15 verduidelijkt dit. Links op Figuur 6.15 zien we een signaal met een grote fanout, en dus een grote poortvertraging. De rechterkant van Figuur 6.15 geeft de oplossing voor dit probleem weer. Het toepassen van deze methode had een positieve impact op de timing van het ontwerp.

Uiteindelijk bleek dat het verlagen van de fanout alleen niet voldoende was. Het resetsignaal werd nog teveel belast. Om het resetsignaal verder te ontlasten, hebben we het ontdubbeld door



Figuur 6.15: Optimalisatie van het FPGA-ontwerp: verlagen van de fanout.

een kopie en een geïnverteerde versie ervan te maken. Het voordeel van dergelijke ontdubbeling kunnen we inzien door na te denken over hoe een conditionele statement, bijvoorbeeld een **if**-constructie, gesynthetiseerd wordt. Een voorbeeld van een gesynthetiseerde **if**-constructie is te zien in Figuur 6.16. Links zien we de “naïve” manier, rechts de geoptimaliseerde synthese. Elke kopie van het resetsignaal moet rechts slechts twee signalen aandrijven, terwijl dat links nog vier was. Op deze figuur lijkt de winst marginaal, maar bij een groter ontwerp zoals dit project levert het ontdubbelen van zwaar belaste signalen significante tijdsvermindering op.



Figuur 6.16: Optimalisatie van het FPGA-ontwerp: ontdubbeling van het resetsignaal.

Na het toepassen van deze optimalisaties was onze eigen implementatie niet meer de beperkende factor van het ontwerp. De traagste signaallijn in de FPGA zit nu in de gesynthetiseerde FIFO's. Binnen deze thesis was helaas er te weinig tijd over om ook dit nog op te lossen. We stranden op de volgende frequenties voor de drie klokdomen:

1. het samplerdomein werkt op 50 MHz;
2. het arbiterdomein werkt op 60 MHz;
3. het PLB-domein werkt op 55 MHz.

Hoofdstuk 7

PC-implementatie

In de vorige hoofdstukken hebben we besproken hoe we nuttige data over de uitvoering van systeemsoftware naar buiten kunnen brengen, hoe deze gegevens geobserveerd worden, en op welke manier ze verzonden worden naar de hostcomputer. Dit hoofdstuk behandelt het laatste stuk van de puzzel: hoe de data opgeslagen en verwerkt wordt.

Net zoals in de vorige hoofdstukken starten we met het geven van een schematisch overzicht, en bespreken we vervolgens elk onderdeel in meer detail.

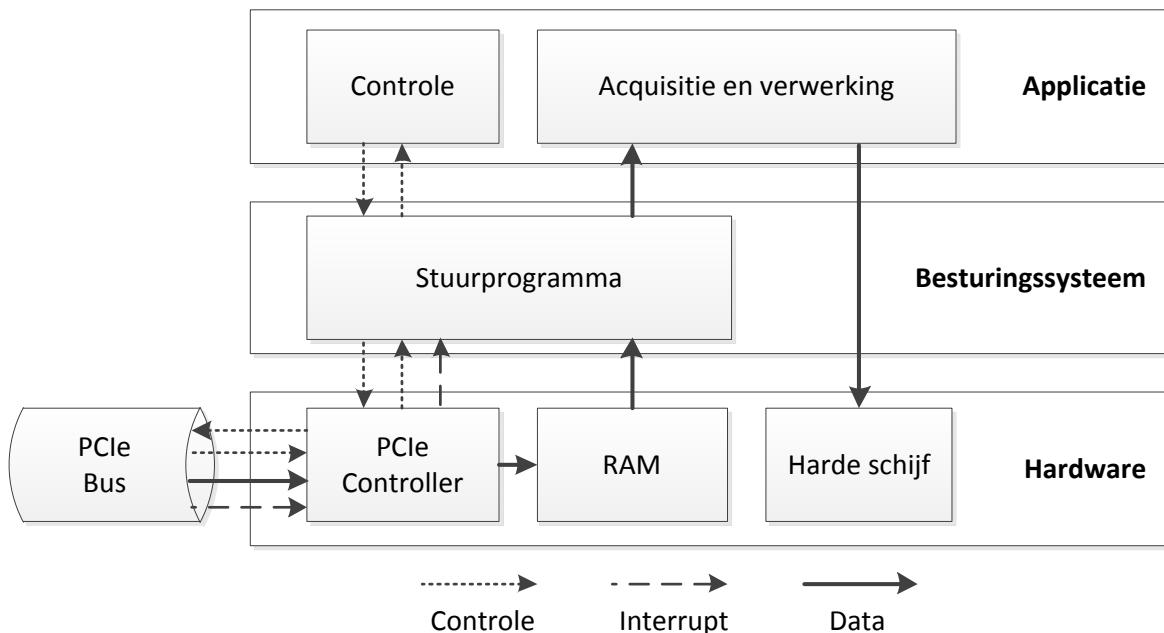
7.1 Ontwerp

Figuur 7.1 geeft de ontworpen softwarestapel weer. Het besturingssysteem dat wij gebruiken is Gentoo Linux 3.0.35. De processor van de PC is een Intel Core2 DUO CPU E8400 die geklokt is op 3 GHz. Bij het opstarten van de PC initialiseert een eigen stuurprogramma het FPGA-bord en zet het er een verbinding mee op. De data komt de PC binnen via de PCIe-bus linksonder op Figuur 7.1. Deze data wordt in het RAM-geheugen opgeslagen. Communicatie tussen het FPGA-ontwerp en de software op de PC gebeurt aan de hand van interrupts. Net zoals de data komen deze de PC binnen via de PCIe-bus. Het stuurprogramma handelt deze interrupts op een gepaste wijze af.

Door het gebruik van geheugenbescherming in moderne besturingssystemen kan een gebruikersapplicatie niet rechtstreeks aan data op willekeurige plaatsen in het RAM-geheugen. Het stuurprogramma zorgt ervoor dat deze data wel toegankelijk is voor de gebruiker.

In de applicatielaag onderscheiden we volgende programma's:

1. een controle-applicatie om de sampler in of uit te schakelen, te resetten, ...;
2. een acquisitie-applicatie die de samples streamt naar een bestand op de harde schijf;
3. een parser die een bestand met samples controleert op fouten en een consistente tijdsbasis genereert door de carrybit weg te werken;
4. een programma om de samples te visualiseren.



Figuur 7.1: Blokschema van de PC-implementatie.

7.2 Het stuurprogramma

De functionaliteit van het stuurprogramma is tweeledig. In de eerste plaats zorgt het stuurprogramma ervoor dat de sampler vanuit een normale gebruikersapplicatie kan worden gecontroleerd. In de tweede plaats stuurt het de PCIe-kaart gepast aan om ontvangst van data mogelijk te maken.

Het stuurprogramma zelf hebben we geïmplementeerd als een dynamisch laadbare kernelmodule. Dit betekent dat ons stuurprogramma niet in de kernel geïntegreerd is, maar dat het op het gepaste moment ingeladen en geactiveerd kan worden.

7.2.1 Userspace en kernelspace

De softwarestack in Figuur 7.1 bestaat uit drie lagen: de hardwarelaag, de besturingssysteemlaag en de applicatielaag. Om applicaties tegen elkaar te beschermen, voorziet het OS voor elke applicatie in een eigen virtuele adresruimte. Deze adresruimte zal de code en data van de applicatie omvatten, maar ze zal geen toegang verlenen tot de hardware. Het proces kan hiervoor communiceren met het besturingssysteem door middel van system calls.

We onderscheiden nu twee privilege niveau's waarin instructies uitgevoerd kunnen worden: het kerneldomein en het gebruikersdomein. In vaktermen spreekt men van kernelspace respectievelijk userspace. Het eerste niveau wordt gebruikt door de kernel en is bevoordeerd: het kan rechtstreeks de hardware aanspreken, en virtuele adresruimten beheren. Het tweede niveau is onbevoordeerd en wordt gebruikt voor applicaties. Hun adresruimte wordt beheerd door de kernel en alle toegangen tot hardware moeten eveneens via de kernel verlopen.

7.2.2 Interface met gebruikersapplicaties

Om communicatie mogelijk te maken tussen gebruikersapplicaties en stuurprogramma's voorziet Linux in een speciale map op het bovenste niveau van het bestandssysteem, `/dev`. In deze map zitten geen echte bestanden, maar eerder symbolische voorstellingen van de hardwarecomponenten die door de kernel gedetecteerd werden, *device nodes* genoemd. Een gebruikersapplicatie kan opdrachten geven aan een hardwarecomponent door het overeenkomende bestand te openen en ernaar te schrijven. Analoog kan een applicatie data opvragen door van dat bestand te lezen.

Alnaargelang de soorten lees- en schrijfoperaties maakt Linux onderscheid tussen de volgende types *device nodes*:

1. *Character devices* - Dit zijn apparaten die byte per byte uitgelezen worden.
2. *Block devices* - Dit zijn apparaten die blok per blok uitgelezen worden. Een blok kan bijvoorbeeld 512 Bytes omvatten.

Een stuurprogramma dat een bepaald type *device node* implementeert, moet afhankelijk van dit type een aantal vooropgestelde functies aanbieden. Rekening houdend met het gebruik van DMA om data over te zetten, lijkt de meest voor de hand liggende keuze een *block device*. Er bestaat echter een efficiëntere manier om de DMA blokken door te geven aan een gebruikersapplicatie dan de blokken gewoon te kopiëren. Als we daarenboven nog weten dat een *character device* over het algemeen makkelijker te implementeren is, kiezen we uiteindelijk toch voor een *character device* [35].

7.2.3 Detectie en initialisatie van het FPGA-bord

De functionaliteit beschreven in deze sectie wordt uitgevoerd bij het inladen van de kernelmodule.

Het allereerste wat er moet gebeuren, is kijken of het FPGA-bord al dan niet aanwezig is in de computer. We doen dit door op zoek te gaan naar een PCIe-apparaat met een **Vendor ID** en **Device ID** die gelijk zijn aan de waarden die wij hebben meegegeven aan de PCIe-core op de FPGA. Deze instellingen zijn terug te vinden in Tabel 6.7. Eens we een dergelijk PCIe-apparaat gedetecteerd hebben, schakelen we dit in.

Zoals beschreven in Sectie 5.4, kan de CPU geheugen op de PCIe-kaart lezen of schrijven via zogenaamde BARs. Aan elk BAR heeft het BIOS bij het opstarten van de PC een stuk van de fysieke adresruimte toegekend. Het stuurprogramma vraagt de locaties van deze stukken geheugen op aan het besturingssysteem om ze vervolgens te mappen in de virtuele adresruimte van de kernel. Voor dit project maken we gebruik van de volgende drie BARs:

1. **BAR 0** - Hiermee kunnen we het controleregister van de arbiter opvragen en instellen, evenals de arbiter FIFO uitlezen.
2. **BAR 1** - Hiermee krijgen we toegang tot de registerbank van de DMA-controller.
3. **BAR 2** - Hiermee kunnen we de configuratieregisters van de PCIe-core op de FPGA opvragen.

Dataoverdracht tussen de FPGA en de PC wordt voornamelijk geregeld aan de hand van interrupts. Zo genereert de FPGA een interrupt wanneer er data klaar staat om te verzenden, maar ook wanneer de DMA-controller klaar is met data over te zetten. Aan de PC-kant krijgt elke interrupt die gegenereerd wordt een uniek nummer toegekend. Dit nummer is afhankelijk van het apparaat dat die interrupt genereert: zo zit de systeemtimer op interrupt 0, het toetsenbord op interrupt 1, enz. Het besturingssysteem houdt voor elke interrupt een pointer bij naar de functie die aangeroepen moet worden telkens wanneer die interrupt voor komt. Om de interrupts die gegenereerd worden door het FPGA-bord op te vangen, moet ons stuurprogramma volgende zaken in orde brengen:

1. vraag het besturingssysteem een interruptnummer toe te kennen aan de PCIe-kaart;
2. geef het adres van de op te roepen functie, de *interrupt handler*, door aan het besturingssysteem.

Teneinde DMA-overdrachten vanop het FPGA-bord mogelijk te maken, moet het PCIe-apparaat expliciet toestemming krijgen om busmaster te mogen zijn over de geheugenbus van de PC. Vervolgens vragen we het besturingssysteem een aantal DMA-buffers toe te wijzen om de samples in op te slaan. We gebruiken hier een circulaire buffer van DMA-buffers om enig verschil in snelheid tussen het ontvangen van samples en het verwerken ervan toe te laten. Dit is vooral handig wanneer samples voor korte perioden sneller binnenkomen dan ze verwerkt kunnen worden.

Tenslotte registreren we ons stuurprogramma in de kernel als een *character device*. De kernel reageert op deze registratie door een *device node* aan te maken. Gebruikersapplicaties kunnen met ons stuurprogramma communiceren door deze *device node* te openen en ernaar te schrijven of ervan te lezen.

7.2.4 Geheugen beschikbaar stellen voor de gebruiker

De meest gebruikte manier om data vanuit de kernel door te geven aan de gebruiker, is deze data te kopiëren naar een buffer in userspace. Het nadeel aan deze methode is dat de data twee keer in het geheugen aanwezig is. Het voordeel is dan weer veiligheid. Doordat de applicatie steeds met een kopie van de data werkt, is het onmogelijk data in de kernel corrupt maken.

Onze toepassing moet zo snel mogelijk zijn, waardoor we liever geen identieke kopie maken van de DMA-buffers, maar rechtstreeks werken met de inhoud ervan. De kopieerstap kan vermeden worden door de DMA-buffers te *mappen* in de virtuele adresruimte van de applicatie. We spreken van een zogenaamde *zero-copy* operatie [47].

Wanneer een DMA-buffer aan het besturingssysteem gevraagd wordt, reserveert de laatste een stuk van het fysieke RAM-geheugen van de PC. Dit stuk geheugen wordt *gemapt* in de virtuele adresruimte van de kernel. Door deze mapping door te zetten naar de virtuele adresruimte van de applicatielaag, kunnen we vanuit een applicatie rechtstreeks bewerkingen doen op de data in de DMA-buffers.

7.2.5 Leesoperaties afhandelen

In essentie implementeren we hetgeen men een *blocking read* noemt [35]. Dit betekent dat de applicatie na het geven van een leescommando blijft wachten tot dat leescommando voltooid is. Voor deze toepassing is een *blocking read* aan de orde, omdat we een streamingapplicatie willen koppelen aan ons stuurprogramma. Deze applicatie wacht continu op data van dit stuurprogramma en voert daartoe een blocking read uit.

De blocking read is voltooid wanneer er een ongelezen DMA-buffer beschikbaar is.

7.2.6 Gebruikerscommando's afhandelen

Ons stuurprogramma voorziet in functionaliteit om de sampler te controleren vanuit een gebruikersapplicatie. Een applicatie kan eveneens informatie van het stuurprogramma zelf opvragen.

De eerste commando's geven informatie door over het stuurprogramma zelf:

- **Test communicatie met het stuurprogramma** - Dit commando kan gebruikt worden om na te gaan of de communicatie tussen een applicatie en het stuurprogramma correct verloopt. Het geeft een vooraf bepaalde constante terug.
- **Vraag het aantal gealloceerde DMA-buffers op** - De gebruikersapplicatie moet kennis hebben van het aantal gealloceerde DMA-buffers om ze allemaal te kunnen mappen in userspace zoals beschreven in Sectie 7.2.4.
- **Klaar met lezen** - Wanneer de gebruikersapplicatie klaar is met het verwerken van een DMA-buffer, laat het dit weten aan het stuurprogramma d.m.v. dit commando. Het stuurprogramma weet dan dat de inhoud van de buffer overschreven mag worden.

De volgende commando's hebben betrekking op de sampler:

- **Status van de sampler opvragen** - Dit commando leest het controle/statusregister uit van de arbiter. De bitlayout van het resultaat is te zien in Figuur 6.11. Een beschrijving van de bits is terug te vinden in Tabel 6.6.
- **In- en uitschakelen van de sampler** - Deze commando's maken het mogelijk voor de gebruiker om de sampler in en uit te schakelen op het gepaste moment.
- **Flush de FIFO van de arbiter** - Als de sampler uitgeschakeld is en er nog data in de FIFO van de arbiter zit, dan zal er nooit een interrupt gegenereerd worden om deze data over te zetten. De sampler staat immers uit. Om deze data wel over te zetten, kunnen we de FIFO van de arbiter legen. Dit commando forceert een DMA-overdracht.
- **Resetten van de sampler** - Dit commando maakt het mogelijk de sampler te resetten zonder de FPGA opnieuw te moeten programmeren, of de PC opnieuw te moeten opstarten. We raden aan elke nieuwe meting vooraf te laten gaan door dit commando.

De laatste twee commando's zijn vooral nuttig bij het debuggen van het systeem:

- **Uitlezen van een DMA-register** - Met dit commando is het mogelijk de registerbank van de DMA-controller uit te lezen. Het argument van dit commando is het nummer van een 32-bit register in de registerbank.
- **Uitlezen van een PCIe-register** - De registerbank van de PCIe-core op de FPGA kan met dit commando uitgelezen worden. Het argument van dit commando is het nummer van een 32-bit register in de registerbank.

7.2.7 Interrupts afhandelen

Interrupts worden afgehandeld met een speciale functie, een *interrupt handler* genoemd. Dit zijn korte, snelle functies die aangeroepen worden telkens de bijhorende interrupt gegenereerd wordt. Het besturingssysteem houdt voor elke interrupt bij welke functie aangeroepen moet worden.

Uit de uiteenzetting van Sectie 6.4.2 weten we dat een interrupt van onze PCIe-kaart twee oorzaken kan hebben:

- **Data staat klaar** - Eerst en vooral controleert de interrupt handler of er wel een vrije DMA-buffer beschikbaar is. Is dit niet zo, dan schrijft ze een boodschap naar het kernel logboek om de gebruiker erop attent te maken dat er data verloren kan gegaan zijn. Is er een DMA-buffer beschikbaar, dan programmeert de interrupt handler de DMA-controller volgens de methode beschreven in Sectie 6.6.
- **DMA-overdracht voltooid** - Bij het ontvangen van deze interrupt kijkt de interrupt handler eerst na of de DMA-overdracht vroegtijdig werd afgebroken, dan wel volledig uitgevoerd werd. Bevinden we ons in het laatste geval, dan voeren we een reset uit van de DMA-controller. Na een dergelijke reset kan er een volgende blok overgeplaatst worden.

7.3 De controle-applicatie

Dit is een zeer eenvoudige applicatie die het aansturen van de sampler op een eenvoudige manier mogelijk maakt. Het programma stelt de controlebits van de arbiter, de DMA-registers en de PCIe-registers bovenindien voor op een gemakkelijk interpreteerbare manier.

Het enige argument dat meegegeven dient te worden aan deze applicatie, kan één van de volgende zijn:

- **enable**: sampler inschakelen
- **disable**: sampler uitschakelen
- **reset**: sampler resetten
- **flush**: FIFO van de arbiter legen
- **status**: controleregister van de arbiter interpreteren en weergeven
- **dma**: registerwaarden van de DMA-controller interpreteren en weergeven
- **pcie**: registerwaarden van de PCIe-core interpreteren en weergeven

7.4 Streamen van data naar de harde schijf

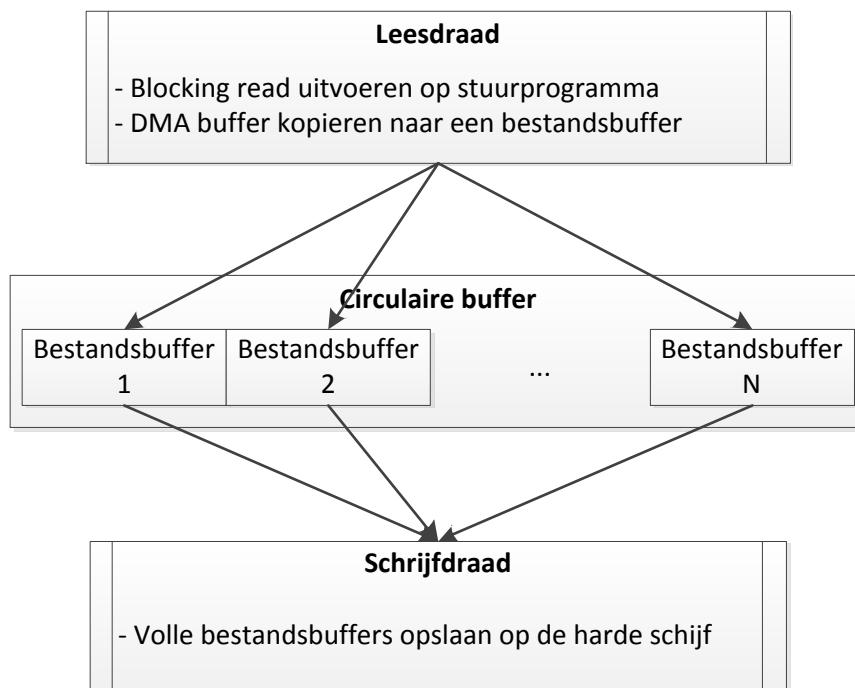
De ontwikkeling van een robuuste streamingapplicatie heeft een aanzienlijke tijd in beslag genomen.

De eerste versie van de streamingapplicatie maakte slechts gebruik van één thread, en schreef elk DMA-blok meteen weg naar de harde schijf. De doorvoersnelheid van deze eerste versie was bijzonder slecht: we haalden slechts 7.5 MB/s.

Een beter ontwerp van de streamingapplicatie ligt voor de hand wanneer we rekening houden met de verschillende lees- en schrijfsnelheid. De FPGA zendt immers samples naar de PC op een ander tempo dan dat deze worden weggeschreven naar de harde schijf. De meest voor de hand liggende oplossing is dan ook gebruik te maken van twee draden binnen de streamingapplicatie: een leesdraad en een schrijfdraad. Een harde schijf is echter niet goed in het verwerken van kleine blokken data. Om de snelheid van de harde schijf optimaal te benutten, bufferen we een aantal DMA-buffers in grotere buffers. Deze laatste noemen we *bestandsbuffers*. Enkel wanneer een bestandsbuffer vol is, wordt deze weggeschreven naar de harde schijf.

7.4.1 Werkingsprincipe

In essentie bestaat de streamingapplicatie uit twee threads die een circulaire buffer delen. Dit buffergeheugen is opgedeeld in een aantal kleinere buffers, die elk één bestandsbuffer voorstellen. Figuur 7.2 geeft hiervan een schematisch overzicht.

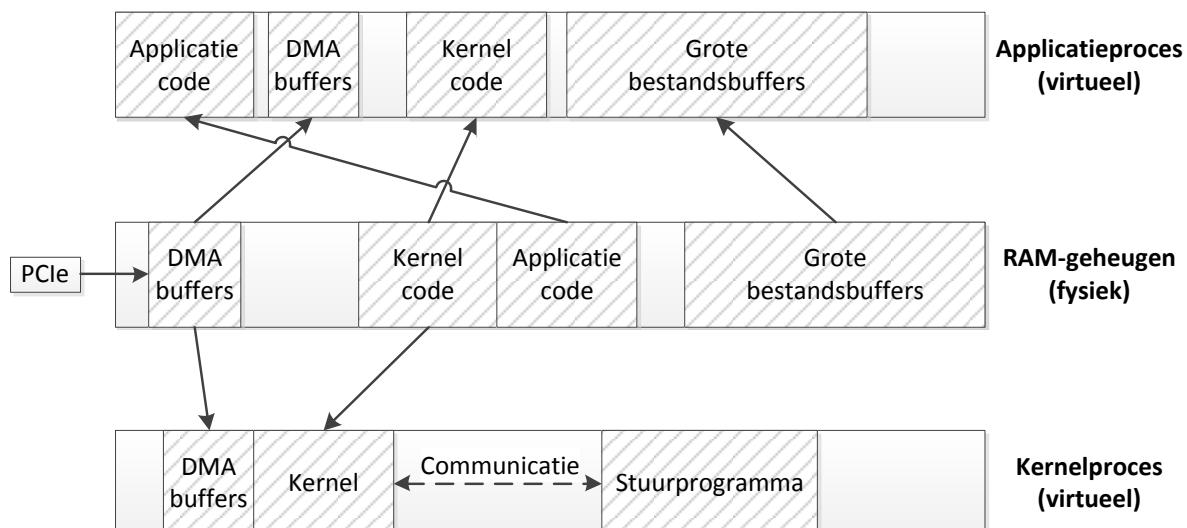


Figuur 7.2: Schematisch overzicht van de streaming applicatie.

Hoe de streamer samenwerkt met het stuurprogramma op vlak van geheugen, wordt weergegeven op Figuur 7.3. De pijlen stellen *mappings* voor. Centraal op deze figuur staat het fysieke RAM-geheugen dat alle instructies en data bevat. Zoals beschreven in Sectie 7.2.1 moeten we rekening houden met de virtuele adresruimtes: de applicatie wordt uitgevoerd in een eigen virtuele adresruimte (bovenaan), terwijl de kernel in (een) andere virtuele adresruimte(s) kan draaien, onderaan op Figuur 7.3. Elk stuk geheugen dat de streamingapplicatie moet kunnen aanspreken, moet gemapt worden in de virtuele adresruimte.

Het eerste geheugen dat de streamingapplicatie aanspreekt, zijn de DMA-buffers. Deze moeten dus gemapt worden in de virtuele adresruimte van de streamer. Om het aantal schrijfoperaties naar de harde schijf te verminderen, worden een hoop DMA-buffers per keer weggeschreven. Om dit mogelijk te maken, kopiëert de streamer elke DMA-buffer naar een grotere bestandsbuffer. Voor deze grote bestandsbuffers wordt een stuk van het RAM-geheugen gereserveerd. Net zoals de DMA-buffers moet ook dit geheugen gemapt worden in de virtuele adresruimte.

Om ons stuurprogramma en dus ook de kernel aan te spreken, moeten er een stuk van de kernel code gemapt worden in de virtuele adresruimte van de applicatie. Diezelfde code bestaat uiteraard ook in de adresruimte van de kernel zelf.



Figuur 7.3: Overzicht van de gebruikte mapping, userspace en kernelspace.

7.4.2 Alloceren van grote bestandsbuffers

Het is efficiënter grotere blokken data in één keer naar de harde schijf te schrijven dan meerdere kleine blokken achtereenvolgens op te slaan. Wij hebben dit opgelost door bestandsbuffers aan te maken waarin een groot aantal DMA-buffers gekopieerd kan worden.

Het werken met dergelijke bestandsbuffers mag de efficiëntie dan wel verhogen, het blijft nog altijd een zekere tijd duren om al die data weg te schrijven. Gedurende deze periode mag de inhoud van de bestandsbuffer niet aangepast worden. Als we dus zouden werken met één bestandsbuffer, dan zouden we kostbare tijd verliezen door te wachten tot de inhoud van deze

buffer is opgeslagen. Dit probleem kan vermeden worden door gebruik te maken van een circulaire buffer van bestandsbuffers. De geheugenvierrarchie van deze bestandsbuffers is te zien op Figuur 7.4.



Figuur 7.4: Hierrarchie van de bestandsbuffers.

De grootte van de bestandsbuffers werd redelijk arbitrair gekozen, de enige voorwaarde was dat ze “groot genoeg” moeten zijn. Wij hebben gekozen voor acht bestandsbuffers van 128 MB elk. Het alloceren van dergelijke blokken geheugen met de `malloc`-functie lukt echter niet. Een korte zoektocht naar manieren om grote buffers te verkrijgen van de kernel brengt ons bij de opstartparameters [5]. Er bestaat een boot parameter, `memmap`, die een bepaald stuk van het RAM-geheugen als “gereserveerd” aanduidt. Linux zal doen alsof dit stuk geheugen niet bestaat, tot een applicatie zelf dat gereserveerde stuk wil mappen in de virtuele adresruimte. Wij kiezen ervoor onze bestandsbuffers op het einde van het fysieke RAM-geheugen te plaatsen.

7.4.3 Multithreading

De eerste thread voert in een lus blokkerende leesoperaties op het stuurprogramma. Dit betekent dat de thread blijft wachten zolang er geen data te lezen valt. Wanneer er een DMA-buffer is die ongelezen data bevat, geeft het stuurprogramma een getal terug dat de index van de ongelezen DMA-buffer bevat.

Eens de streamerapplicatie weet welke DMA-buffer nieuwe data bevat, kopieert het de inhoud van deze buffer naar een bestandsbuffer. Deze bestandsbuffer wordt sequentieel opgevuld. Op het moment dat een bestandsbuffer volledig opgevuld is, signaleert de streamer dit aan de circulaire buffer en begint het met het opvullen van de volgende bestandsbuffer.

De tweede thread gebruiken we voor het wegschrijven van opgevulde bestandsbuffers naar de harde schijf. Deze voert blokkerende leesoperaties doet op de circulaire buffer in een lus uit. Als de circulaire buffer een bestandsbuffer bevat die vol is, maar nog niet weggeschreven werd, geeft het deze bestandsbuffer terug aan de thread.

Vervolgens vragen we de kernel de inhoud van de verkregen bestandsbuffer weg te schrijven naar de harde schijf. Als dit voltooid is, melden we aan de circulaire buffer dat een bestandsbuffer met succes opgeslagen werd en dat deze kan overschreven worden.

7.4.4 Het streamen beëindigen

Sectie 7.4.1 maakt duidelijk dat de streamingapplicatie uit twee threads bestaat die elk in een lus belanden. Aangezien we het streamproces vroeg of laat willen beëindigen, moeten we

voorzien in een mechanisme dat het afbreken van de lussen mogelijk maakt.

De allerlaatste DMA-transfer wordt geïnitieerd door de gebruiker nadat die laatste de sampler uitgeschakeld heeft. De gebruiker doet dit door aan de controle-applicatie het **flush**-commando mee te geven, zie ook Sectie 7.2.6. Wanneer het stuurprogramma dit flush-commando ontvangt, zet het een vlag hoog die aangeeft dat het laatst geschreven DMA-blok de allerlaatste buffer is. We gebruiken deze vlag in de streamingapplicatie om de lus van de leesthread te beëindigen.

Telkens een DMA-buffer uitgelezen werd in de leesthread van de streamingapplicatie, moet het stuurprogramma hiervan op de hoogte gesteld worden. Hiervoor zenden we een commando naar de kernel. We kunnen zorgen dat het stuurprogramma op dit commando reageert door de “allerlaatste buffer”-vlag terug te sturen naar de streamer.

Wanneer de leesthread ziet dat de laatst gelezen DMA-buffer de allerlaatste buffer is die gegenereerd werd, wordt de lus beëindigd. We stellen eveneens de circulaire buffer op de hoogte van het feit dat dit de laatste samples zijn. Hoe ik dit geïmplementeerd heb, bespreken we in de volgende sectie.

7.4.5 Implementatiедetails

We hebben ervoor gekozen de streamingapplicatie te schrijven in C++, waarbij we gebruik gemaakt hebben van de nieuwste standaard C++11.

Threading Één van de nieuwe mogelijkheden van de C++11, is ondersteuning voor threading in de STL. Het gebruik van verschillende threads met gedeeld geheugen, gaat onvermijdelijk gepaard met *mutuele uitsluiting* (Eng. *mutual exclusion*). Ook hier biedt C++11 nu ondersteuning voor verschillende synchronisatiemechanismen in de STL [29].

Circulaire buffer Het geheugen dat gedeeld wordt tussen de twee threads, is een circulaire buffer die de verschillende bestandsbuffers (slots) bevat. We hebben zelf een object geïmplementeerd dat zich gedraagt als een circulaire buffer. Hierbij hebben we speciale aandacht besteed aan de noodzaak dat twee threads hetzelfde object moeten kunnen aanspreken. We voorzien in de volgende methoden:

- een methode om het eerstvolgende vrije slot op te vragen;
- een methode om het eerstvolgende bezette slot op te vragen;
- een methode om te signaleren dat een slot werd opgevuld;
- een methode om te signaleren dat een slot werd verwerkt.

Beëindigen van het streamen Om uit te maken wanneer het streamen beëindigd moet worden, wordt er een signaal vanuit het stuurprogramma naar de leesthread verstuurd. Deze laatste geeft dit door aan de circulaire buffer door het aantal geschreven bytes in het laatste slot mee te geven. De achterliggende filosofie is de volgende.

Als de sampler aan staat en er wordt data gegenereerd, dan vult de leesthread één van de slots in de circulaire buffer op. Wanneer zo'n slot volledig opgevuld is, signaleert de leesthread dit

aan de circulaire buffer. Deze laatste houdt bij hoeveel bytes er geschreven werden naar het laatste slot. Zolang het een volledig opgevuld slot betreft en de gebruiker het flushcommando nog niet gegeven heeft, slaan we hier nul op.

Bij het kopiëren van de laatste DMA-buffer naar een bestandsbuffer kan het zijn dat die bestandsbuffer nog ver van opgevuld is. We delen aan de circulaire buffer mee hoeveel data er in de laatste bestandsbuffer staat. In tegenstelling tot de situatie in regime, slaat de circulaire buffer nu een getal op dat verschillend is van nul.

Wanneer de schrijfthread een volle bestandsbuffer vraagt aan de circulaire buffer, krijgt deze ook te horen hoeveel nuttige bytes er in die bestandsbuffer zitten. Is dit getal verschillend van nul, dan betekent dit dat de zonet opgevraagde bestandsbuffer de laatste is. De lus van de schrijfthread kan afgebroken worden.

7.5 Verwerken van de gestreamde samples

Om de gestreamde data te verwerken, hebben we een eenvoudige parser geschreven. De dataverloopgraaf van dit programma is te zien in Figuur 7.5.

We lezen de ruwe samples één voor één uit, waarna we de data, carrybit en tijdsstempel scheiden. Vervolgens kijken we na of de gelezen data wel geldig is. Een sample moet aan de volgende twee voorwaarden voldoen om geldig te zijn:

1. De tijdsstempel moet strikt groter zijn dan de tijdsstempel van het vorige sample;
2. De carrybit mag enkel 1 zijn als de tijdsstempel gelijk is aan 0.

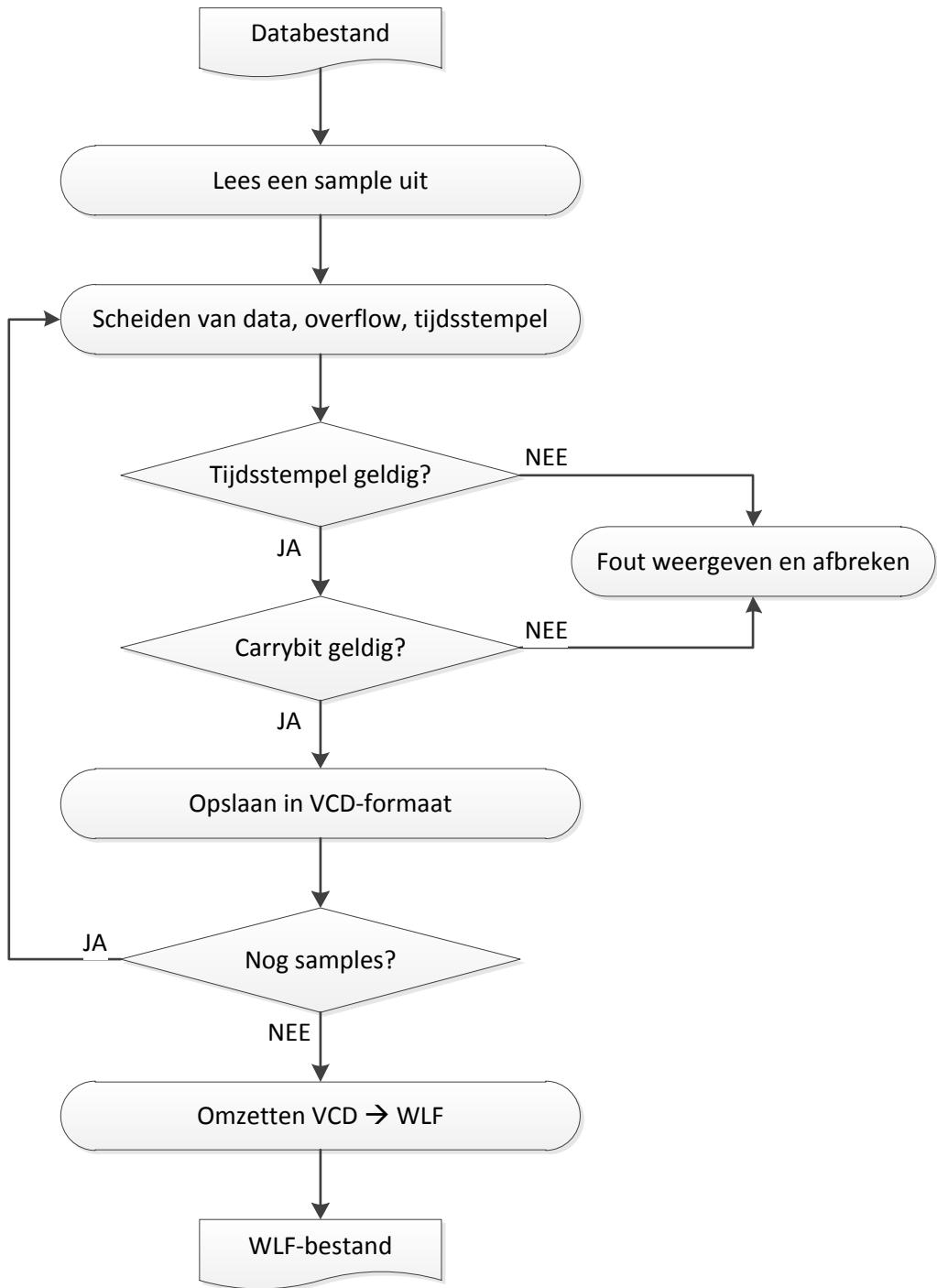
Is één van deze voorwaarden niet voldaan, dan is er een fout opgetreden bij het verkrijgen van de data. We beschouwen het bestand dan als onbetrouwbaar.

Hebben we een geldig sample, dan slaan we dit samen met de eerder verwerkte samples op in een Value Change Dump (VCD)-bestand. Wanneer alle samples verwerkt zijn, genereren we een Wave Log Format (WLF)-bestand. Waarom we deze bestandstypes gebruiken, wordt duidelijk in de volgende Sectie.

7.6 De visualisatie

In eerste instantie hadden we een eigen applicatie geschreven om de gestreamde data te visualiseren. Al snel bleek echter dat het werken met grote bestanden problemen zou opleveren, en dat het ontwikkelen van gereedschappen om tijd te meten (bv. cursors) teveel tijd zou innemen. We hebben daarom besloten de eigen applicatie te laten vallen, en te kiezen voor een bestaande oplossing, ModelSim.

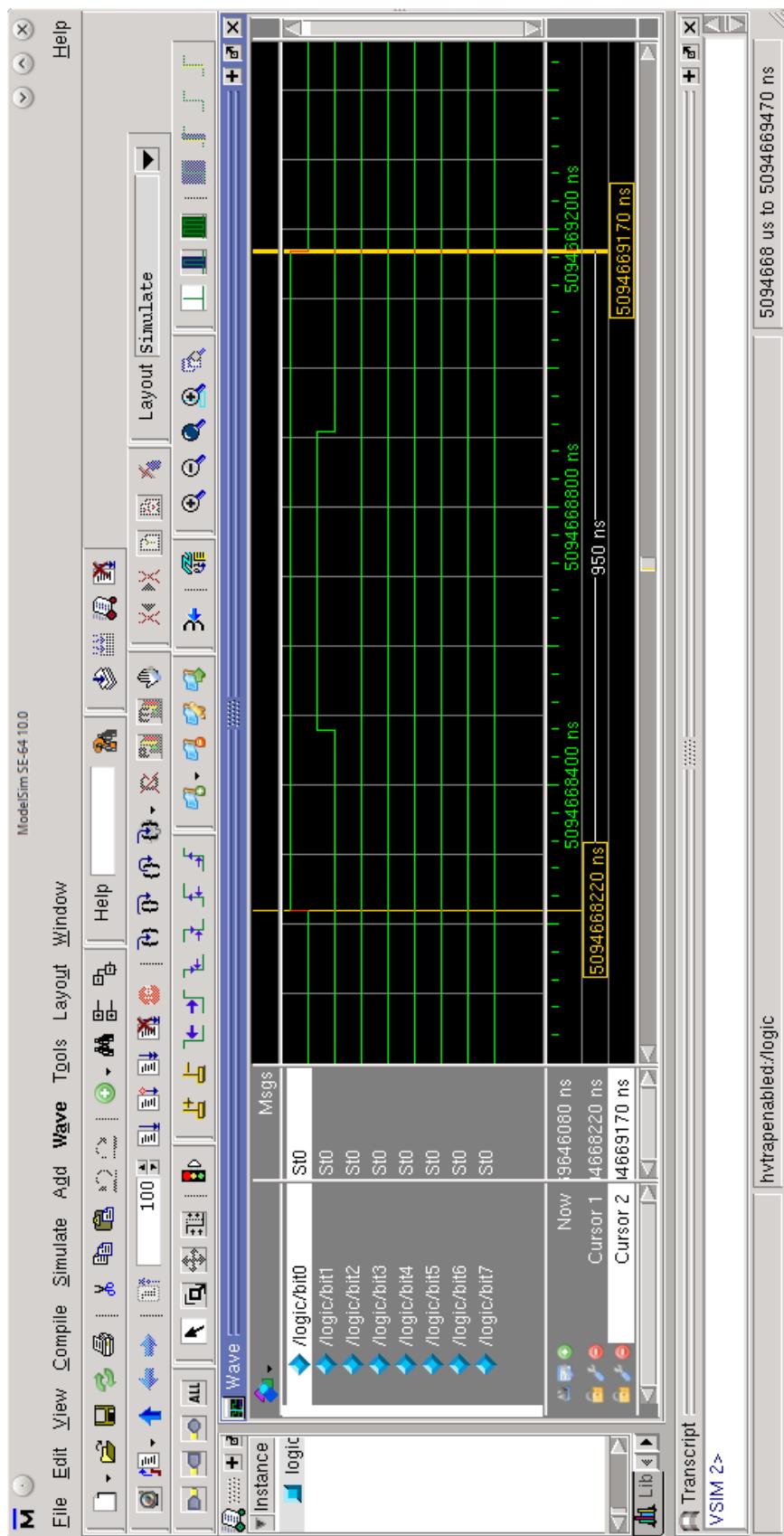
In principe is ModelSim bedoeld om VHDL-code te simuleren, maar wij kunnen de tool “misbruiken” om digitale signalen te plotten. Het bestandsformaat waarmee ModelSim werkt, heet WLF. ModelSim komt met een tool die het mogelijk maakt een bestand in VCD-formaat om te zetten naar het proprietaire WLF-formaat. Dat is de reden waarom we in Sectie 7.5 kozen voor het genereren van een VCD-bestand om dit op het einde van het verwerkingsproces

**Figuur 7.5:** Programmaflow voor het verwerken van de samples.

om te zetten in een WLF-bestand. Het VCD-bestandsformaat is niet bijster ingewikkeld, en is bijgevolg eenvoudig te implementeren [3, 2].

Figuur 7.6 geeft de interface weer die we krijgen bij het inladen van een WLF-bestand in ModelSim. De plot toont de 8 GPIO-signalen. De tijd tussen twee events is te meten aan de hand van *ursors*, de verticale lijnen op Figuur 7.6. De tijdstippen waarop de cursors staan, worden onder de plot omkaderd weergegeven. Het tijdsinterval tussen deze cursors is eveneens onder de plot terug te vinden.

Bij het opslaan van de smaples in het VCD-formaat, stoten we echter op een probleem. Zoals besproken in Sectie 6.7, werkt de sampler op een frequentie van 50 MHz. Dit komt overeen met een periode van 20 ns. Het VCD-bestandsformaat laat deze tijdseenheid echter niet toe [3]. We gebruiken daarom een eenheid die het gemakkelijkste met onze sampleperiode te relateren is: 10 ns. Dit impliceert dat één kloktik van de sampler overeenkomt met twee eenheden in het VCD-bestand. We compenseren deze afwijking door de tijdsstempels van de samples te verdubbelen vooraleer we ze wegschrijven naar het VCD-bestand.



Figuur 7.6: Interface van ModelSim.

Hoofdstuk 8

Verificatie

In dit hoofdstuk leggen we uit op welke manier we de ontwikkelde oplossing getest hebben en onderzoeken we wat het traagste onderdeel is van het systeem. We sluiten af met een overzicht van de gebruikte bronnen op de FPGA.

8.1 Verificatie van het ontwerp

Tijdens de ontwikkeling hebben we de werking van elk onderdeel apart geverifieerd. De verificatie van de sampler en de geheugencontroller hebben we uitgevoerd met ModelSim. Om de leesfunctionaliteit van de geheugencontroller te testen, hadden we nood aan een VHDL-model van het SRAM-geheugen dat zich op het FPGA-bord bevindt. Een simulatiemodel van een gelijkaardige chip was beschikbaar op de website van Berkeley University [33].

De communicatie tussen het FPGA-bord en de PC hebben we geverifieerd door op de FPGA een teller te implementeren die de ingang van de sampler simuleert. Omdat we de volgorde van gegenereerde samples op voorhand kennen, kunnen we verifiëren of alle data goed op de hostcomputer wordt opgeslagen.

Het interfacebordje tussen het testplatform en het FPGA-bord hebben we uitvoerig getest door gebruik te maken van multimeters, een oscilloscoop en een signaalgenerator. Een beschrijving van de uitgevoerde testen evenals de meetresultaten zijn terug te vinden in Bijlage A.

Een belangrijke parameter van het ontwerp is de maximale frequentie waaraan de ingangen van de sampler mogen veranderen opdat het geheel nog probleemloos werkt. Om deze parameter te bepalen, hebben we gebruik gemaakt van een functiegenerator. We verbinden een van de ingangen van de sampler met de uitgang van de functiegenerator, en genereren een blokgolf met een frequentie die we langzaam laten toenemen. De hoogste frequentie waaraan onze oplossing nog steeds probleemloos werkt, is:

$$f_{max} = 6.5 \text{ MHz} \quad (8.1)$$

Dit komt overeen met een minimumperiode van:

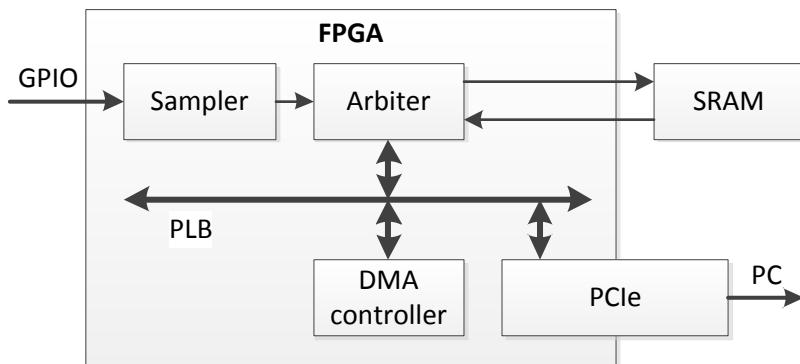
$$T_{min} = \frac{1}{f_{max}} = 154 \text{ ns} \quad (8.2)$$

8.2 De traagste component van het systeem

In de vorige sectie hebben we de maximale frequentie bepaald waaraan veranderingen op de GPIO-lijnen door onze oplossing verwerkt kunnen worden. We willen uiteraard weten welk onderdeel van het systeem de beperkende factor is. Dit doen we door het datapad, d.i. de weg die de samples afleggen, te volgen vanaf de ingang van de sampler tot op de harde schijf van de hostcomputer.

8.2.1 De FPGA

Het pad dat de samples afleggen in de FPGA is te zien in Figuur 8.1. De sampler genereert een sample wanneer een van de GPIO-signalen verandert. De arbiter schrijft deze samples weg naar het SRAM-geheugen. De arbiter leest eveneens de inhoud van het SRAM-geheugen uit om de DMA-controller van data te voorzien. Deze DMA-controller zendt de samples tenslotte via PCIe naar de PC.



Figuur 8.1: Het datapad in de FPGA.

Zoals besproken in Sectie 6.4.2 maken we binnen de FPGA gebruik van drie klokdomainen: de sampler, de arbiter met het geheugen en de PLB-bus. De frequentie van deze klokdomainen die in realiteit gehaald wordt, hangt af van heel wat parameters. Voorbeelden zijn inherent trage componenten, beperkingen qua routering en de weg dat een signaal moet afleggen. De in realiteit haalbare maximumfrequenties van de klokdomainen halen we uit een van de rapporten die de toolchain genereert. Het rapport dat ons hier aanbelangt, heet “Post-Place and Route (PAR) Static Timing Report”. De maximale frequenties van de gesynthetiseerde klokdomainen zijn:

- 52 MHz voor het samplerdomein, waar we 50 MHz opgegeven hebben;
- 61 MHz voor het geheugendomein, waar we 60 MHz opgegeven hebben;
- 55 MHz voor het PLB-domein, waar we 100 MHz opgegeven hebben.

We zien dus dat het PLB-domein lang niet onze gevraagde frequentie haalt en dat dit hoogst-waarschijnlijk de beperkende factor van het systeem is. Uit het tijdsanalyserapport van het FPGA-ontwerp blijkt dat de FIFO van de arbiter de boosdoener is.

Uit Sectie 8.1 weten we dat de maximumfrequentie van een GPIO-singalaal gelijk is aan 6.5 MHz. Er worden bijgevolg samples gegenereerd aan 13 MHz: één sample op de stijgflank en één op de daalflank van de blokgolf. Houden we nu rekening met de grootte van een sample, 32 bits, dan bekomen we een datadebiet van 52 MB/s.

Beschouwen we nu even de regimewerking van het systeem, waarbij de arbiter afwisselend schrijft naar en leest van het SRAM-geheugen. Vermits het klokdomain van de arbiter geklokt is op 60 MHz, kunnen we aan 30 MHz samples schrijven naar en lezen van het SRAM-geheugen. Er worden dus samples klaargezet voor de DMA-controller aan een debiet van maximaal 120 MB/s.

Het bepalen van het datadebiet van de DMA-controller is moeilijker omdat de hostcomputer hier een invloed op kan hebben. Daarom meten we de snelheid waaraan de DMA-controller data kan verzenden op de PC zelf.

8.2.2 De PC

Eenmaal uit het datapad in de FPGA, leggen de samples nog een zekere weg af vooraleer ze in een bestand op de harde schijf worden opgeslagen. Willen we te weten komen hoe lang een operatie duurt, dan moeten we een of andere manier hebben om tijdsverschillen te meten. De Linux kernel voorziet in een aantal functies om de tijd te meten. Wij maken gebruik van de functie `do_gettimeofday`, waarmee tijd en datum opgevraagd kunnen worden met een nauwkeurigheid van één microseconde. Om een statistisch relevant resultaat te hebben, middelen we uit over ongeveer 50000 metingen.

Het eerste wat ons interesseert, is hoe lang één dataoverdracht van de FPGA naar het RAM-geheugen van de PC duurt. Om dit te meten, kijken we hoeveel tijd er verstrijkt tussen het initialiseren van de DMA-controller en het ontvangen van de interrupt die aangeeft dat een overdracht voltooid is. De gemiddelde duur van één dataoverdracht is 114 μ s. Rekening houdend met het feit dat een blok bestaat uit 2048 samples, bekomen we het volgende datadebiet:

$$2048 \text{ samples/blok} \cdot 4 \text{ Bytes/sample} \cdot \frac{1}{114} \text{ blok}/\mu\text{s} = 71 \text{ MB/s} \quad (8.3)$$

Nadat de samples in de DMA-buffers in het RAM-geheugen zijn gedumpt, kopieert de streaming-applicatie de samples naar een grotere buffer. Metingen van de duur van deze kopieeroperatie leren ons dat het kopiëren van één DMA-buffer 5.137 μ s in beslag neemt. Het datadebiet is dus gelijk aan:

$$8192 \text{ Bytes/blok} \cdot \frac{1}{5.137} \text{ blok}/\mu\text{s} \approx 1.60 \text{ GB/s} \quad (8.4)$$

De laatste stap die de samples moeten doorlopen, is het schrijven van een in ons geval 128 MB-buffer naar de harde schijf. Per buffer meten we een schrijftijd die gemiddeld gelijk is aan 1.493 μ s. Dit betekent dat we het volgende datadebiet hebben:

$$128 \text{ MB} \cdot \frac{1}{1.493} \text{ buffer/s} \approx 85.7 \text{ MB/s} \quad (8.5)$$

8.2.3 Besluit

Uit het tijdsanalyserapport blijkt dat de traagste verbindingen binnen de FPGA zich in de FIFO van de arbiter bevinden. Ondanks we een leesfrequentie opgegeven hebben van 100 MHz, kan de *Core Generator* blijkbaar geen FIFO genereren die aan deze frequentie gelezen kan worden. Tijdens deze thesis hebben we geen tijd meer gehad om de oorzaak van dit probleem uit te zoeken. Als de FIFO van de arbiter sneller gemaakt kan worden, is een hogere frequentie van de GPIO-signalen toegestaan.

We kunnen echter wel besluiten dat de ontwikkelde oplossing voldoende snel is om de beschouwde testplatformen te observeren. Na de PLB-bus is de harde schijf van de hostcomputer de flessenhals van het systeem. Dit betekent dat we met de huidige hardwareconfiguratie maximaal 85.7 MB/s kunnen verwerken. Dit komt overeen met een maximale GPIO-frequentie van 10.5 MHz. Door gebruik te maken van een ander opslagmedium, zoals een SSD, kunnen we deze flessenhals wegwerken.

8.3 Gebruik van de FPGA-bronnen

De toolchain die we gebruiken voor het synthetiseren van het FPGA-ontwerp genereert een aantal rapporten van de synthese. Een van deze rapporten geeft een overzicht van de gebruikte bronnen op de FPGA. Tabel 8.1 geeft een overzicht van de belangrijkste bouwblokken.

Het basisblok van een FPGA is een zogenaamd Configureerbaar Logisch Blok (CLB). Elk CLB is opgebouwd uit twee *slices* [63]. Het aantal gebruikte *slices* geeft dus een indicatie van hoeveel oppervlakte een FPGA-ontwerp nodig heeft. Deze waarde moet zo laag mogelijk zijn om veel optimalisatie tijdens de PAR mogelijk te maken.

Naast de CLBs voorziet de FPGA ook in I/O-buffers om communicatie met de buitenwereld mogelijk te maken. Het aantal gebruikte I/O-buffers geeft een indicatie van hoeveel uitbreidingsmogelijkheden er nog zijn.

Ons FPGA-ontwerp maakt gebruik van FIFO's om data te bufferen. Deze FIFO's worden on-chip gesynthetiseerd uit een aantal BRAM-primitieven. Een BRAM-primitief is een basisblok voor geheugen in de FPGA, en kan 36 kbit aan data opslaan. Het aantal gebruikte primitieven geeft een indicatie van de uitbreidingsmogelijkheden voor de FIFO's.

	Gebruikt	Totaal	
Slices	4551	17280	26%
I/O-buffers	76	640	11%
BRAM primitieven	20	148	13%
BRAM geheugen (kB)	648	5328	12%

Tabel 8.1: Gebruiksstatistieken van de FPGA

Al bij al kunnen we besluiten dat de FPGA bijna niet bezet is door onze logica. Dit impliceert dat optimalisatie naar snelheid zeker goed moet werken, omdat de router over veel ongebruikte oppervlakte beschikt. Ondanks grote hoeveelheid beschikbare oppervlakte, kan de router

de arbiter van de FIFO toch niet genoeg optimaliseren om de PLB-bus op de gevraagde werkfrequentie van 100 MHz te krijgen. Er was echter geen tijd meer over om de achterliggende reden te onderzoeken.

Hoofdstuk 9

De praktijk

In dit hoofdstuk voeren we enkele metingen uit met de ontworpen meetopstelling. Wegens tijdgebrek was er niet veel tijd meer over om veel metingen uit te voeren, en hebben we enkel metingen gedaan op het BeagleBoard. Voor we overgaan tot de metingen en de meetresultaten, bespreken we hoe de meetopstelling opgebouwd is en hoe deze gebruikt moet worden.

9.1 De meetopstelling

In de volgende secties bespreken we eerst kort hoe de meetopstelling geassembleerd moet worden; vervolgens lichten we de opstartprocedure toe.

9.1.1 Assemblage van de hardware

Voor we kunnen beginnen meten, moeten we onze oplossing assembleren. Zoals besproken in de vorige hoofdstukken, bestaat deze uit:

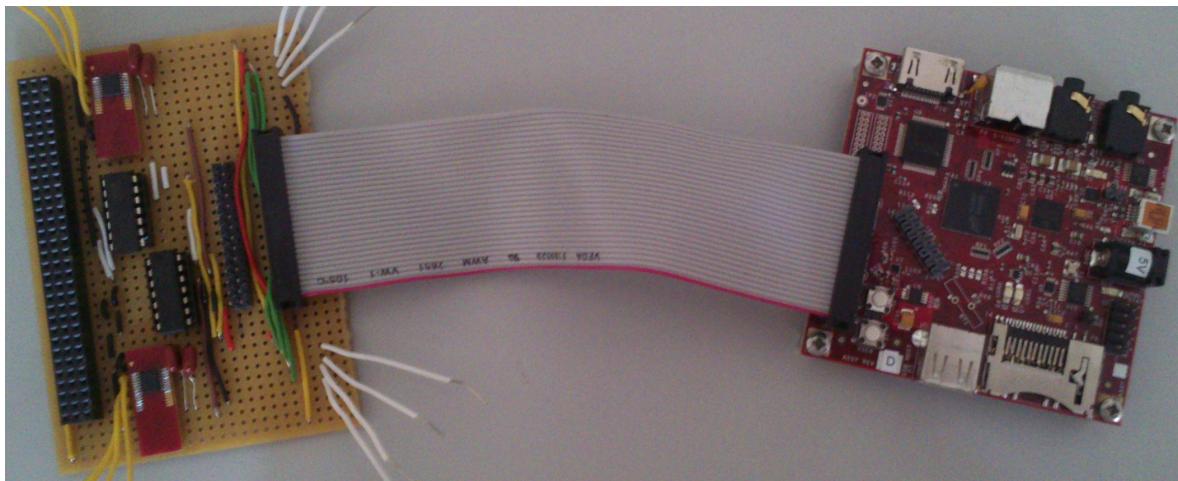
- een testbord met geïnstrumenteerde systeemsoftware;
- het eerder ontwikkelde interfacebord om het testbord te verbinden met de FPGA;
- een computer met het FPGA-bord in een vrij PCIe-slot;
- een computer met de Xilinx software om de FPGA te programmeren;

Wij maken gebruik van de ARM DSTREAM om de systeemsoftware naar het BeagleBoard te uploaden, maar in principe is geen dure debugger vereist. De software kan bijvoorbeeld ook via een SD-geheugenkaart ingeladen worden.

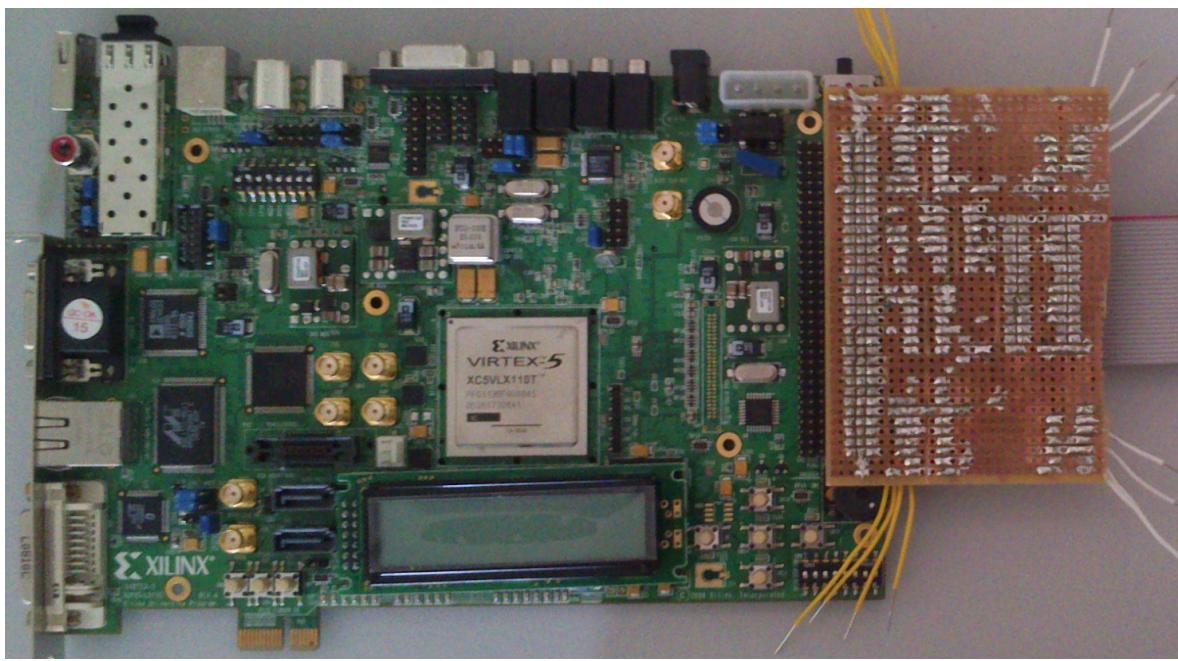
We beginnen met het verbinden van het BeagleBoard en het interfacebord. Hierbij letten we erop dat de 1.8 V voedingspin van het BeagleBoard met de juiste pin van het interfacebord wordt verbonden. Figuur 9.1 toont een correcte verbinding tussen beide bordjes.

Eens het BeagleBoard en het interfacebord verbonden zijn, kunnen we de verbinding maken met het FPGA-bord. Hoe dit gebeurt, is te zien in Figuur 9.2. We vergewissen ons ervan dat de juiste uitbreidingsconnector van het FPGA-bord verbonden is met het interfacebord.

Tenslotte kunnen we het FPGA-bord in een vrij PCIe-slot van de meetcomputer steken. Vervolgens verbinden we de programmer met het FPGA-bord zodat we de FPGA kunnen

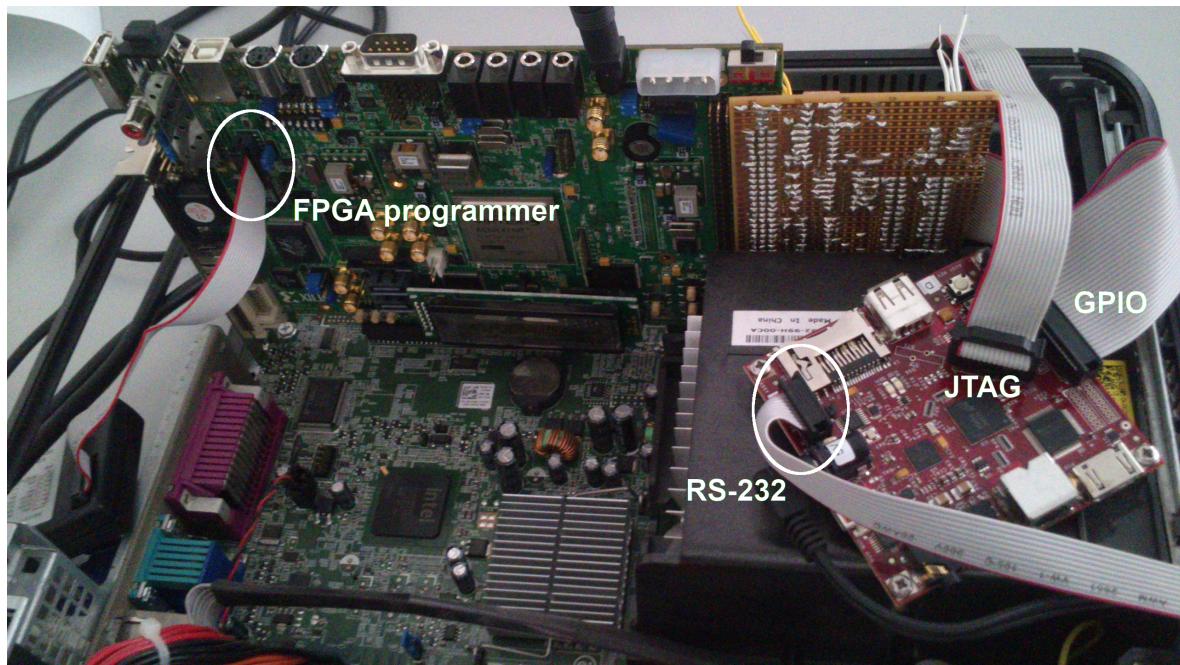


Figuur 9.1: Het interfacebord verbonden met het BeagleBoard.



Figuur 9.2: Het interfacebord verbonden met het FPGA-bord.

programmeren. We sluiten de DSTREAM aan op het BeagleBoard met behulp van een JTAG-verbinding. Optioneel kunnen we de tekstuele output van de systeemsoftware op het BeagleBoard volgen op een computer door een RS-232 seriële verbinding op te zetten. Figuur 9.3 toont de verschillende kabels.



Figuur 9.3: Assemblage van de hardware.

9.1.2 Opstartprocedure

Het eerste wat we moeten doen nadat de hardware is geassembleerd, is de FPGA programmeren. We doen dit vanop een computer waarop de Xilinx toolchain geïnstalleerd staat. Eens de FPGA geprogrammeerd is, kunnen we de meetcomputer opstarten.

Eens de meetcomputer is opgestart, moeten we het stuurprogramma, de controle-applicatie en de streamingapplicatie compileren. Dit gebeurt met het commando `make all`. Vervolgens laden we het stuurprogramma in in de kernel door het commando `insmod xpcie.ko` uit te voeren. Het stuurprogramma schrijft eventuele foutmeldingen naar het kernel logboek, dat opgevraagd kan worden met het commando `dmesg`.

Het volgende dat moet gebeuren, is het opstarten van de streamingapplicatie. De sampler op de FPGA kan in- of uitgeschakeld worden met de controle-applicatie.

Tenslotte moeten we het BeagleBoard nog programmeren met behulp van de DSTREAM, en de veranderingen op de GPIO-lijnen worden opgeslagen op de meetcomputer.

9.2 Experiment: uitvoeringstijd van een programma

Zoals aangehaald in Sectie 1.2 gebruiken we de binnen CSL ontwikkelde hypervisor om onze oplossing te testen. Een van de veel voorkomende acties die een hypervisor moet afhandelen, zijn uitzonderingen (Eng.: exceptions). De ARM-architectuur voorziet in een uitzonderingstabell, de *exception vector*, die voor elk type uitzondering een zogenaamde *handler* bijhoudt. In het geval een programma een uitzondering genereert, moet de processor de tabel met handlers uit het geheugen lezen en de gepaste functie aanroepen. Teneinde de uitvoeringstijd van programma's te minimaliseren, voorziet een processor in cachegeheugen. Dit cachegeheugen wordt beheerd door de MMU. Om het cachegeheugen te kunnen gebruiken, moeten we dus de MMU inschakelen.

Met dit experiment willen we het volgende te weten komen:

- Wat is de invloed van de locatie van de uitzonderingstabell op de uitvoeringstijd van het programma?
- Heeft het inschakelen van de MMU een grote impact op de uitvoeringstijd?

Met een “niet verplaatste uitzonderingstabell” bedoelen we dat de tabel in de directe buurt van de code staat die de uitzondering genereert. Met een “verplaatste uitzonderingstabell” bedoelen we dat de tabel ver van de genererende code in het geheugen staat.

De meetresultaten zijn te vinden in Tabel 9.1. De tabel bevat de gemeten uitvoeringstijden van het gehele programma en de *handler*. We zien dat het inschakelen van de MMU, en dus ook het cachegeheugen, een significante verbetering van de uitvoeringstijd oplevert: het programma loopt maar liefst de helft sneller!

	Zonder MMU		Met MMU		Winst door MMU	
Uitzonderingstabell...	Totaal	Handler	Totaal	Handler	Totaal	Handler
...niet verplaatst	4564 ns	3048 ns	2604 ns	1916 ns	48.88 %	53.03 %
...verplaatst	4709 ns	3194 ns	2836 ns	1943 ns	44.37 %	52.41 %
Verlies door verplaatsen	-3.18 %	-4.11 %	-8.81 %	-1.41 %		
Foutmarge	0.4 %	0.6 %	0.7 %	1 %		

Tabel 9.1: Meting van de uitvoeringstijd van een programma

Bij deze metingen moet gezegd zijn dat de resolutie van het systeem beperkt is door de frequentie van de sampler, die gelijk is aan 50 MHz. Bijgevolg is de tijdsresolutie gelijk aan 20 ns, dit bepaalt dan ook de foutmarge.

9.3 Experimenteren met het PandaBoard

Wanneer we identiek hetzelfde testprogramma zouden uitvoeren op het PandaBoard, zou dit niet de juiste resultaten opleveren. In tegenstelling tot het BeagleBoard is het PandaBoard gebaseerd op een out-of-order architectuur.

Ons testprogramma voert dezelfde code ongeveer 50000 keer uit. Wanneer we het cachegeheugen inschakelen, zouden we vanaf de tweede meting werken met data uit het cachegeheugen. Om alle metingen identiek te laten verlopen, moeten we dus het cachegeheugen legen voor elke iteratie. Omdat het PandaBoard met een out-of-order architectuur werkt, moeten we na deze *flush*-operatie zogenaamde *instruction barriers* plaatsen. Dit zorgt ervoor dat de pijplijn van de processor geleegd wordt, waardoor het programma even snel (of traag) uitvoert als de eerste keer.

Hoofdstuk 10

Conclusie

Bij de ontwikkeling van software is kennis over de uitvoeringstijd van codeblokken van belang. Voor gebruikerstoepassingen bestaan er heel wat (gratis) softwarepakketten die profiling mogelijk maken. Het profileren van systeemsoftware vereist vaak gespecialiseerde, dure, tools die op hardwareniveau kijken naar het gedrag van de systeemsoftware.

In deze thesis hebben we een oplossing ontwikkeld die op een goedkope manier profiling van systeemsoftware mogelijk maakt en die bovendien gericht is op het uitvoeren van *langdurige* metingen. Dat laatste is iets wat de meest gangbare tools niet kunnen. Onze oplossing vereist bovendien geen exotische hardware-interfaces: een aantal GPIO-lijnen zijn voldoende om systeemsoftware te kunnen profileren. Hierdoor is deze oplossing bijzonder generiek.

De uitvoering van de systeemsoftware wordt in de gaten gehouden door een FPGA, die zijn observaties doorgeeft aan een hostcomputer via PCIe. Deze laatste slaat de observaties op voor latere verwerking. We kunnen GPIO-signalen observeren met een minimumtijd tussen twee toestandsveranderingen van 77 ns. Dit komt overeen met een frequentie van 13 MHz.

10.1 Toekomstig werk

Ondanks de vele tijd die in dit project is gekropen, kunnen er nog steeds verbeteringen aangebracht worden. Wij stellen hier een aantal aanpassingen voor die de performantie van de implementatie kunnen verbeteren.

Diodes op het interfacebord Zowel het FPGA-bord als de testborden moeten beschermd worden tegen spanningspieken. Dit kan gebeuren door het toevoegen van twee diodes op elke GPIO-lijn. Dit moeten dan diodes zijn met een doorslagspanning van respectievelijk 3.3 V en 1.8 V.

De snelheid van het FPGA-ontwerp opdrijven De traagste verbindingen van het FPGA-ontwerp kunnen sneller gemaakt worden door de toolchain te vragen die verbindingen te beschouwen als kloksignalen. Op die manier zou het mogelijk moeten zijn de frequentie van de sampler, en dus ook de meetresolutie, te verhogen.

De trage FIFO van de arbiter verbeteren Uit een tijdsanalyse van het FPGA-ontwerp blijkt dat de FIFO van de arbiter de beperkende component is van het systeem. Het opvoeren van de hoogst mogelijke werkfrequentie van deze FIFO verhoogt de maximale frequentie waaraan GPIO-signalen mogen veranderen opdat het systeem nog probleemloos werkt.

Optimaliseren van de FIFO grootte Zowel de FIFO van de arbiter als die van de sampler hebben we wegens tijdgebrek niet kunnen optimaliseren naar grootte. Teneinde oppervlakte op de FPGA te besparen en het ontwerp sneller te maken, is optimalisatie van de FIFO-grootte nuttig.

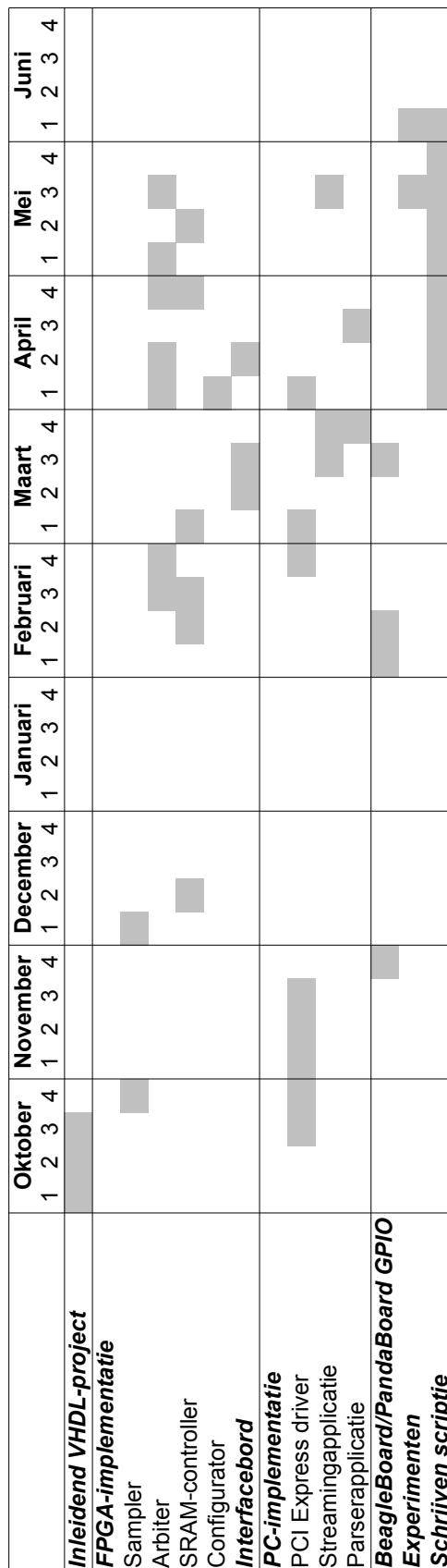
Trigger voor het starten van de sampler Momenteel moet de sampler gestart worden door een gebruikerscommando te geven vanop de meetcomputer. Door een extra GPIO-pin te gebruiken van het testbord kan de sampler automatisch gestart worden vanop een bepaalde plaats in de programmacode van de systeemsoftware.

Betere software De applicaties die ontwikkeld werden voor het controleren van de sampler en het verwerken van de samples zijn op een korte termijn geschreven. Ze zijn goed genoeg geweest voor onze metingen en testen, maar het is mogelijk dat ze nog bugs bevatten. Een grafische applicatie heeft een grote meerwaarde.

(Semi-)automatische instrumentatie Het instrumenteren van de systeemsoftware gebeurt momenteel nog manueel. Een tool ontwerpen die de systeemsoftware automatisch instrumenteert zou een mooie toevoeging zijn. Een andere mogelijkheid is het aanpassen van toolchains als GCC of clang zodat deze automatisch de GPIO-sturende code genereren.

10.2 Tijdsbesteding

We sluiten deze scriptie af met een overzicht van de tijdsbesteding van de voorbije maanden. Vermits er gedurende het jaar geen gedetailleerd logboek is bijgehouden, is deze aanduiding slechts een schatting.



Figuur 10.1: Tijdschema van deze thesis

Bijlage A

Datasheet van het interfacebord

In dit hoofdstuk geven we eerst en vooral een bedradingsschema van het interfacebord tussen de FPGA en de testbordjes; vervolgens leggen we uit hoe we het interfacebord getest hebben en wat het stroomverbruik is voor een aantal realistische situaties.

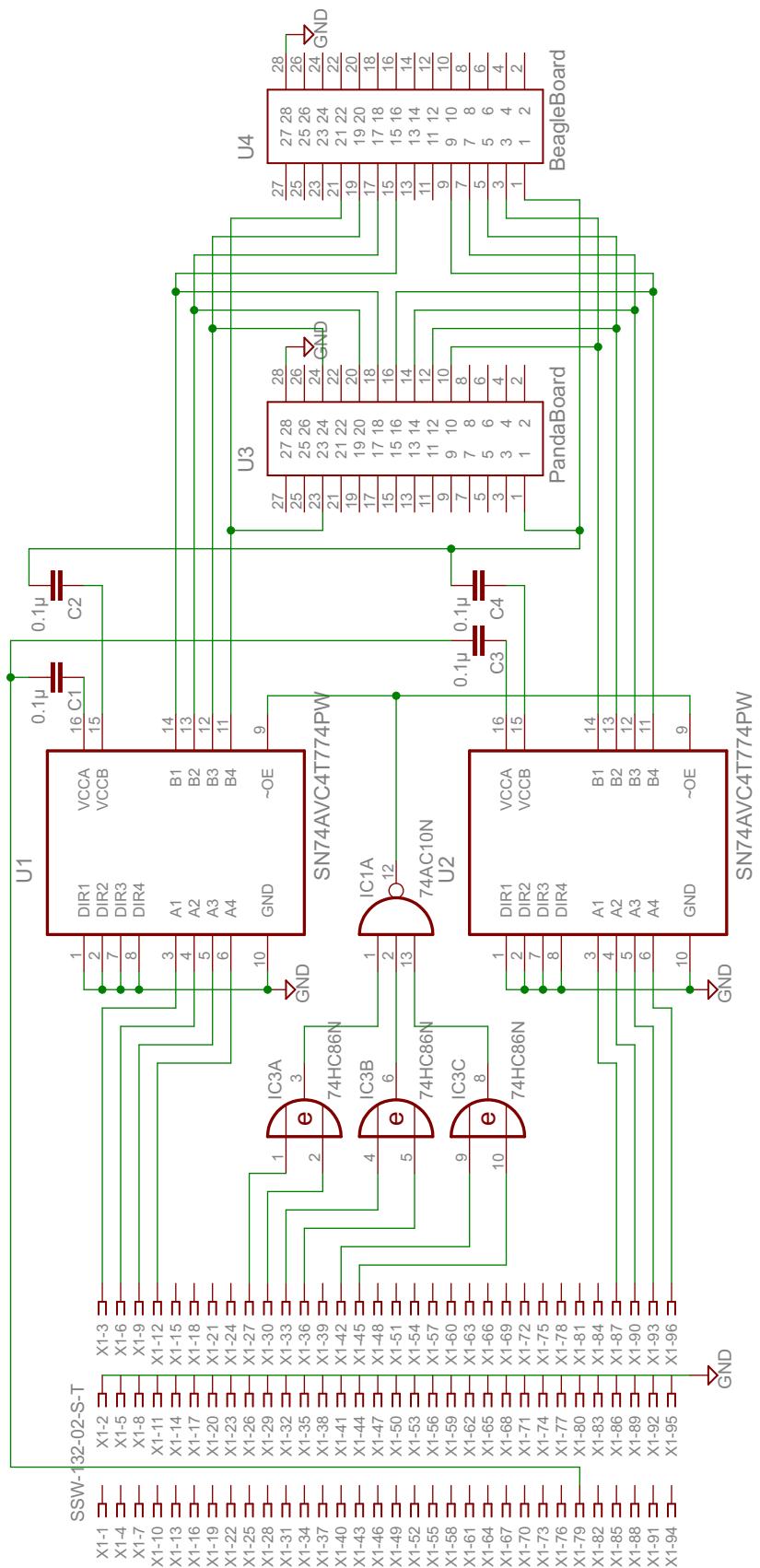
A.1 Bedradingsschema

Het bedradingsschema van het interfacebord om het BeagleBoard of PandaBoard te verbinden met het FPGA-bord is te zien in Figuur A.1. De level shifter IC's staan centraal, met ontkoppelde voedingen. De isolatielogica bevindt zich links van de level shifters, samen met de connector voor het FPGA-bord. Rechts van de level shifters vinden we de connectoren terug voor het PandaBoard en het BeagleBoard.

A.2 Verificatie en testen

Om er zeker van te zijn dat het interfacebord niet teveel stroom onttrekt uit zowel het FPGA-bord als het testbord, voeren we een aantal metingen uit. Hierbij gebruiken we twee spanningsbronnen om het interfacebord te voorzien van 1.8 V en 3.3 V. We gebruiken een functiegenerator om de GPIO-signalen aan te drijven met een blokgolf en een oscilloscoop om het vertaalde signaal te bekijken. Twee multimeters meten de stroom die onttrokken wordt uit beide spanningsbronnen.

Bij het uitvoeren van de metingen variëren we de frequentie van de blokgolf en het aantal aangedreven GPIO-pinnen. We onderzoeken eveneens wat er gebeurt als de GPIO-signalen worden aangedreven, maar het FPGA-bord uitgeschakeld is ($V_{3.3} = 0$). De resultaten van deze metingen zijn terug te vinden in Tabel A.1. We besluiten dat het stroomverbruik van het interfacebord voldoende laag is om een veilige verbinding te maken tussen het BeagleBoard of PandaBoard en het FPGA-bord.



Figuur A.1: Bedradingsschema van het interfacebord.

Frequentie blokgolf [MHz]	# GPIO	$ I_{3.3} $ [μ A]	$ I_{1.8} $ [μ A]	$ I_{1.8} $ ($V_{3.3} = 0$) [μ A]
0	1	1.5	0	0
	2	1.5	0	0
	3	1.5	0	0
	4	1.5	0	0
0.5	1	32.4	3.0	1.4
	2	61.3	4.8	2.7
	3	81.4	6.9	4.1
	4	103.9	8.1	5.6
1	1	62.4	5.9	2.8
	2	118.3	9.3	5.4
	3	156.8	13.8	8.1
	4	213	15.7	11.2
2.5	1	148.6	14.7	6.9
	2	303	21.7	12.8
	3	407	31.3	18.7
	4	523	33.2	28.7
5	1	308	28.9	14
	2	596	46.2	26.4
	3	799	58.8	35.4
	4	1024	56.2	43.8
10	1	611	53.5	27.1
	2	1172	76.9	48.3
	3	1567	103.9	70.1
	4	2140	112.7	88.5

Tabel A.1: Verificatie van het interfacebord: meetresultaten.

A.3 Gebruik

Bij het gebruik van het interfacebord moet gelet worden op de volgende dingen:

1. In geen geval mogen zowel het PandaBoard als het BeagleBoard gelijktijdig aangesloten zijn, vermits de GPIO-signalen van beide bordjes kortgesloten worden.
2. De FPGA mag enkel en alleen de isolatie uitschakelen door een patroon “010101” of “101010” aan de XOR-poorten aan te leggen als de GPIO-pinnen geconfigureerd zijn als ingang.

Bijlage B

Maximale GPIO-togglefrequentie

Om informatie door te geven van de testbordjes naar het FPGA-bord maken we gebruik van GPIO-pinnen. Teneinde enkele ontwerpscriteria vast te leggen hebben we gekeken naar de maximale frequentie waarmee deze GPIO-pinnen aangestuurd kunnen worden. Hiertoe hebben we een programma geschreven dat in een oneindige lus de toestand van de GPIO-pinnen wisselt tussen 0 en 1. We observeren het signaal op de GPIO-pinnen met een oscilloscoop.

Bij het BeagleBoard meten we een GPIO-schakelfrequentie van 2.2 MHz. Houden we hierbij rekening met de klokfrequentie van de OMAP-processor, 500 MHz, dan zien we een significant verschil. Hetzelfde fenomeen observeren we bij het PandaBoard, al is de GPIO-schakelfrequentie hier gelijk aan 6.3 MHz. De processor van het PandaBoard is geklokt op 1 GHz.

In een poging meer te weten te komen over dit verschijnsel, hebben we de klokdistributie binnen de OMAP-processor van het BeagleBoard bestudeerd [51, 49, 50]. Hoewel we veel hebben bijgeleerd over de interne werking van de processor, heeft dit onderzoek geen nuttige bijdrage geleverd tot deze thesis. We zijn er ondanks alles niet in geslaagd de GPIO-schakelfrequentie op te drijven.

Bibliografie

- [1] USB.org - Hi-Speed FAQ. <http://www.usb.org/developers/usb20/faq20/>.
- [2] Value Change Dump. http://en.wikipedia.org/wiki/Value_change_dump.
- [3] Value Change Dump VCD. <http://www.beyondttl.com/vcd.php>.
- [4] Xilinx Virtex-5 Family FPGAs, 2006.
- [5] Linux kernel parameters. <http://www.cyberciti.biz/howto/question/static/linux-kernel-parameters.php>, Januari 2009.
- [6] Zippy 2 - Ethernet Combo Board Schematic, Oktober 2009.
- [7] Trainer Board Schematic, Maart 2011.
- [8] Introduction To RS232 Serial Communication. <http://www.wcsnet.com/Tutorials/SerialComm/Page1.htm>, Juli 2012.
- [9] XUPV5-LX110T. <http://www.xilinx.com/univ/xupv5-lx110t.htm>, oktober 2012.
- [10] 16801A 34-Channel Portable Logic Analyzer. <http://www.home.agilent.com/en/pc-1000001966%3Aepsg%3Apgr/logic-analyzer?pm=SC&nid=-536902443.0&cc=BE&lc=dut>, Mei 2013.
- [11] ARIES - ADAPTOR, SSOP/DIP, 20. <http://be.farnell.com/aries/20-351000-11-rc/adaptor-ssop-dip-20/dp/1674755>, April 2013.
- [12] ARM DSTREAM High Performance Debug & Trace. <http://www.arm.com/products/tools/software-tools/ds-5/arm-dstream-high-performance-debug-trace.php>, Mei 2013.
- [13] BUNDs-KT-3PE00 - ARM - BUNDLE, DSTREAM / DS-5, PRO, NODE. <http://be.farnell.com/arm/bunds-kt-3pe00/bundle-dstream-ds-5-pro-node-locked/dp/2309929>, Mei 2013.
- [14] Compare Specifications. <http://www.tek.com/logic-analyzer/comparison/796086%2C796093>, Mei 2013.
- [15] DS-5 Debugger: Trace. <http://www.arm.com/products/tools/software-tools/ds-5/debugger/trace.php>, Mei 2013.

- [16] DSTRM-KT-0181A - ARM - DEBUG -TRACE UNIT, ARM, LINUX. http://be.farnell.com/arm/dstrm-kt-0181a/debug-trace-unit-arm-linux-platform/dp/2309950?in_merch>New%20Products, Mei 2013.
- [17] Farnell België. be.farnell.com, Juni 2013.
- [18] GlowCode. <http://www.glowcode.com/>, Juni 2013.
- [19] GNU gprof. <http://sourceware.org/binutils/docs/gprof/>, Juni 2013.
- [20] Introducing Saleae Logic16. Flexible sample rates, lots of channels. <http://www.saleae.com/logic16>, Mei 2013.
- [21] Lauterbach TRACE32. <http://www.testequipmentconnection.com/56993/Lauterbach-TRACE32.php>, Mei 2013.
- [22] List of performance analysis tools. http://en.wikipedia.org/wiki/List_of_performance_analysis_tools, April 2013.
- [23] Logic analyzer. http://en.wikipedia.org/wiki/Logic_analyzer, Mei 2013.
- [24] LOGICSTUDIO 16 Teledyne LeCroy. http://www.digikey.com/product-search/en?WT.z_header=search_go&lang=en&site=us&keywords=logicstudio&x=0&y=0, Mei 2013.
- [25] Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page, Februari 2013.
- [26] Teledyne LeCroy - LogicStudio - Logic Analyzer. <http://teledynelecroy.com/logicstudio/logicstudio.aspx>, Mei 2013.
- [27] Test and Measurement Equipment. <http://www.tek.com/>, Mei 2013.
- [28] The Xen Project, the powerful open source industry standard for virtualization. <http://www.xenproject.org/>, Mei 2013.
- [29] Thread support library. <http://en.cppreference.com/w/cpp/thread>, Mei 2013.
- [30] Windows Performance Toolkit (WPT). <http://msdn.microsoft.com/en-us/performance/cc825801.aspx>, Juni 2013.
- [31] BEAGLEBOARD.ORG. *BeagleBoard Schematic*, c4b ed., December 2009.
- [32] BEAGLEBOARD.ORG. *BeagleBoard System Reference Manual*, revision c4 ed., December 2009.
- [33] BRODERSEN, B. CS152: Computer Architecture. <http://bwrcs.eecs.berkeley.edu/Classes/cs152/projects/lab6.htm>, April 2002.
- [34] COHEN, W. E. Tuning Programs with OProfile, 2004.
- [35] CORBET, J., KROAH-HARTMAN, G., AND RUBINI, A. Linux device drivers, 3rd edition. <http://www.makelinux.net/ldd3/>, Februari 2005.
- [36] GASCHET, C. Understanding Setup and Hold Times, One Key to Successful Designs. Tech. rep., Xilinx, 2000.

- [37] ISSI. *ISSI 256Kx36 and 512Kx18 9Mb, Pipeline 'No Wait' state bus SRAM*, a ed., May 2005.
- [38] NUMONYX. *Numonyx StrataFlash Embedded Memory (P30)*, November 2007.
- [39] PANDABOARD.ORG. *OMAP4 PandaBoard System Reference Manual*, revision 0.6 ed., November 2010.
- [40] PANDABOARD.ORG. *OMAP4430 Panda Board, 8-Layer Schematic*, b1 ed., November 2010.
- [41] PANDABOARD.ORG. *OMAP4460 Panda Board, 8-Layer Schematic*, b ed., Oktober 2011.
- [42] PANDABOARD.ORG. *OMAP4460 Pandaboard ES System Reference Manual*, revision 0.1 ed., September 2011.
- [43] PCIDATABASE.COM. http://www.pcidatabase.com/vendor_details.php?id=624.
- [44] PENNEMAN, N. Een vernieuwd samplersysteem voor het evalueren en benchmarken van (RT)OS. Master's thesis, Vrije Universiteit Brussel, 2008-2009.
- [45] PENNEMAN, N., PERNEEL, L., TIMMERMAN, M., AND DE SUTTER, B. An FPGA-based Real-Time Event Sampler, 2010.
- [46] SHETTY, A. http://testbench.in/introduction_to_pci_express.html.
- [47] STANCEVIC, D. Zero Copy I: User-Mode Perspective. <http://www.linuxjournal.com/article/6345>, Januari 2003.
- [48] TEXAS INSTRUMENTS. *4-bit Dual-Supply Bus Transceiver with Configurable Voltage Translation and 3-state Outputs*, Mei 2008.
- [49] TEXAS INSTRUMENTS. *OMAP3530/25 Applications Processor*, Oktober 2009.
- [50] TEXAS INSTRUMENTS. *TPS65950 OMAP Power Management and System Companion Device Silicon Revision 1.2 Technical Reference Manual*, version g ed., December 2010.
- [51] TEXAS INSTRUMENTS. *OMAP35x Applications Processor Technical Reference Manual*, Juni 2012.
- [52] TEXAS INSTRUMENTS. *OMAP4430 Multimedia Device Silicon Revision 2.x Technical Reference Manual*, version ae ed., Mei 2012.
- [53] XILINX. *ML50X Board Bill Of Materials, ROHS Compliant*, Maart 2008.
- [54] XILINX. *Xilinx ML505/6/7 Schematics*, a02 ed., Januari 2008.
- [55] XILINX. *Xilinx XAPP1030 Reference System: PLBv46 Endpoint Bridge for PCI Express in a ML505 Embedded Development Platform*, Application Note, april 2008.
- [56] XILINX. *LogiCORE IP Endpoint Block Plus v1.14 for PCI Express*, User Guide, april 2010.
- [57] XILINX. *LogiCORE IP Processor Local Bus (PLB) v4.6 (v1.05a)*, 1.3 ed., September 2010.

- [58] XILINX. *LogiCORE IP XPS Central DMA Controller (v2.03a)*, December 2010.
- [59] XILINX. *LogiCORE IP PLBv46 RC/EP Bridge for PCI Express (v4.07.a)*, Juni 2011.
- [60] XILINX. *ML505/ML506/ML507 Evaluation Platform User Guide*, v3.1.2 ed., May 2011.
- [61] XILINX. *Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs, User Guide*, Juni 2011.
- [62] XILINX. *LogiCORE IP FIFO Generator v9.2*, 1.0 ed., Juli 2012.
- [63] XILINX. *Virtex-5 FPGA User Guide*, 5.4 ed., Maart 2012.
- [64] XILINX. *Xilinx Platform Specification Format Reference Manual*, April 2012. p.68-72.

Lijst van figuren

3.1 Blokschema van de meetopstelling.	8
3.2 Gebruikte testborden (niet op schaal)	9
3.3 Blokschema van het testplatform.	9
3.4 Het observatieplatform: een XUPV5 bord.	11
3.5 Blokschema van het observatieplatform.	12
3.6 Blokschema van het meetplatform.	15
3.7 Top-level ontwerp van het gehele systeem.	16
4.1 Blokschema van de level shifter.	20
4.2 Foto van het interfacebord.	21
5.1 Grafische voorstelling van de PCIe-architectuur.	27
5.2 Schematische voorstelling van een PCIe-verbinding.	28
6.1 Blokschema van de FPGA-implementatie.	32
6.2 Blokschema van de sampler.	34
6.3 FPGA-ontwerp van de sampler.	35
6.4 Bitlayout van een sample.	35
6.5 Tristate buffer om een bidirectionele bus te implementeren.	40
6.6 Structuur van een custom peripheral.	42
6.7 Suggestie van Xilinx om een tristate buffer te synthetiseren.	43
6.8 Implementatie van een bidirectionele bus.	43
6.9 Top-level schema van de arbiter.	44
6.10 Blokschema van de arbiter.	46
6.11 Bitlayout van het controleregister van de arbiter.	48
6.12 Bitlayout van het FIFO uitgangsregister van de arbiter.	48
6.13 Toestandsdiagram voor de communicatie tussen de FPGA en de PC.	51
6.14 Bitlayout van het BCR.	55
6.15 Optimalisatie van het FPGA-ontwerp: verlagen van de fanout.	58
6.16 Optimalisatie van het FPGA-ontwerp: ontdubbeling van het resetsignaal.	58
7.1 Blokschema van de PC-implementatie.	61
7.2 Schematisch overzicht van de streaming applicatie.	66
7.3 Overzicht van de gebruikte mapping, userspace en kernelspace.	67
7.4 Hiërarchie van de bestandsbuffers.	68
7.5 Programmaflow voor het verwerken van de samples.	71
7.6 Interface van ModelSim.	73

LIJST VAN FIGUREN

96

8.1 Het datapad in de FPGA.	75
9.1 Het interfacebord verbonden met het BeagleBoard.	80
9.2 Het interfacebord verbonden met het FPGA-bord.	80
9.3 Assemblage van de hardware.	81
10.1 Tijdschema van deze thesis	86
A.1 Bedradingsschema van het interfacebord.	88

Lijst van tabellen

2.1	Specificaties van enkele logic analyzers.	7
3.1	Vergelijking van de verbindingsmogelijkheden FPGA-PC	13
3.2	Schatting van de kosten voor dit project	16
4.1	Overzicht van de gebruikte GPIO-pinnen van het BeagleBoard.	22
4.2	Geheugenadressen van de pad configuratie registers voor het BeagleBoard.	23
4.3	Geheugenadressen GPIO-configuratieregisters voor het BeagleBoard.	23
4.4	Overzicht van de gebruikte GPIO-pinnen van het PandaBoard.	24
4.5	Geheugenadressen van de pad configuratie registers voor het PandaBoard.	25
4.6	Geheugenadressen GPIO-configuratieregisters voor het PandaBoard.	25
4.7	Overzicht van de gebruikte uitbreidingspinnen van het FPGA-bord.	26
5.1	Configuration space van een PCIe-kaart [35].	30
6.1	Adresruimte van de PLB-bus.	33
6.2	Overzicht van de in- en uitgangssignalen van de geïmplementeerde sampler.	35
6.3	Overzicht van de in- en uitgangssignalen van de SRAM-controller: interface met de arbiter.	37
6.4	Overzicht van de in- en uitgangssignalen van de SRAM-controller: interface met de geheugenchip.	39
6.5	Registerbank van de arbiter.	48
6.6	Bitvelden van het controleregister van de arbiter.	49
6.7	Parameters van de PCIe-core.	55
8.1	Gebruiksstatistieken van de FPGA	77
9.1	Meting van de uitvoeringstijd van een programma	82
A.1	Verificatie van het interfacebord: meetresultaten.	89

