Universiteit Antwerpen

Faculteit Wetenschappen
Departement Fysica
Onderzoeksgroep Elementaire Deeltjesfysica

Academiejaar 2012–2013

Implementatie van een pulse height analyser op een FPGA met behulp van high level synthesis tools

Simon Kuitenbrouwer

Promotor: Prof. dr. Nick Van Remortel
Copromotor: Ir. Wim Beaumont

Proefschrift tot het behalen van de graad van
Master in de Fysica

Dankwoord

Vooraleer we de aandacht op dit werk richten zou ik een aantal mensen zeer hartelijk willen bedanken. Als eerste wil ik mijn promotor Prof. Nick Van Remortel bedanken. Primair natuurlijk voor het aandragen van dit thesisonderwerp. Maar ook vooral voor de heldere en motiverende manier van begeleiden en voor alle kansen die hij –naast het uitvoeren van dit thesisproject– voor mij heeft gecreëerd.

Voort zou ik ook graag Ir. Wim Beaumont bedanken voor alle begeleiding en ondersteuning. Uit onze talloze gesprekken zijn veel ideëen voortgekomen die in deze thesis toegepast zijn. Ook zijn nuchtere kijk op een ontwikkelingsproces heeft me veel bijgebracht over de pragmatische aanpak die vaak nodig is.

Een derde persoon die niet mag ontbreken in dit dankwoord is Erik De Langhe. Ik bedank hem graag voor de technische ondersteuning. Hij stond altijd klaar met oplossingen en antwoorden bij problemen met de experimentele opstelling.

Ook zou ik de UA, het departement Fysica, en in het speciaal de onderzoeksgroep EDF willen bedanken voor de vele mogelijkheden en kansen die die elke individuele student Fysica krijgt. Experimenteel materiaal, van aankoop van een nieuwe FPGA, tot de gammaspectrometeropstelling, werd allemaal vlot voor mij beschikbaar gemaakt. En daarbovenop kreeg ik ook nog een eigen werkstek waar ik gedurende twee jaar aan dit project kon werken.

Als laatste zou ik mijn ouders willen bedanken omdat ze mij niet alleen de mogelijkheid hebben gegeven deze fantastische studies te doen, maar ze mij daar ook altijd en onvoorwaardelijk in gesteund hebben.

Simon Kuitenbrouwer
Antwerpen ***6 Januari 2013***

Samenvatting

In dit proefschrift hebben we het gebruik van een High Level Synthesis tool voor een FPGA applicatie onderzocht. Met als doelstelling te evalueren hoe toegankelijk de FPGA technologie met deze tools wordt voor wetenschappers die geen sterke achtergrond in de elektronica hebben. Om dit te onderzoeken is een testcase met bestaande gammaspectroscopie hardware opgezet. Hierin krijgt de FPGA de opdracht om de taak van de multi channel analyser over te nemen. Om dit te bereiken werd op de FPGA een niet-triviaal Pulse Height Analysis algoritme geïmplementeerd dat via fits aan de signalen van gedetecteerde gammastraling de energieën van deze straling afschat.

Door de beperkingen van FPGA hardware op de vlakken van complexiteit en flexibiliteit diende de noodzaak voor een tweeledig ontwikkelingsproces zich aan: Het eerste zwaartepunt van ons ontwikkelingsstrategie is het streven naar mathematische eenvoud van fixed point berekeningen. Het tweede zwaartepunt van het ontwikkelingsproces ligt op de computersimulaties die we gebruikt hebben om ons pulse height analysis algoritme te optimaliseren.

Aan de hand van deze eenvoudige strategieën zijn we erin geslaagd een werkend systeem op te zetten dat de performantie van een commerciele gammaspectrometer benadert.

Abstract

This thesis presents a study on the application of a High Level Synthesis tool for FPGA programming. Our primary goal is to evaluate how accessible the FPGA technology becomes, by applying these tools, for researchers without a substantial background in electronics. To study this, a test case with existing gammaspectroscopy hardware was created, wherein the FPGA has to perform the role of a multi channel analyser. In order to achieve this a non-trivial Pulse Height Analysis algorithm, which fits the signals of detected gamma rays in order to acquire their energy, was implemented on the FPGA.

The limitations of the FPGA hardware to handle complexity and flexibility imposed the necessity for a dual development process. The first focus in our development strategy is the endeavour towards mathematical simplicity of fixed point calculations. The second focus in the development process lies into the computersimulations we used to optimise the pulse height analysis algorithm.

By means of these two simple strategies, we achieved to set up a working system with an operational performance close to an existing commercially available gammaspectrometer.

Inhoudsopgave

Hoofdstuk 1

Inleiding

Wij hebben ons tot doel gesteld om een *Field Programmable Gate Array* (FPGA) te gebruiken als gereedschap om de data-acquisitie in een experiment in de deeltjesfysica uit te voeren. Dit programmeerbaar stuk hardware lijkt door zijn potentieel hoge verwerkingssnelheid en parallelle werking een veelzijdig stuk elektronica om mee te werken. Klassiek wordt een FPGA geprogrammeerd door het gebruik van een programmeertaal zoals VHDL. Tegenwoordig bestaan er echter high-level description talen die claimen dat het programmeren van een FPGA mogelijk is met weinig of geen kennis van de elektronica zelf. De nadruk wordt hierin verschoven naar het formaliseren van het probleem in plaats van het programeren van de hardware.

Gammaspectroscopie is een experiment waarbij de energie van gammadeeltjes wordt gemeten, en is een techniek die veelvuldig wordt toegepast om radioactieve bronnen te identificeren. Een typisch resultaat van gammaspectroscopie wordt verkregen door middel van een *Multi-Channel Analyser* (MCA) en weergegeven in een histogram zoals in Fig. 2.1. De data-acquisitie bij zo'n experiment is een uitdagend gegeven omdat we spreken over het verwerken van zeer korte signalen aan potentieel hoge *data rates*. Een FPGA zou dus in aanmerking kunnen komen om dit uit te voeren.

In dit proefschrift onderzoeken wij in de eerste plaats of we met een FPGA, en de high level taal LABVIEW de nodige data-acquisitie voor gammaspectroscopie kunnen uitvoeren.

In hoofdstuk 2 beginnen we met een basis gammaspectroscopie om daar vervolgens het gebruik van een FPGA als MCA in te kaderen. Vervolgens werpen we in hoofdstuk 3 een blik op de FPGA en de gammadetector die we gebruiken bij het ontwikkelen van onze MCA.

Op deze basis kunnen we ons ontwikkelingsproces funderen, dit proces wordt beschreven in verschillende hoofdstukken: Zo geeft hoofdstuk 4 een uitgebreide beschrijving van het algoritme dat we gebruiken om pulshoogtes te bepalen, gevolgd door een bespreking van de eerste algemene keuzes die de implementatie op de FPGA bepaald hebben. Hierna volgt hoofdstuk 5, waarin we aan de hand van PC simulaties met MATLAB de parameters van de fit optimaliseren voor onze gammadetector. Pas in de volgende hoofdstukken bespreken we de effectieve implementatie op de FPGA: In hoofdstuk 6 werpen we een blik op de verschillende individuele onderdelen van ons programma. In hoofdstuk 7 worden enkele belangrijke details toegelicht die bij de programmatie van een FPGA de aandacht verdienen. Om het plaatje

compleet te maken schetsen we in hoofdstuk 8 ook een complementair systeem dat op een PC zou kunnen draaien om een complete gebruikersinterface voor gammaspectroscopie te bekomen.

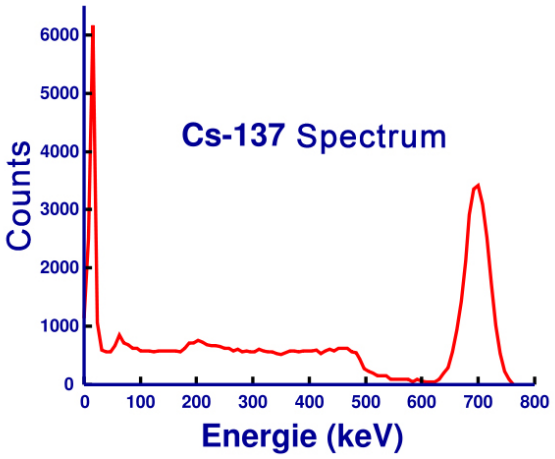
Uiteindelijk volgt het rapporteren van de resultaten van metingen van enkele radioactieve bronnen met behulp van onze gammaspectroscoop in hoofdstuk 9.

Nadat we een volledig werkend systeem bekomen hebben bespreken we een belangrijke toevoeging in de spectroscopie, en de implementatie hiervan op de FPGA: hoofdstuk 10 gaat over *pile-up rejection*.

Hoofdstuk 2

De plaats van een FPGA in Gammaspectroscopie

Vooraleer we aan het echte werk beginnen is een korte inleiding gammaspectroscopie op zijn plaats. Dit wordt behandeld in het eerste deel van dit hoofdstuk, alvorens over te schakelen naar de meer technische beschrijving van de gammaspectroscoop zelf geven we een korte inleiding gammaspectroscopie. In de latere delen van dit hoofdstuk zullen we verder uitweiden over de plaats van een FPGA in dit geheel, gevolgd door een korte inleiding over de FPGA technologie.



Figuur 2.1: Het gammaspectrum van de radioactieve isotoop Cs137. Met de karakteristieke piek op 662 keV

2.1 Inleiding gammaspectroscopie

Gammaspectroscopie is een experiment waarbij de energie van hoog energetische fotonen (gammadeeltjes) wordt gemeten. Dit wordt in twee velden veelvuldig toegepast: In de elementaire deeltjesfysica en in de astrofysica worden zeer hoog energetische (tot de orde van TeV) gammastralen gemeten. In de nucleaire fysica worden gammaspectra van radioactieve bronnen (vb. Fig. 2.1) opgemeten om deze te identificeren en de intensiteit ervan te meten,

de energieën hiervan schalen van keV tot MeV. Dit wordt wijdverspreid toegepast in de nucleaire industrie. Het ontwikkelen van detectoren is dan ook een lucratieve bezigheid.

Het typische resultaat van een gammaspectroscopie experiment is een energiespectrum zoals weergegeven in Fig. 2.1. Hierin worden de gedetecteerde gamma's ingedeeld volgens de energie die ze bezitten. Omdat bij het verval van radioactieve isotopen vaak gammastraling vrijkomt met zeer karakteristieke energieën kunnen we aan de hand van een gammaspectrum een radioactieve bron identificeren. In het gammaspectrum van de bron worden deze energieën weergegeven als scherpe pieken. De rest van het spectrum bestaat voornamelijk uit comptonstraling met een eventuele piek veroorzaakt door *backscatter*, en eventuele achtergrondstraling. Voor een vakkundige bespreking van gammaspectra verwijzen we naar [1].

Voor een experimentator zijn de belangrijkste karakteristieken van een gammaspectroscoop de energieresolutie, het dynamisch bereik en de snelheid. Een goede resolutie laat toe om preciezere metingen uit te voeren en onderscheid te maken tussen bronnen waarvan de pieken dicht bij elkaar liggen. De snelheid is -naast omwille van het gebruiksgemak- ook belangrijk voor de veiligheid: Radioactieve bronnen opmeten is niet zonder gevaar, dus hoe sneller een meting kan gebeuren hoe geringer de blootstelling van de experimentator.

In dit hoofdstuk laten we de fundamenteel-natuurkundige details die belangrijk zijn voor een spectroscopie-experiment zoveel mogelijk achterwege vermits deze niet noodzakelijk zijn bij de ontwikkeling van een detector, hiervoor verwijzen we graag door naar [2]. We leggen de nadruk op zaken die tijdens de ontwikkeling van groot belang zijn, zoals de technische kant van de detector, inclusief karakteristieken zoals de resolutie.

2.1.1 De werking van een gammaspectrometer

Een gammaspectrometer bestaat meestal uit twee delen: een detector, en de elektronica die de signalen hiervan bewerkt en analyseert. De detectoren zelf zijn technisch al zeer ver verfijnd, terwijl in de elektronica meer ruimte voor innovatie is.

Een gamma deeltje kan op verschillende manieren met materie interageren. De voornaamste processen zijn ionisatie en excitatie, de creatie van elektron-gaten paren in een halfgeleider, het foto-elektrisch effect en de creatie van elektron-positron paren. Voor laagenergetische gammaspectroscopie gebruikt men vooral scintillerende materialen of zuivere halfgeleiders. De detector die we in dit werk gebruikt hebben bestaat uit NaI, een kristal dat de gammastraling opvangt en via scintillatie omzet in licht. Deze is gekoppeld met een *Photo Multiplier Tube* (PMT) die een fotogevoelige laag bevat die fotonen omzet in elektronen via het fotoelectrisch effect en deze vervolgens versnelt met behulp van een elektrostatische potentiaal om op hun beurt een elektronenlawine in de detector te veroorzaken. Deze is waar te nemen als een zeer korte spanningspiek ($\sim$ ns) op de output van de detector. De hoeveelheid lading die deze puls bevat is evenredig met de energie van het ingevallen gammadeeltje.

De uitdaging is om uit deze korte piek een waarde voor de energie te distilleren. De lading is evenredig met de oppervlakte onder de puls, maar het rechtstreeks bepalen van deze oppervlakte in zo'n smalle en scherpe piek is technisch onmogelijk. We willen dus een andere –zo eenvoudig mogelijke– meting uitvoeren die de energie van de ingevallen gammadeeltjes oplevert. Dit wordt meestal gedaan door het signaal door middel van versterkers en filters te

vervormen naar een vaste vorm, waarbij op een versterkingsfactor na de oppervlakte onder de curve behouden blijft. Als we van deze curves nu een eenvoudig meetbare karakteristiek kunnen vinden die als maat voor de energie kan dienen, hebben we een mogelijk systeem. Goede voorbeelden hiervan zijn driehoekige of gaussische pulsen: de oppervlakte onder deze pulsen in evenredig met de amplitude ervan, dus de amplitude heeft een rechtstreeks en lineair verband met de hoeveelheid lading die de detector uitstuurt.

Een spectroscopische opstelling bestaat dus naast de detector uit een *preamplifier* die de initiële signalen versterkt, een *shaping amplifier* die de pulsen vervormt, en een *Multi Channel Analyser* (MCA) die de amplitude bepaalt. Omdat we over zeer korte pulsen en hoge pulsfrequenties spreken bestaan oude MCA's uit een grote hoeveelheid analoge elektronica. In ons digitale tijdperk beschikken we over steeds snellere digitale elektronica, die meer complexe analysemethodes toelaat.

2.2 De gebruikte opstelling

Het zou mooi zijn als we de analoge elektronica in de MCA allemaal konden vervangen door een digitaal systeem. In een digitale analyse van de pulsen zien we namelijk één groot voordeel: Het zou moeten toelaten om door het gebruik van fits de te meten pulsen zeer exact te reconstrueren, wat zou kunnen resulteren in een hoge resolutie. Een digitaal platform maakt het bovendien ook mogelijk om te werken met flexibele ruisonderdrukking en filters.

Het nadeel van een klassiek digitaal systeem –een gewone seriële processor dus– is de beperking in snelheid. Een processor kan maar een beperkte hoeveelheid data per tijd verwerken, en voldoet bij hoge *data-rates* niet als verwerkingseenheid. Zelfs het uitvoeren van een eenvoudige fit op een seriële processor vraagt al snel teveel tijd voor processen met hoge *data rates*. Eén oplossing is het synthetiseren van een algoritme in de vorm van een logische digitale schakeling om zo de ballast van een *operating* systeem en uitvoerbaar stukje software te vermijden. We gaan dus van de software-oplossing op de pc, naar een op maat gemaakt stuk hardware. Een hardware oplossing kan veel sneller zijn en laat ook het parallel uitvoeren van verschillende taken toe, en parallellisme is een begrip dat in data-acquisitie goed tot zijn recht komt. Het ontwikkelen van chips is echter erg duur.

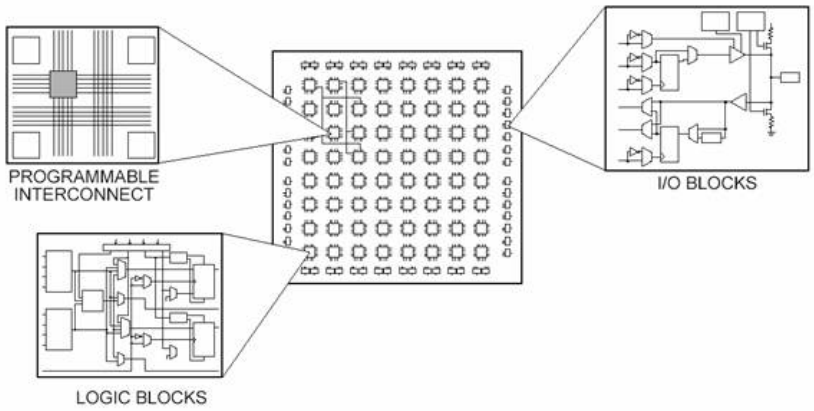
Een *Field Programmable Gate Array* (FPGA) biedt een elegante oplossing. Een FPGA is een digitale elektronische chip waarvan de functionaliteit programmeerbaar is. Een aantal rekenkundige bewerkingen kunnen worden gerealiseerd met een (digitale) logische functie. Door deze functie meerdere malen te implementeren is echte parallelle verwerking van data mogelijk. Het aantal digitale bouwstenen in een FPGA is beperkt en dus is het aantal algoritmes dat kan worden geïmplementeerd beperkt. Tegenover een gewone chip is dit een zeer flexibel systeem omdat de programmeerbaarheid een ontwikkelingsproces toelaat waarbij de programmeur relatief snel een nieuw algoritme kan uittesten. De enige stap die nodig is om een stuk firmware op een FPGA te laten draaien is een compilatie, dit staat in schril contrast met het prototype van een gewone chip dat volledig gefabriceerd moet worden alvorens er testen kunnen gebeuren.

2.3 FPGA's

In deze sectie leiden we het concept FPGA in, eerst geven we een korte toelichting over de hardware, gevolgd door een bespreking van de verschillende tools die beschikbaar zijn om een FPGA te programmeren. Voor een meer gedetailleerde beschrijving van de hardware verwijzen we graag naar [3]. Verder is veel informatie te vinden op de websites van de twee grootste producenten: Xilinx[1] en Altera[2]. Voor de verdere bespreking in dit hoofdstuk is ook de website van National Instruments[3] een goede leidraad geweest.

2.3.1 De hardware

Een FPGA bestaat is een chip waarop logische blokken en I/O blokken zijn gerangschikt zoals weergeven wordt in Fig. 2.2. Tussen de blokken ligt een netwerk van programmeerbare connecties dat ervoor kan zorgen dat de verschillende blokken op de FPGA aan elkaar gelinkt kunnen worden. De logische blokken of *slices* kunnen logische functies zoals AND, OR, NAND, ... uitvoeren. Door de logische blokken te programmeren om een bepaalde functie uit te voeren, en deze via de programmeerbare connecties op de juiste manier met elkaar te verbinden kunnen we tal van taken uitvoeren. Op deze manier worden allerlei binaire bewerkingen mogelijk. Naast deze basiscomponenten beschikt een FPGA vaak ook over een aantal geïntegreerde RAM blokken en een aantal vaste logische blokken die (wiskundige) bewerkingen kunnen uitvoeren.



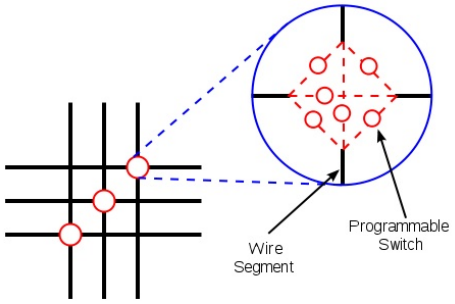
Figuur 2.2: Een schematische voorstelling van een FPGA. Bron: National Instruments

Het basiselement van de FPGA is dus het logische blok of *slice*. De logische functies die deze *slices* kunnen uitvoeren worden geïmplementeerd in *Look-Up Tables* (LUTs), waarin voor elke logische functie de output voor elke mogelijke input is opgeslagen. Deze LUTs worden gekoppeld met een aantal *First In First Outs* (FIFOs) die ervoor zorgen dat de uitkomsten van de bewerkingen opgeslagen en gecommuniceerd kunnen worden. De FIFOs zijn altijd gekoppeld met een klok, zodat de data op de FPGA mooi synchroon gecommuniceerd worden. Elke slice kan meerdere LUTs en FIFOs bevatten om complexere bewerkingen mogelijk te

[1] `http://www.xilinx.com/`
[2] `http://www.altera.com/`
[3] `http://www.ni.com/white-paper/6983/en`

maken. Deze technologie is gebaseerd op*Static Random Acces Memory* (SRAM), dit is zeer snel geheugen dat zijn data verliest bij het uitschakelen van de voeding.

Het netwerk van programmeerbare connecties heeft voor verschillende FPGAs een verschillende lay-out. Het is echter altijd gebaseerd op het principe waarbij de kruising van twee verbindingen op elke mogelijke manier geprogrammeerd kan worden, zoals weergegeven in Fig. 2.3.



Figuur 2.3: Een detailopname van een programmeerbare kruising van twee connecties op een FPGA.

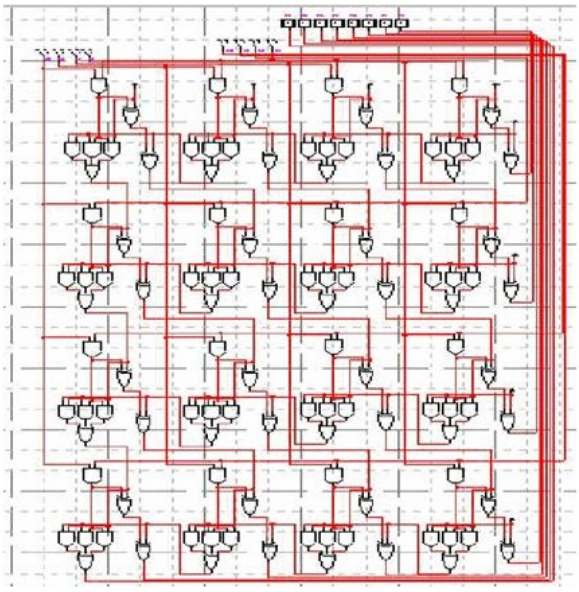
Voor het geheugen dat nodig is om gegevens op te slaan tijdens de werking van de FPGA worden meestal geïntegreerde RAM blokken gebruikt. De logische blokken op de FPGA zelf kunnen ook als geheugen dienen, maar hebben maar een beperkte ruimte. Om de logische blokken vrij te houden voor de functies die uitgevoerd worden worden er dus RAM blokken aan de FPGA toegevoegd. Deze kunnen zowel dienen als grote FIFO's -die nodig zijn voor de communicatie tussen verschillende parallel werkende secties- als vaste geheugencomponenten. Het is echter ook een soort SRAM, en het verliest dus zijn data bij uitschakeling van de voeding[4].

De laatste belangrijke component zijn de vaste logische blokken. Deze blokken, ook wel *Digital Signal Processing* (DSP) *slices* genoemd, zijn aanwezig om wiskundige bewerkingen uit te voeren. Het zijn in de regel geavanceerde *multipliers* die grote binaire getallen met elkaar kunnen vermenigvuldigen. En omdat ook complexere wiskundige bewerkingen (delingen, logaritmes, ...) in digitale logica worden geïmplementeerd aan de hand van meerdere *multipliers*, zijn het veelgebruikte componenten. In wezen kan dit eigenlijk allemaal al met de logica op de FPGA worden uitgevoerd, maar deze bewerkingen zouden een zeer groot aantal logische blokken op de FPGA gebruiken. Ter illustratie is in Fig. 2.4 de digitale logica weergegeven nodig voor de vermenigvuldiging van twee getallen van vier bits.

2.3.2 Programmeertalen

Om een FPGA te programmeren moet aan elke individuele component op de FPGA precies gezegd worden hoe deze geconfigureerd moet zijn. Dit gaat van elk onderdeel van de logische blokken tot alle programmeerbare connecties; een systeem van miljoenen componenten. Door de jaren heen zijn de *tools* die hiervoor gebruikt worden sterk geëvolueerd.

[4]Daarom wordt op een kaart naast een FPGA vaak ook een permanent geheugen toegevoegd. Dit wordt dan bijvoorbeeld gebruikt om de programmatie van de FPGA in op te slaan, zodat deze bij het opstarten van de FPGA terug naar dezelfde toestand kan terugkeren.



Figuur 2.4: Een afbeelding van een 4x4 *multiplier* opgesplitst in de benodigde digitale logica. Bron: National Instruments

In het begin van het FPGA tijdperk (1985) werd met CAD programma's geprogrammeerd. Hierbij moest werkelijk elke component apart door de programmeur beschreven en ingesteld worden. FPGAs zijn ondertussen echter zo groot en complex geworden dat deze benadering onwerkbaar is geworden. Tegenwoordig gebruikt men twee zogenaamde *Hardware Discription Languages* (HDL): VHSIC (Very High Speed Integrated Circuit) HDL (VHDL) en Verilog. Dit zijn hybride talen die een aan C verwante taal combineren met een omgeving die verband houdt met de architectuur van een FPGA. Ze laten weliswaar toe om op een hoger niveau te programmeren, maar een sterke kennis van de hardware blijft een vereiste.

In deze HDLs moet elke vereiste bewerking als een apart blokje met input en output gespecificeerd worden. Deze bewerkingen worden ook wel modules genoemd, en door ze aan elkaar te koppelen en voor elke module de timing en *dataflow* netjes te specificeren kan een algoritme geschreven worden. Vervolgens wordt deze code door een compiler gesynthetiseerd naar een *bitstream* die de configuratie van alle elektronica op de FPGA bevat. Deze compilers zijn ingewikkelde tools die meerdere stadia doorlopen om de *bitstream* te genereren. In wezen zetten ze de code gewoon om in digitale logica; maar om dit op een efficiënte manier te doen, en al deze logica op een efficiënte manier te verbinden is een krachtige en gesofisticeerde tool nodig.

De huidige HDL programmeeromgevingen laten zowel het gebruik van deze HDLs, als de integratie van een meer low-level design toe. Dit is opgezet om een ontwikkelaar de kans te geven om zo flexibel en efficiënt mogelijk te programmeren.

Tegenwoordig zijn er echter ook zogenaamde *High Level Synthesis* (HLS) *tools* beschikbaar. Deze tools zijn zeer toegankelijk, omdat een gebruiker erg weinig van een FPGA moet afweten om een algoritme op een FPGA te implementeren. De gebruiker hoeft namelijk niet per se veel aandacht te besteden aan de synchronisatie en data flow van een algoritme, op voorwaarde

dat een compiler dit met minimale input van de gebuiker zelf kan opstellen. SIMULINK –verbonden aan het MATLAB pakket van Wolfram systems– en LABVIEW –van Natianal Instruments (NI)– zijn de twee programma's die een HLS omgeving aanbieden waarmee je FPGAs kunt programmeren. Beide omgevingen zijn zeer gelijkaardig aan de klassieke versies van de programma's, maar bieden enkel een beperkt aantal functies aan die op de FPGA kan runnen. De tools zijn er op gericht om een specifiek algoritme te implementeren op de FGPA. Zaken zoals zelf lerende algoritmes of recursieve algoritmes zijn niet te implementeren.

In dit proefschrift is voor LABVIEW gekozen. Het bedrijf dat deze tool ontwikkelt -National Instruments (NI)- claimt dat deze visuele programmeertaal uitermate geschikt is voor het programmeren van FPGA's: "*The LabVIEW programming environment is distinctly suited for FPGA programming because it clearly represents parallelism and data flow, so users who are both experienced and inexperienced in traditional FPGA design processes can leverage FPGA technology.*"

Voor een fysicus is de keuze voor een *high level synthesis tool* snel gemaakt. De bovenstaande claim van NI staat immers in schril contrast met de HDLs die een lange leercurve hebben, en een goede basiskennis en begrip van de hardware op de FPGA vereisen.

De beloftes van de producenten zijn dat je op een makkelijke manier met een FPGA kunt werken. Eén van de doelen van deze thesis is om uit te testen hoe moeilijk het is voor iemand zonder achtergrond in FPGA's, en met een beperkte achtergrond in de digitale elektronica, een systeem te implementeren op een FPGA. Een belangrijke vraag die we in deze thesis onderzoeken is of het effectief waar is dat je met behulp van zo'n high level tool met een zeer beperkte kennis van FPGAs al in staat moet zijn om deze hardware te gebruiken.

2.3.3 Fixed point arithmetic

Een van de nadelen van de FPGA –maar tegelijk een van de redenen waarom een FPGA sneller kan werken dan een gewone seriële processor– is het feit dat deze enkel kan werken met dataformaten met vaste afmetingen. Voor gehele en natuurlijke getallen worden integers gebruikt, en voor de reeële getallen worden zogenaamde *fixed points* (FXP) gebruikt. Net als een integer is een FXP gewoon een reeks bits op een FPGA, het enige verschil is dat er voor een FXP aangeduid is welke bits voor het gehele deel staan, en welke voor het decimale deel staan. Op een FPGA moet het aantal bits waaruit deze getallen bestaan van te voren gedefinieerd worden. Voor een integer wordt dit de Integer *Word Length* (IWL) genoemd; voor een FXP wordt een set van twee getallen gebruikt (WL,IWL), het totale aantal bits wordt aangeduid met de *Word Length* (WL), en het aantal bits daarvan dat de gehele getallen aanduidt wordt de *Integer Word Length* (IWL) genoemd. Verder krijgen integers en FXP de aanduiding die vertelt of ze enkel positief (*unsigned*), of positief en negatief (*signed*) kunnen zijn. Bij *signed* integers en *signed* FXPs wordt een bit van de IWL gebruikt om het teken aan te geven.

Integers bestaan in LABVIEW in drie formaten: 8 bits, 16 bits, en 32 bits, waarvan elk een *signed* en een *unsigned* versie bestaat. Voor FXPs is er meer flexibiliteit, voor elke FXP kun je aangeven of deze al dan niet *signed* is, en de WL en de IWL definiëren. Deze zaken komen in dezelfde volgorde terug in de notatie die wij hier gebruiken om een FXP aan te duiden: zo duidt $(\pm, 18, 6)$ een *signed* integer aan met een WL van 18 bits en een IWL van 6 bits. Aan de hand van deze kennis kan nu voor elke integer en FXP berekend worden wat

de maximale positieve en negatieve waarden zijn die deze kan bevatten, alsook de maximale precisie. Voor een $(\pm, 18, 6)$ FXP is dit bijvoorbeeld: $\pm$ betekent signed, een IWL van 6 bits betekend voor een signed integer uiterste waarden van $\pm 2^{6-1} = \pm 32$, en een maximale precisie van $2^{-(18-6)} = 0.000244$.

Berekeningen met getallen die uit meer bits bestaan zijn voor een FPGA meer *resource* intensief om uit te voeren. Voor bewerkingen die door de DSP48E componenten uitgevoerd worden zullen voor grotere FXP ook meerdere DSP48E blokken per berekening nodig zijn. Het omspringen met het beperkte aantal DSP48E componenten zal ons dan ook dwingen om de afwegingen tussen precisie en *resource* gebruik meerdere malen te maken in deze thesis. Bovendien is het niet makkelijk om deze afweging te maken, omdat het –voor iemand zonder een sterke achtergrond in deze elektronica– onmogelijk is om a priori aan de hand van informatie die LABVIEW en Xilinx over de DSP48E componenten ter beschikking stellen te bepalen hoeveel van deze componenten een bewerking nodig zal hebben.

Hoofdstuk 3

Ontwikkelingsplatform

In dit hoofdstuk wordt het in dit werk bestudeerde ontwikkelingsplatform besproken. Deze bestaat uit een analoog deel en een digitaal deel (de FPGA). Het analoge deel bestaat uit 2 belangrijke delen: de detector en de *shaping amplifier*. Het is belangrijk om het geheel van het ontwikkelingsplatform goed te begrijpen omdat dit het vertrekpunt is van de keuzes die doorheen dit project gemaakt zijn. De in dit hoofdstuk besproken hardware legt namelijk het kader vast waarin we kunnen werken. Een kader dat zeer belangrijk zal blijken omdat de beperkingen van onze hardware (analoog en digitaal) de grenzen van de performantie van onze digitale verwerking.

3.1 De FPGA

Voor onze toepassing is gekozen voor de reeds aanwezige National Instruments PXI7851R kaart (zie Fig. 3.1). Deze kaart bevat een Virtex-5 LX30 FPGA, die beschikt over 4800 *slices* bestaande uit vier FIFO's en vier 6-input LUTs per slice. Verder beschikt deze FPGA over 32 DSP48E *slices* (25x18bit *multipliers*), 18 64Kb RAM blokken, en 96 digitale IO kanalen. De standaard kloksnelheid van de FPGA is 40 MHz, wat naar gelang van de geïmplementeerde code nog opgevoerd kan worden. Naast de FPGA beschikt deze kaart nog over tal van andere I/O mogelijkheden. Voor ons zijn enkel de analoge I/O kanalen relevant: deze bestaan uit acht 750 kHz *Analog to Digital Converters* ADCs die signalen kan omzetten naar een digitaal signaal met een resolutie van 16 bits en een bereik van -10 V en 10 V.

Figuur 3.1: Een afbeelding van de NI PXI7851R kaart. De Virtex-5 LX30 FPGA is het grijze vierkantige blok.

3.2 Onze analoge opstelling

3.2.1 De detector

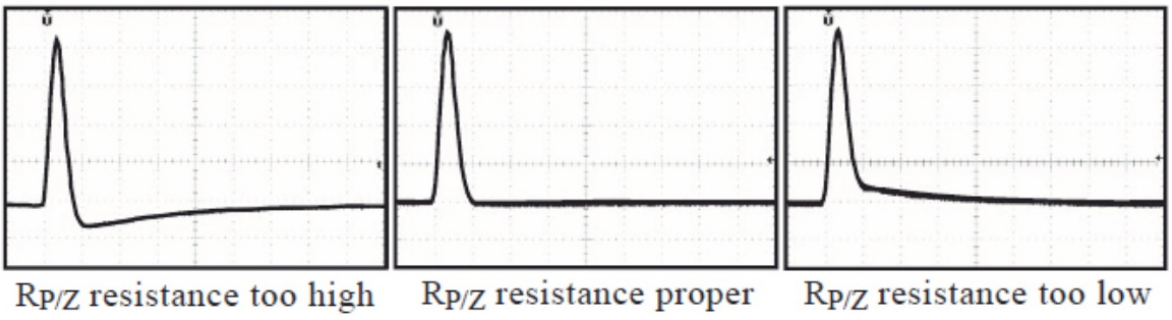
De de detector is een cilindervormig NaI kristal gekoppeld aan een fotomultiplier met een ingebouwde *preamplifier*. Dit geheel wordt aangedreven door een hoogspanningsbron in de uitvoering van een standaard NIM-module [4]. De gebruikte hoogspanning bij al onze experimenten is 650 V, die is vastgelegd door de specificaties van de detector.

3.2.2 De *shaping amplifier*

De *shaping amplifier* is een CSA 4 [5] unit. Het hoofdonderdeel is een *shaping amplifier* die signaal pieken van de de detector output omzet in gaussische pulsen. De pulsen liggen dicht bij een gauss, maar zijn niet perfect: Zo is de breedte na het maximum bijvoorbeeld iets groter dan ervoor. Er zijn vier verschillende keuzes van *shaping times*: 8 µs, 2 µs, 500 ns en 100 ns, die de breedte van de uitgangspuls bepalen. De *shaping amplifier* kan ook een regelbare versterking van 1x tot 2500x leveren. Omdat de input vanuit de detector niet altijd een sterkl genoeg signaal oplevert om de *shaping amplifier* direct aan te sturen bevat de unit ook een *charge sensitive preamplifier* die het signaal versterkt en de piek versmalt. In onze opstelling was het signaal van de detector te zwak, en is voor het gebruik van de *charge sensitive preamplifier* geopteerd.

Het bereik van de output van de versterker ligt tussen 0 V en +10 V, de gaussische pulsen schalen echter maar lineair met de input tot 8 V. De versterkingsfactor dient dus zo afgeregeld te worden dat het merendeel van de output onder de 8 V blijft.

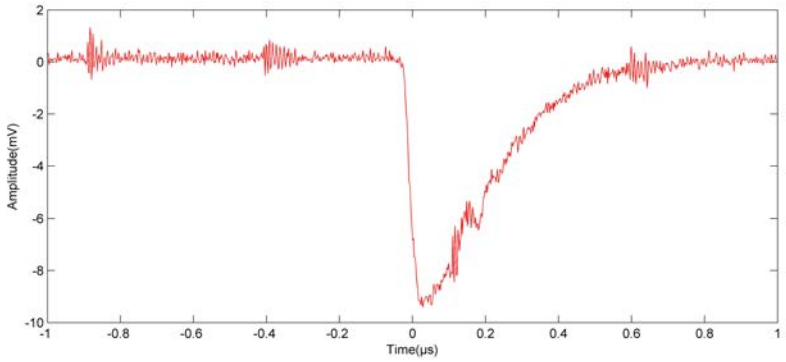
Verder bevat de *shaping amplifier* ook een pole-zero correction, en een *baseline restorer*. Deze twee correcties gereedschap zijn daar omdat de eindige *fall time* van de input puls in de *shaping amplifier* een kleine fout veroorzaakt, waarvoor gecorrigeerd moet worden. De effecten hiervan zijn weergegeven in Fig. 3.2 . Het is de bedoeling om de puls zo dicht mogelijk bij een pure gauss te krijgen, maar dit lukt niet altijd omdat de fout groot kan worden als de *fall time* van de input puls lang is.



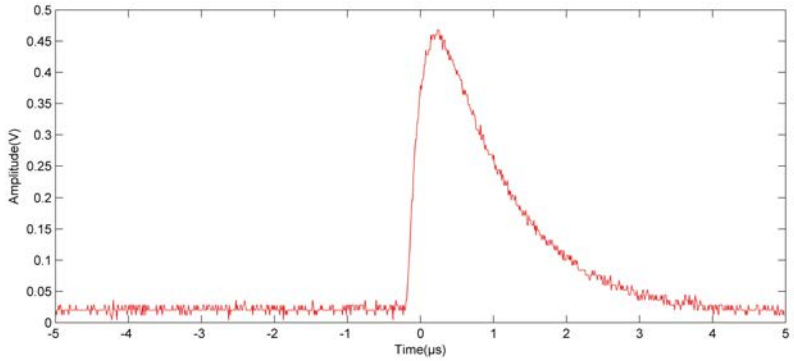
Figuur 3.2: Een afbeelding van de effecten van pole-zero correction. De middelste afbeelding geeft de gauss weer die we als resultaat wensen.[5]

3.2.3 De signalen in onze opstelling

Een voorbeeldpuls die de detector levert vind je in Fig. 3.3. Deze puls sturen we vervolgens door de *charge sensitive preamplifier*, die ruisonderdrukking uitvoert, en het signaal inverteert en met een factor 10 versterkt. Zo wordt de puls omgezet in een piek zoals te zien in Fig. 3.4. Deze piek wordt vervolgens door de *shaping amplifier* omgezet in een signaal dat sterk lijkt op een gausscurve. Deze puls moet echter gecorrigeerd worden door middel van de *pole-zero correction* en de *baseline restoration*, en dat is waar de problemen ontstaan. Voor een puls met een *shaping time* van 2 µs en minder krijgen we een puls die visueel een gauss benadert (zie bv. Fig. 3.5). Voor pulsen met een *shaping time* van 8 µs (zie Fig. 3.6) is het echter niet meer mogelijk om de puls mooi te corrigeren. Dit heeft te maken met het feit dat er in dit geval een hogere versterkingsfactor nodig is voordat de shaping plaatsvindt[1], hierdoor wordt de *fall time* langer, en de fout dus groter. De fout wordt in ons geval zo groot dat deze niet meer volledig gecorrigeerd kan worden.

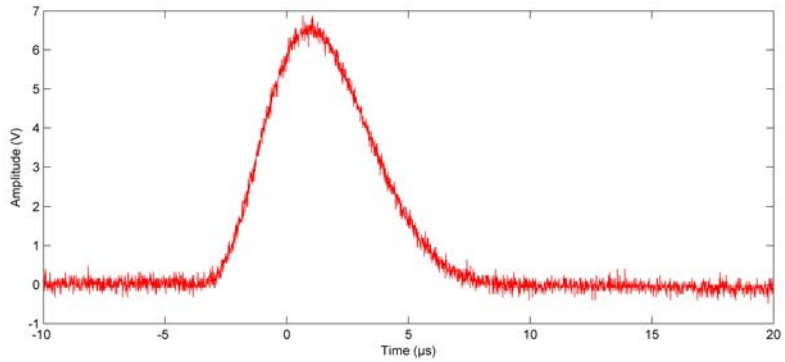


Figuur 3.3: Het signaal dat onze detector oplevert bij het invallen van één enkel gammadeeltje. Dit signaal is al versterkt door een *charge sensitive preamplifier* die gemonteerd is in de basis van de PMT. Een ruw signaal van de PMT zelf kunnen we niet meten, maar is normaal een gelijkaardige puls met een lengte van enkele ns, en een sterkte van de orde van mA.

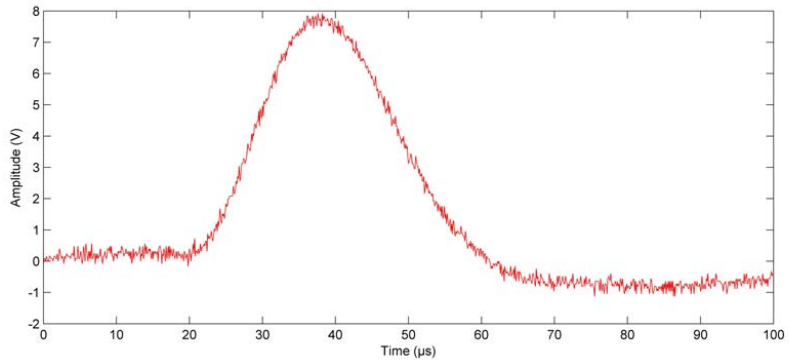


Figuur 3.4: De CSA4 *charge sensitive preamplifier* versterkt het signaal uit de detector, met als resultaat deze puls.

[1]Omdat de oppervlakte onder een bredere puls groter is, is er een grotere versterkingsfactor nodig om dezelfde pulshoogte te verkrijgen als bij een smallere puls.



Figuur 3.5: De vorm van de puls na bewerking met onze *shaping amplifier* met 2 µs *shaping time*.



Figuur 3.6: De vorm van de puls na bewerking met onze *shaping amplifier* met 8 µs *shaping time*.

In de technisch eenvoudigste opstelling die we kunnen opbouwen gebruiken we één enkele ADC[2] om het signaal te samplen. Een 750 kHz ADC zal elke 1.33 µs een sample nemen. Als we deze informatie combineren met de breedte van de gaussische puls kunnen we voorspellen hoeveel punten de ADC per puls zal opmeten. Als we de breedte van de puls afschatten als vier keren de *shaping time* (4σ) beschouwen we bijna de volledige gausscurve. Voor een shapingtime van 2 µs zal een ADC maximaal 6 punten kunnen samplen, en voor 8 µs maximaal 24. Als we in de keuze van het aantal punten dat we meenemen in de fit wat flexibiliteit willen behouden zullen we –alleszins in een initieel systeem– toch de 8 µs *shaping time* moeten gebruiken. Dit is enerzijds een nadeel, omdat de licht vervormde gaussische pulsvorm zeker zijn invloed zal hebben op de resultaten. Anderzijds is het ook een voordeel om een systeem te ontwikkelen aan de hand van deze pulsen, vermits perfect gaussische pulsen niet makkelijk te verkrijgen zijn, en het ontwikkelen van een systeem dat die perfectie niet nodig heeft wel eens zou kunnen resulteren in een robuuster en algemener te gebruiken systeem.

Hoewel dit geen perfecte gausscurves zijn is een gausscurve toch de enige ons bekende functie die de vorm van de pulsen in ons systeem best benadert.

Belangrijk om te onthouden naar volgende hoofdstukken toe is het feit dat we nu over gausscurven beschikken, waarvan de integraal proportioneel is met de in de detector ingevallen

[2] Als we meerdere ADCs parallel willen gebruiken moet een externe *delay line* aangelegd worden.

gammastraal. Voor vaste breedtes van deze curve (en dus *shaping times*) zal dan ook de amplitude proportioneel zijn met de energie. Het doel van deze thesis zal dan ook zijn om deze amplitude snel en nauwkeurig te bepalen.

Hoofdstuk 4

Het fitten van een gaussische puls op een FPGA

Het uitvoeren van *Pulse Height Analysis* (PHA) met een FPGA is geen nieuw concept, en gaat van zeer eenvoudige systemen [6] tot meer geavanceerde oplossingen [7] [8]. Het wordt in diverse toepassingen gebruikt. Het uitvoeren van fits aan gaussische curven wordt weinig gedaan met FPGA's, zeker in het gebied van gammaspectroscopie.
In dit hoofdstuk beginnen we met de beschrijving van de fit die we gebruikt hebben. Dit gaat van een eenvoudige theoretische basis over in een vereenvoudiging die de implementatie op de FPGA ten goede moet komen. De volgende stap bestaat uit het wiskundig uitwerken van de fit op de manier waarop deze op de FPGA geïmplementeerd wordt. Tenslotte geven we een overzicht van de meest belangrijke stappen in het ontwikkelingsproces, om te kaderen op welke manier we deze fit best op een FPGA implementeren.

4.1 Van een gauss naar een kwadratische functie

Zoals in hoofdstuk 3 benaderen we onze pulsen door middel van een gaussfunctie:

$$Y = Ae^{-\frac{(X-\mu)^2}{2\sigma^2}} \tag{4.1}$$

Hierin is A de amplitude van de puls, μ de positie van het middelpunt, en σ een maat voor de breedte van de puls (ofwel de standaardafwijking). We beschouwen al deze parameters voorlopig als vrij.

We moeten dus op zoek naar een manier om deze gaussfunctie te fitten, en omdat we weten dat een FPGA maar een beperkte hoeveelheid logica bevat gaan we op zoek naar een methode met zo min mogelijk complexiteit. Een eenvoudige manier om een fit met deze functie te doen is door middel van een kwadratische fit. In deze redenering zijn we niet de eerste en de enige, andere voorbeelden zijn te vinden in [9] en [10]. Als we het logaritme van de gaussfunctie nemen krijgen we

$$\begin{aligned}\ln(Y) &= ln(A) - \frac{(X-\mu)^2}{2\sigma^2} \\ &= -\frac{1}{2\sigma^2}X^2 + \frac{\mu}{\sigma^2}X - \frac{\mu^2}{2\sigma^2} + ln(A).\end{aligned}$$

Wat zich met

$$\begin{aligned} x &= X \\ y &= \ln(Y) \\ a &= -\frac{1}{2\sigma^2} \\ b &= \frac{\mu}{\sigma^2} \\ c &= -\frac{\mu^2}{2\sigma^2} + \ln A \end{aligned}$$

herleidt tot de eenvoudige kwadratische functie:

$$y = ax^2 + bx + c.$$

4.2 De kleinste kwadraten methode voor een kwadratische fit

Een eenvoudige methode voor het uitvoeren van een kwadratische fit aan n punten (x_i, y_i) gebeurt door middel van de kleinste kwadraten methode. Deze bestaat eruit een fit $f(x)$ te bepalen waarvoor de kwadratische fout

$$\sum_{i=1}^{n} \left[y_i - f(x_i)\right]^2 = \sum_{i=1}^{n} \left[y_i - ax_i^2 - bx_i - c\right]^2 \tag{4.2}$$

minimaal is. De parameters van de fit vinden we door dit minimum te bepalen, dit kan eenvoudigweg gedaan worden door (4.2) partieel af te leiden naar de verschillende parameters, en deze vergelijkingen gelijk aan nul te stellen. Dan krijgen we, met behulp van

$$M_{kl} = \sum_{i=1}^{n} x_i^k y_i^l, \tag{4.3}$$

een stelsel van drie vergelijkingen:

$$\begin{cases} M_{01} &= cM_{00} + bM_{10} + aM_{20} \\ M_{11} &= cM_{10} + bM_{20} + aM_{30} \\ M_{21} &= cM_{20} + bM_{30} + aM_{40}. \end{cases}$$

4.2.1 Vereenvoudigingen

Door een eenvoudige ingreep –ook toegepast in track fitting met een FPGA[11]– kunnen we dit stelsel nog vereenvoudigen. Dit is zeer belangrijk zoals zal blijken in de volgende secties: Als we de x-coördinaten van onze punten equidistant, en symmetrisch rond 0 kiezen wordt dit stelsel een stuk eenvoudiger. Alle M_{k0} met oneven machten in x zullen wegvallen. Het stelsel herleidt zich tot

$$\begin{cases} M_{01} &= cN + aM_{20} \\ M_{11} &= bM_{20} \\ M_{21} &= cM_{20} + aM_{40}, \end{cases} \tag{4.4}$$

waarbij we het aantal punten $N = M_{00}$ ook hebben ingevoerd.

Met het oog op de FPGA is dit niet zo'n kunstgreep, want omdat de ADC een vaste *sample rate* heeft zijn de punten sowieso al equidistant. En de waarde die we aan onze x-coördinaten geven staat ons vrij, zolang we ons, voor een enkele reeks punten waaruit een puls bestaat, maar houden aan het feit dat de punten equidistant moeten zijn, en in de juiste volgorde gekozen moeten worden. Deze keuzevrijheid kunnen we dus gebruiken om de punten symmetrisch rond het nulpunt te kiezen. Praktisch realiseren we dit op de FPGA door een stapgrootte op de x-as te kiezen van 2. Voor oneven puntenaantallen kiezen we dan als coördinaten 0, en de dichtstbijzijnde even getallen rond het nulpunt (voor 7 punten is dit bv. -6, -4, -2, 0, 2, 4, 6). Voor even getallen kiezen we gewoon de dichtstbijzijnde oneven getallen rond het nulpunt (voor 10 punten is dit bv. -9, -7, -5, -3, -1, 1, 3, 5, 7, 9). In wezen is één klokpuls van de ADC dus equivalent aan een sprong van 2 in deze integer waarden.

4.2.2 Oplossing

Zowel het eerste, als het vereenvoudigde stelsel kunnen worden opgelost. Wij zullen kiezen voor de vereenvoudigde versie, omdat eenvoud ook op FPGA niveau de benodigde resources beperkt. Een oplossing van stelsel (4.4) is:

$$\begin{cases} a &= \frac{M_{01}-cN}{M_{20}} \\ b &= \frac{M_{11}}{M_{20}} \\ c &= \frac{M_{20}M_{21}-M_{10}M_{40}}{M_{20}^2-NM_{40}}. \end{cases} \tag{4.5}$$

Deze oplossingen kunnen we gebruiken om de parameters van de gaussfunctie (4.1) te bepalen via:

$$\begin{cases} \sigma &= \sqrt{\frac{1}{-2a}} \\ \mu &= -\frac{b}{2a} \\ A &= \exp\left(c - \frac{b^2}{4a}\right), \end{cases}$$

waarmee we het gereedschap hebben om een eenvoudige en analytisch juiste fit aan een gaussische puls te doen.

4.2.3 Een efficiente implementatie met de minste delingen.

Zoals in sectie 2.3.1 al besproken is het uitvoeren van delingen op een FPGA een zeer inefficiënt proces, dat dus relatief lang kan duren, en vooral veel resources vereist. Om dit zoveel mogelijk te vermijden rekenen we onze fit uit op een manier die zo min mogelijk delingen bevat. De volgende berekening van het stelsel bevat maar twee delingen: Eerst rekenen we de coëfficient c uit exact zoals deze is voorgesteld in (4.5), dit bestaat uit drie vermenigvuldigingen en twee verschillen, gevolgd door de eerste deling. Vervolgens berekenen we

$$\begin{aligned} M_{20}a &= M_{01} - cN, \\ \frac{b^2}{4a} &= \frac{M_{11}^2}{4M_{20} \cdot M_{20}a}, \\ \text{en finaal } \ln A &= c - \frac{b^2}{4a}. \end{aligned}$$

Wat wederom maar één deling bevat.

In totaal kunnen we nu een implementatie maken waarin de zwaarste bewerkingen van het algoritme een logaritme, gevolgd door twee delingen en een exponentiële zullen zijn.

4.3 Implementatie op de FPGA

4.3.1 Het ontwikkelingsproces op de FPGA

Nu we over het belangrijkste deel van ons gereedschap –de fit– beschikken kunnen we naar een implementatie op de FPGA toewerken. Dit is een tijdrovend ontwikkelingsproces geweest omdat het programmeren van een FPGA –zelfs met de high level LABVIEW omgeving– niet eenvoudig bleek. De grenzen die de hardware ons opleggen spelen hier een grote rol in. Omdat de LABVIEW omgeving als *high level synthesis tool* toch ver van implementatie op hardware zelf staat is het erg moeilijk om in te schatten en te begrijpen welke delen van algoritmes deze grenzen overschrijden. De compiler zal waarschuwingen geven indien er zich problemen voordoen, maar de bijhorende indicaties naar de aard van deze problemen zijn vaak sub-optimaal. Daardoor is een vorm van *debugging* vereist, een gegeven dat niet zo eenvoudig omdat het onmogelijk is om zaken die op een FPGA runnen te *proben*. Ervaring met het programmeren van FPGA's, zal dit proces sterk kunnen verkorten.

In de rest van dit hoofdstuk overlopen we de belangrijkste stappen in het ontwikkelingsproces die uiteindelijk geleid hebben tot het resulterende product, waarvan we de robuustheid en performantie in de volgende hoofdstukken zullen bepalen. We pogen hier niet alleen aan te tonen hoe we tot ons finale ontwerp geraakt zijn, maar ook welke moeilijkheden je kunt tegenkomen bij het werken met de LABVIEW FPGA toepassing zonder veel kennis van de lagere programmatieniveaus. Het is in de LABVIEW FPGA toepassing vooral moeilijk om in te schatten hoeveel resources bepaalde zaken van de FPGA zullen vragen. Omdat dit ook nergens in de LABVIEW omgeving of documentatie op een duidelijke manier wordt aangegeven is dit iets dat je enkel uit ervaring kunt leren.

4.3.2 Een systeem met een variabel puntenaantal

De eerste manier om het fit algoritme te implementeren was een eenvoudige keuze. Een eerste poging moest bestaan uit een zo algemeen en flexibel mogelijk systeem. Hiermee bedoelen we dat dit systeem zou moeten werken voor een breed scala aan pulsbreedtes. Praktisch gezien bestaat dit uit twee delen: (i) Een module die enkel de punten die tot één puls behoren uit een signaal distilleren: de trigger. De eenvoudigste vorm hiervan selecteert een reeks punten boven een vooraf ingestelde drempelwaarde waarboven het signaal (voor een afzienbare tijd) moet uitstijgen alvorens beschouwd te worden als een signaal (ook wel het triggerniveau genoemd). (ii) Een hiernavolgende module die de beste fit aan deze punten zoekt volgens de methode die eerder in dit hoofdstuk besproken werd. Deze aanpak *sampled* de puls met zoveel mogelijk punten, waardoor de ruis maximaal zou moeten worden uitgemiddeld. Omdat het aantal punten die boven het triggerniveau liggen zal afhangen van de amplitude en de pulsbreedte van de puls is het aantal punten waarvoor dit algoritme werkt variabel.

Tests van een alleenstaande versie van de fit, geïmplementeerd op een PC, en simulaties op PC van het gehele programma dat op de FPGA loopt toonden beiden aan dat deze methode

werkte. Beiden leverden ons een amplitude op die overeenkwam met wat we op het oog waarnamen op de testpulsen. Ook een vergelijking met de waarden die een algemene fit door middel van *Gaussian Peak Fit Coefficients.vi* –een LABVIEW methode die gaussische functies fit– bevestigden de juistheid van de fit.

De compilatie leerde ons echter dat dit algoritme niet op de FPGA paste, het zou teveel resources[1] gebruiken. De compileren kwantificeert deze zaken jammer genoeg niet, maar het probleem bevond zich duidelijk in het fit algoritme, er zouden namelijk te weinig DSP48E componenten op de FPGA zijn voor deze implementatie. Hierin worden alle zaken die fixed-point vermenigvuldigingen en delingen vereisen uitgevoerd.

4.3.3 Optimalisatie van de formaten van de FXP

Een eerste aanpak voor het verminderen van de complexiteit –en dus ook het aantal benodigde DSP48E's– is eenvoudig te begrijpen: als we de grootte (dus zowel de range als de precisie) van onze FXP's waarmee de vermenigvuldigingen en de delingen uitgevoerd worden kleiner kunnen krijgen, zullen er ook minder DSP48E's nodig zijn. Omdat –als je zelf niets verandert– LABVIEW automatisch de grootte van een FXP die de uitkomst van een bewerking is (+,-,x,...) toekent naargelang het formaat van de FXP waarmee deze berekeningen worden uitgevoerd is dit voor verbetering vatbaar. Wij weten immers meer dan LABVIEW, wij kunnen voorspellen hoe groot, en hoe precies de maximale amplitude van een puls is, en we kunnen van te voren uitrekenen wat de maximale waarden van M_{xy} kunnen zijn voor een vast aantal punten. Vervolgens kunnen we voor elke berekening in 4.2.3 uitrekenen welke formaten we moeten toekennen. Omdat het over een keten van bewerkingen gaat maakt dit verder in de keten een groot verschil. Als laatste kunnen we ook bekijken wat voor precisie (en dus resolutie) we van ons systeem eisen. Als we dit afschatten kan de keten van achter naar voor doorlopen worden om de nauwkeurigheid van de FXP aan te passen. Het heeft immers geen zin om de bewerkingen uit te voeren met een resolutie die veel groter is dan hetgeen we van ons systeem verwachten.

Om de formaten van de FXP uit te rekenen is in LABVIEW de Format Calc.vi geschreven, dit is een tool die de benodigde groottes van de FXPs weergeeft naargelang het aantal punten, en het formaat van de amplitude. Hiermee kan alles behalve de exacte precisie netjes uitgerekend worden, dit moet nog altijd manueel bekeken worden. In 7.1 is een voorbeeld te vinden van zo'n optimalisatie. Laat het in ieder geval duidelijk zijn dat als voor elk FXP waar we mee werken apart de precieze grootte gedefinieerd moet worden de last van het programmeren drastisch verhoogd wordt, dit omdat zelfs een eenvoudige berekening op deze manier veel aandacht van de programmeur vraagt.

Deze optimalisatie hebben we eerst uitgevoerd voor een systeem dat maximaal 24 punten aankan. Een logische keuze, vermits dit het maximale puntenaantal is dat we kunnen verkrijgen in onze opstelling bij een *shaping time* van 8 µs (zie sectie 3.2.3). In deze Wederom paste dit niet op de FPGA omdat er niet genoeg DSP48E componenten waren op de FPGA. En we hebben geen mogelijkheden ter beschikking om erachter te komen hoeveel DSP48Es er precies vereist zijn, om de vergelijking met het vorige algoritme te maken. In een volgende

[1] We hebben pogingen ondernomen om dit te kwantificeren door het algoritme te compileren voor een grotere FPGA. Hoewel dit mogelijk moet zijn liet ons softwarepakket dit niet toe.

poging hebben we dezelfde optimalisatie uitgevoerd, maar nu voor lagere aantallen punten (wat resulteert in kleinere M_{xy}): 12, en 4. Dit gaf ons opnieuw hetzelfde negatieve resultaat. Alle documentatie die NI ter beschikking stelt[2] wijst er echter op dat de grootte van FXP een grote rol speelt in het aantal benodigde DSP48Es per bewerking. We hebben de voetafdruk op de FPGA dus verlaagd, maar nog niet in voldoende mate.

4.3.4 Een systeem met een vast aantal punten

Op zoek naar een volgende optimalisatie is er nog een kandidaat: proberen om het aantal delingen nog te verminderen. Zoals al aangehaald kunnen delingen van binaire getallen –in vergelijking met vermenigvuldigingen– enkel op zeer inefficiënte manieren uitgevoerd worden, en is er per deling dus een groot aantal DSP48E blokken vereist. Dit leidt ons naar een systeem met een vast aantal punten: als we van te voren het aantal punten vastleggen kennen we de coëfficiënten M_{00},M_{20} en $M40$ al van te voren, wat betekent dat we de eerste deling in sectie 4.2.3 kunnen vervangen door een vermenigvuldiging met een van te voren uitgerekend getal (dit wordt ook wel een schaling genoemd). Bijkomend voordeel is dat de vermenigvuldigingen voor de berekeningen van M_{00},M_{20} en $M40$ ook wegvallen. Een nadeel is echter dat de trigger complexer wordt, deze moet nu immers uit alle punten waar een puls uit bestaat een vast aantal relevante punten kiezen.

Een eerste implementatie bij 12 punten die op deze wijze geoptimaliseerd is paste wel op de FPGA, en gebruikt zelfs niet alle beschikbare DSP48E's. Verder bleek deze eerste implementatie bij 18 punten net 32 DSP48E's te gebruiken, en dat implementaties met meer punten niet op de FPGA passen. In dit stadium was echter nog niet zo geoptimaliseerd als in het eindresultaat, dus een hoger aantal punten zou zeker mogelijk moeten zijn.

Dit is het ontwikkelingsproces dat tot de basis van ons systeem heeft geleid. In de volgende hoofdstukken wordt het basisidee van een systeem met een vast aantal punten besproken en geoptimaliseerd, en vervolgens geïmplementeerd.

4.3.5 Een hybride systeem

Als laatste moet de opmerking nog gemaakt worden dat ook de mogelijkheid tot een hybride systeem nu bestaat. Als we voor alle puntenaantallen waarvoor het vaste systeem op de FPGA past de schalingsfactoren uitrekenen en in een geheugen op de FPGA opslaan, kan het algoritme wel voor verschillende aantallen punten uitgevoerd worden. Als er echter één ding duidelijk is, is het dat een eenvoudiger systeem beter werkt op een FPGA. Het is dus geen slecht idee om eerst te kijken hoe een systeem met een vast aantal punten werkt, met de optie om dit uit te breiden naar een complexere hybride versie.

In het volgende hoofdstuk wordt echter een analyse van het optimale aantal punten om te gebruiken in een fit uitgevoerd. Deze toont aan dat een situatie waarin we alle punten uit een puls meenemen niet noodzakelijk de beste fit oplevert, het implementeren van een hybride systeem zal dan ook overbodig blijken.

Tot slot dienen we ook nog op te merken dat we één voor de hand liggende optimalisatiemethode niet gebruikt hebben: Het is degene waarbij we bewerkingen die veel resources vragen

[2] `http://belgium.ni.com/`

serieel door hetzelfde blok op de FPGA laten uitvoeren. Dit is –omdat de gevraagde FXP formaten meestal niet hetzelfde zijn– niet zo eenvoudig, maar kan potentieel wel een grote winst aan resources opleveren. Deze optimalisatie hebben we achter de hand gehouden om eventuele toekomstige complicaties op te lossen. Een bijkomend nadeel van deze methodes is dat de functies die veel resources vragen ook vaak functies zijn die wel wat rekentijd vragen, deze ingreep kan dus invloed hebben op de snelheid van het systeem.

Hoofdstuk 5

PC simulaties van de gaussische fit

Door onze onderzoeken in het vorige hoofdstuk beschikken we nu over een fit en weten dat we de implementatie met een vast aantal punten willen uitvoeren. Vooraleer we tot een implementatie op de FPGA overgaan moeten er nog een aantal zaken nader onderzocht worden:

- Allereerst zouden we de juistheid van fit en vooral van de resulterende amplitude moeten beoordelen. Hiervoor zullen we deze zaken eerst moeten parametriseren.
- Ten tweede bestuderen we de invloed van ruis op ons systeem.
- Voorts willen een optimaal systeem kiezen uit twee mogelijke triggersystemen: Eén dat de te fitten punten zo dicht mogelijk rond het maximum van de pulsen kiest. En een tweede dat de te fitten punten zo over de puls verdeelt dat een zo breed mogelijk sample van de puls beschouwd wordt.
- Verder vergelijken we twee variaties op de kwadratische fit: Eén waarbij we de parameter σ als vrij beschouwen, en een tweede waarbij we deze parameter van te voren vastleggen aan de hand van de *shaping time* van onze *shaping amplifier*.
- Door deze proeven te bekijken voor systemen die met verschillende aantallen triggerpunten werken willen we finaal ook onderzoeken of er een aantal punten waarvoor ons systeem optimaal presteert. Het minimaal aantal punten leggen we op 4, één meer dan nodig is om ons stelsel van 3 vergelijkingen (4.5) op te lossen. Het maximale aantal ligt op 24 omdat dit het maximale aantal punten is dat we met een enkele 750 kHz ADC kunnen samplen in een puls met een *shaping time* van 8 µs (zie sectie 3.2.3).

Hoewel we a priori in sectie 3.2.3 al vastgelegd hebben dat we een initieel systeem met een *shaping time* van 8 µs willen werken zodat we indien nodig genoeg flexibiliteit hebben om het aantal punten per puls aan te passen, evalueren we in dit hoofdstuk al deze zaken voor pulsen met verschillende *shaping times* tussen 1 µs en 8 µs (zodat de waarden van de *shaping times* van onze twee *shaping amplifiers* die de breedste gaussen geven hierin zitten) om een beter en breder beeld te krijgen van de invloed van deze parameter.

Al deze zaken bestuderen we door gekende referentiepulsen te laten verwerken door onze algoritmes, en nauwkeurig te bestuderen wat er gebeurt. Door het aantal gebruikte punten te variëren en verschillende implementaties van de trigger en fit algoritmes te gebruiken kunnen we onderzoeken wat de optimale configuratie van ons systeem is. We zouden al deze zaken

kunnen testen op de FPGA zelf; het is namelijk mogelijk om op een compacte manier input te simuleren op een FPGA. Dit pad is uitgeprobeerd en deels verworpen: Het uittesten van verschillende algoritmes, en het variëren van het aantal punten is een zeer tijdrovend proces omdat voor elke kleine verandering de gehele compilatie[1] opnieuw uitgevoerd moet worden.

De benadering waarvoor gekozen is, is om complementair aan een LABVIEW implementatie op de FPGA een implementatie in MATLAB te schrijven. De MATLAB code laat het variëren van de parameters toe, en de FPGA code is daar om te kijken naar zaken zoals snelheid en *resource usage*. Zo wordt de FPGA code bijvoorbeeld enkel gecompileerd voor een bepaald aantal punten per puls, terwijl we in MATLAB kunnen kiezen hoeveel punten we per puls uitkiezen. We kunnen niet genoeg benadrukken dat alles wat in MATLAB geïmplementeerd wordt volledig identiek is aan hetgeen op de FPGA moet draaien, de enige beperking vinden we in het feit dat de FPGA in *fixed point* (FXP) formaat zal werken, terwijl MATLAB het gebruik van *doubles* toelaat.

In dit hoofdstuk pogen we dus om aan de hand van simulaties van ons systeem met MATLAB de in de eerste paragraaf opgenoemde zaken te onderzoeken. Zodat er maar voor een enkel optimaal systeem firmware voor de FPGA gecompileerd moet worden.

5.1 Parametrisatie van de juistheid van de amplitude en de fit

Om de keuzes in dit hoofdstuk te onderbouwen hebben we een manier nodig om af te schatten hoe juist de via de fit afgeschatte amplitude is. In deze sectie gaan we op zoek naar een maatstaf die de juistheid van de fit, en belangrijker, de juistheid van de bekomen amplitude weerspiegelt.

5.1.1 RMSE

De maat voor de juistheid van de fit ligt voor de hand. Een kwadratische methode schat je af door middel van de kwadratische fout (4.2). Deze passen we lichtjes aan door te delen door het aantal punten waaraan deze fout wordt afgeschat, en de wortel te nemen, zo krijgen we de *Root Mean Square Error*:

$$\mathrm{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}[y_i - f(x_i)]^2} \tag{5.1}$$

Deze fout is makkelijk te bepalen in MATLAB, en moet –indien gewenst– zelfs met een relatief beperkte *resource usage* ook te implementeren zijn op de FPGA.

5.1.2 Standaard deviatie van de amplitude

Voor een betere maatstaf voor de voor de kwaliteit van ons systeem moeten we kijken naar onze doelstellingen. We willen spectra kunnen herkennen, en hebben dus een detector nodig met een goede lineariteit en een goede resolutie. We willen spectra reconstrueren aan de hand van een correcte bepaling van de amplitude (en niet de volledige vorm) van de discreet

[1] De tijdsduur van de compilaties varieert voor onze gehele systemen tussen de 15 en de 45 minuten.

gesamplede pulsen. Hiervoor moeten we een maatstaf invoeren die de juistheid van onze amplitude weergeeft.

De juistheid van de amplitude kunnen we onderzoeken door voor verschillende gekende pulsen met dezelfde amplitude de te bepalen, en te kijken wat voor amplitudes ons systeem hieruit distilleert. Uit zo'n experiment kunnen we twee interessante parameters en één controle parameter halen:

- De gemiddelde amplitude of *mean* vertelt ons hoe dicht onze algoritmes de amplitude van de ingevallen pulsen benaderen. Zo kunnen we ook beoordelen of er geen *bias* aanwezig is. En door deze te bekijken voor een aantal pulsen van verschillende amplitudes kan ook de lineariteit beoordeeld worden.
- De spreiding of standaarddeviatie op de gemiddelde amplitude weerspiegelt hoe groot de variatie in de waarden van de berekende amplitudes is. Dit geeft weer hoeveel de spectroscopiepieken verbreed worden door ons systeem, en is een maat voor de resolutie van de detector.
- Omdat we later in dit hoofdstuk genoodzaakt zijn deze statistiek uit te voeren voor een klein aantal samples voeren we een extra controleparameter in: het verschil tussen de twee verst uiteenliggende waarden van de amplitude, of de *range*. Dit is een indicatie voor de aanwezigheid van *outliers*[2] in onze samples, een gegeven dat duidt op een slecht statistisch sample.

5.2 De reactie van het systeem op gesimuleerde pulsen

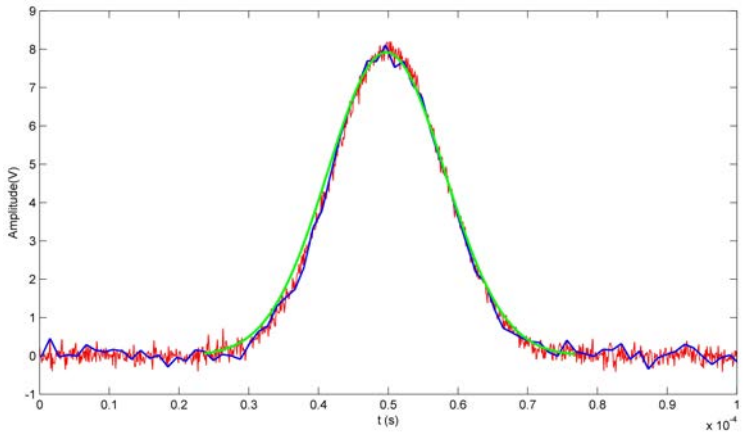
5.2.1 Pure gauss

De eerste basiscontrole die we moeten uitvoeren is nakijken hoe ons systeem reageert op een perfecte gaussische puls: Zoals verwacht reproduceert deze foutloos de fit en de amplitude (RMSE=0, standaarddeviatie=0) in zowel MATLAB als op de FPGA, en dit voor alle variaties in de trigger- en fit- algoritmes die we verder in dit hoofdstuk beschouwen.

5.2.2 Pure gauss + ruis

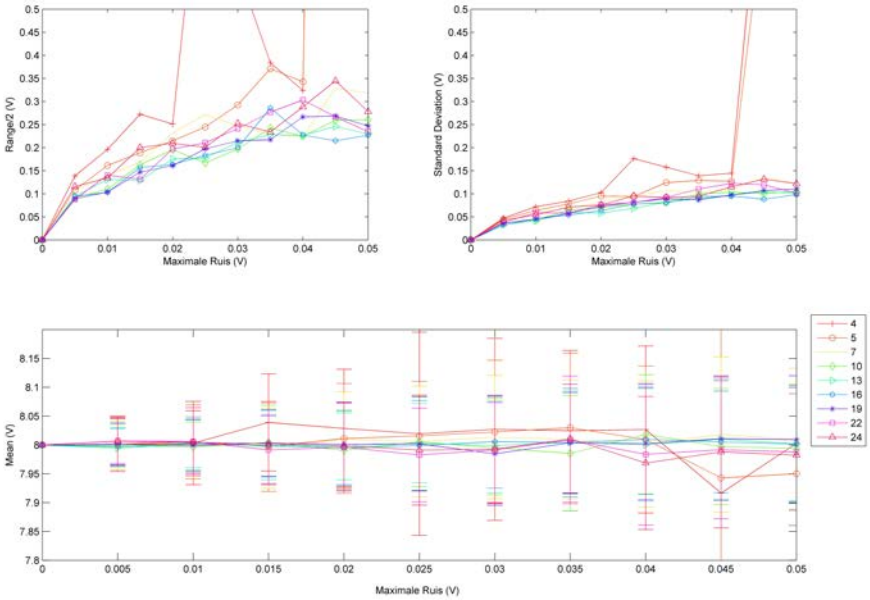
Om de invloed van ruis op ons systeem te bekijken voren we een tweede proef uit, waarbij we witte ruis toevoegen aan deze perfecte gaussische pulsen zoals weergegeven in Fig. 5.1. We zetten een simulatie op waar we de maximale uitwijking van normaal verdeelde ruis varieren, en kijken wat voor invloed dit heeft op het gemiddelde en de standaarddeviatie van de resulterende amplitudes. Om deze statistiek te vergaren zijn telkens 50 pulsen gegenereerd. Deze test voeren we uit voor verschillende aantallen punten en de verschillende triggeralgoritmes. Het resultaat bleek onafhankelijk te zijn van de keuze van triggeralgoritme, en is te zien in Fig. 5.2. Om de leesbaarheid te verbeteren zijn er puntenaantallen weggelaten in deze figuur, maar de waarden puntenaantallen volgen de globale tendens.

[2] *Outliers* zijn punten uit een gecorreleerd dataset die sterk afwijken van de rest van de punten in dit dataset.



Figuur 5.1: Een figuur van een gesimuleerde gaussische puls (rode lijn) met een amplitude van 8 V en een *shaping time* van 8 µs. De maximale uitwijking van de toegevoegde ruis is 0.4 V. De blauwe lijn geeft de punten die de 750 kHz ADC hieruit zal samplen weer, en de groene lijn is het resultaat van een gaussische fit door middel van ons algoritme.

In deze figuur zien we dat de ruis voor alle puntenaantallen behalve 4 en 5 nauwelijks invloed heeft op de gemiddelde amplitude. De instabiliteit van de fits aan 4 en 5 punten is ook te zien aan de hoge en onregelmatige standaarddeviatie. Voor hogere puntenaantallen wordt de ruis echter meer uitgemiddeld en blijven de standaarddeviaties laag, en krijgen we een bijna lineair verband tussen de maximale uitwijking van de ruis en de standaard deviatie. Het verschil voor verschillende aantallen punten is ook verwaarloosbaar, vanaf 6 punten bereiken we al een met grotere puntenaantallen vergelijkbare ruisonderdrukking.



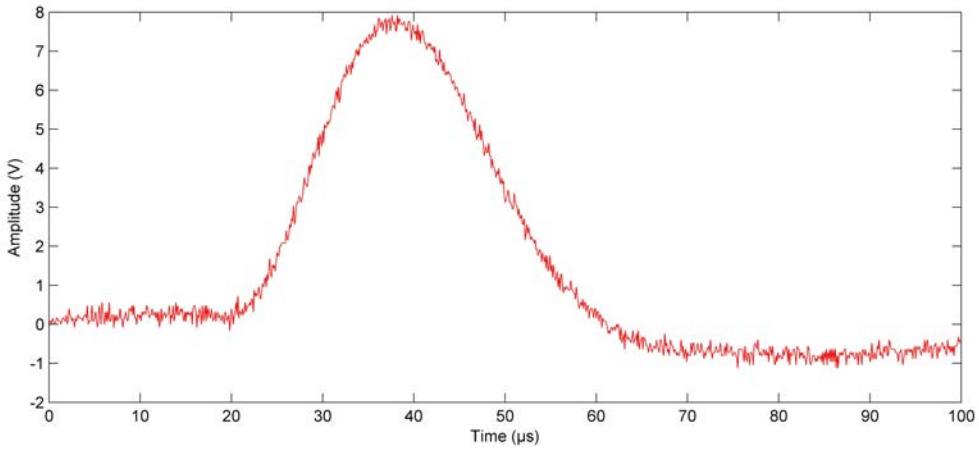
Figuur 5.2: Deze figuur is het resultaat van een studie van de invloed van ruis op ons finale algoritme. Voor een gesimuleerde gaussische puls met een *shaping time* van 8 µs, en een amplitude van 8 V wordt de resulterende gemiddelde amplitude gegeven, alsook de standaarddeviatie en de range hiervan. De foutenvlaggen worden gegeven door de standaarddeviatie.

5.3 De reactie van ons systeem op echte testpulsen

De volgende stap in de studie van ons systeem is een onderzoek in de omstandigheden die we vinden in onze opstelling (zie 3.2). Aan de hand hiervan kunnen we de performantie van onze algoritmes in onze gammaspectroscopieopstelling evalueren. We moeten dus de pulsen die uit de *shaping amplifier* komen als input voor onze simulaties gebruiken.

5.3.1 Pseudo-experimenten met echte testpulsen

In het optimale geval zouden we de pulsen die uit de *shaping amplifier* komen moeten kunnen simuleren. We zijn er echter niet in geslaagd om de invloed van de onvolledige *pole-zero* correctie na te bootsen. Daarom zijn we genoodzaakt om voor een andere aanpak te kiezen: we gebruiken een set van testpulsen (vb. Fig. 5.3), opgenomen door een oscilloscoop, en onderzoeken hoe ons systeem hierop reageert. Dit stelt echter een probleem: als we dezelfde statistiek willen vergaren om de gemiddelde amplitude en de standaardafwijking hierop te bepalen moeten we met de oscilloscoop een set pulsen opmeten met exact dezelfde amplitude. Dit is onmogelijk, omdat we de amplitude niet exact genoeg met de oscilloscoop kunnen bepalen. We kunnen met de oscilloscoop bijvoorbeeld wel het maximum in het signaal bepalen, maar dit is niet exact genoeg; als controlewaarde voor de gemiddelde amplitude geven we deze wel altijd met de figuren mee.



Figuur 5.3: Een testpuls opgenomen met onze opstelling, met een *shaping time* van 8 µs. De maximale amplitude bepaald met de oscilloscoop is 7.92 V.

Er is echter een tweede mogelijkheid om een statistisch sample te vergaren: omdat de *sample rate* van de oscilloscoop (10 kHz) hoger ligt dan de *sample rate* van de ADC (750 kHz) die wij gebruiken bevat elke puls meerdere sets punten die representatief zijn voor dezelfde opgemeten puls. Dit is equivalent aan de situatie waarin eenzelfde testpuls op verschillende tijdstippen invalt in de ADC. Uit één testpuls met een 8 µs *shaping time* kunnen we nu een set van (10 MHz)/(750 kHz) $\approx$ 13 samples distilleren om statistiek op uit te voeren. We moeten er dus rekening mee houden dat we met een klein statistisch sample werken. Om beter te kunnen bestuderen of dit sample geen vreemde karakteristieken vertoont is dan ook de *range*

als controleparameter toegevoegd in de proeven waarbij we maar over deze kleine ensembles beschikken.

De gebruikte testpulsen: Met een oscilloscoop zijn tien verschillende testpulsen opgenomen, met amplitudes tussen 1 V en 8 V, en een *shaping time* van 8 µs. Een voorbeeldpuls is gegeven in figuur 5.3, en een tweede voorbeeld vind je in Fig. A.1. De aanwezige ruis is afhankelijk van de interne afsluitweerstand van de oscilloscoop: voor pulsen onder de 4 V is de maximale uitwijking van de ruis ± 0.05 V en voor de pulsen met amplitudes tussen 4 V en 10 V is deze ± 0.25 V. De drempelwaarde van de trigger voor de proeven in de rest van dit hoofdstuk ligt dan ook niet toevallig op 5 V, veilig boven deze ruisniveaus.

In dit hoofdstuk illustreren we onze bevindingen met de resultaten van één enkele testpuls om het zo duidelijk mogelijk te maken. Alle vermelde resultaten zijn zijn het gevolg van studies die een consistent resultaat opleverden voor elk van deze 10 pulsen. Om een kritische lezer te overtuigen is voor alle verdere figuren in dit hoofdstuk een figuur van een corresponderende tweede testpuls in de appendix A opgenomen.

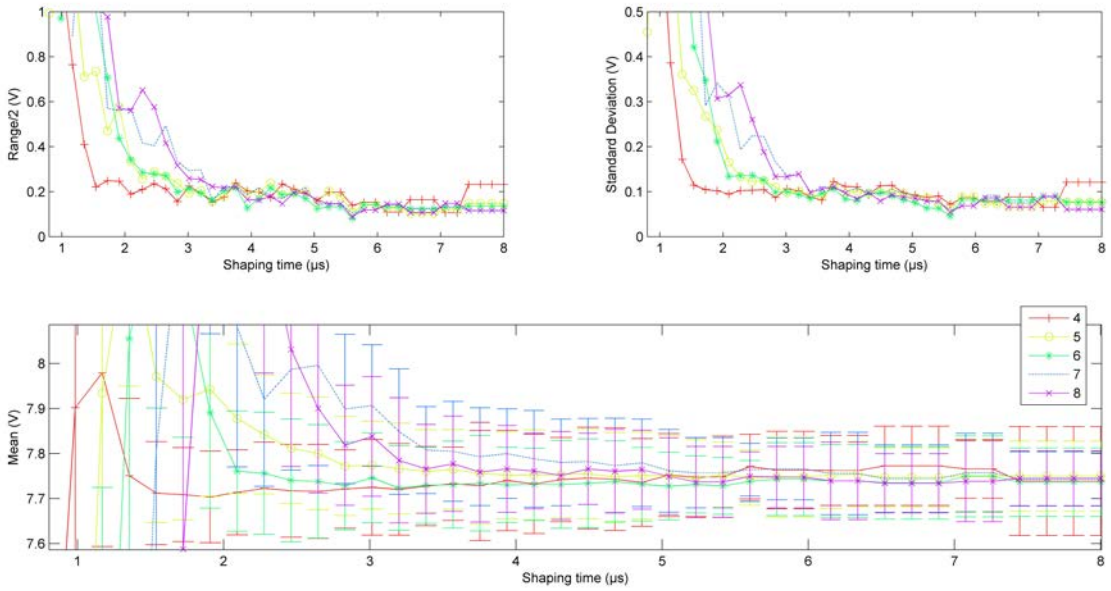
Variaties van de *shaping time* In het begin van dit hoofdstuk hebben we ook vermeld dat we graag zouden onderzoeken hoe ons systeem reageert op pulsen met verschillende *shaping times*. Omdat we onze pulsen niet kunnen simuleren zullen we ook hier met dezelfde echte testpulsen moeten werken. We beschikken echter enkel over *shaping amplifiers* met *shaping times* van 2 µs en 8 µs, en willen eigenlijk kijken naar het hele gebied tussen deze twee *shaping times*. De methode die we hiervoor gebruiken is eenvoudig: we nemen testpulsen met *shaping times* van 8 µs, en comprimeren deze naar testpulsen met andere *shaping times* door een lineaire transformatie van de tijdscoördinaat van onze testpulsen uit te voeren. Voor een gaussische puls is dit equivalent aan een transformatie van de parameter σ (zie uitdrukking 4.1).

Een bijkomende zaak is dat we bij deze kunstgreep de *sample rate* van de gemeten testpuls eigenlijk kunstmatig verhogen. Voor kleinere *shaping times* zal in onze proeven dan ook het aantal samples om statistiek op uit te voeren stijgen. Van de 13 samples bij 8 µs stijgen we naar 52 samples bij 2 µs.

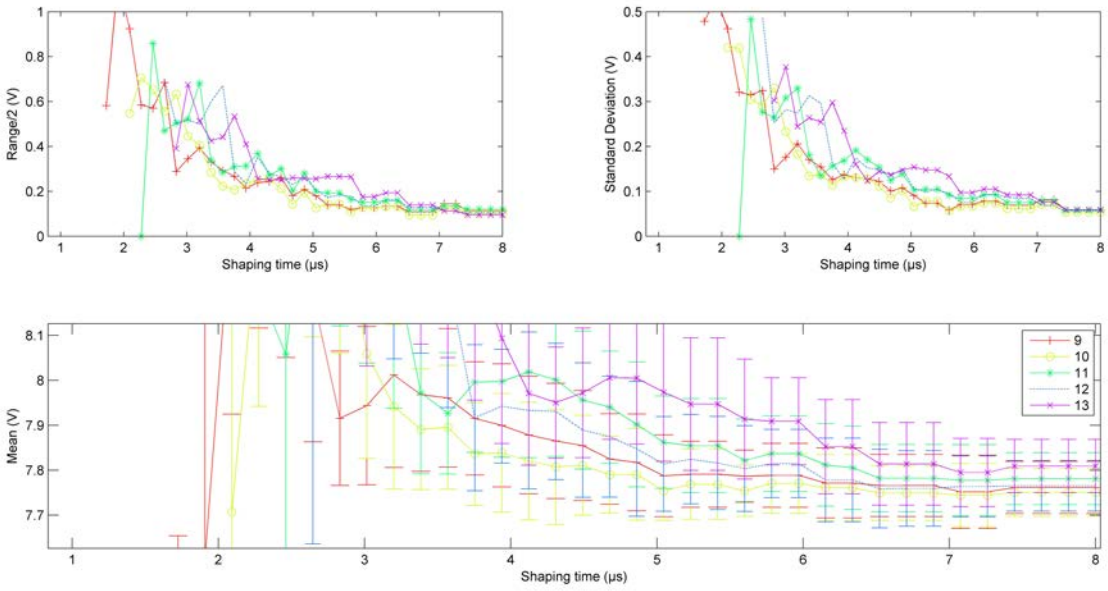
5.3.2 De kwaliteit van ons systeem voor echte testpulsen

Nu we een kader hebben gecreëerd waarin we onze algoritmes kunnen onderzoeken in realistische omstandigheden willen we in de volgende paragrafen de kwaliteit van de amplitude van ons finale systeem aftoetsen aan de hand van de voorgestelde parameters. Omdat we pas later in dit hoofdstuk de keuze voor een optimaal trigger- en fit-algoritme specifiëren, en nu al willen kijken naar ons finale systeem moeten we even vooruitlopen op de feiten. De uitweiding over de in de figuren vermeldde algoritmes zijn te vinden in secties 5.4 en 5.5.

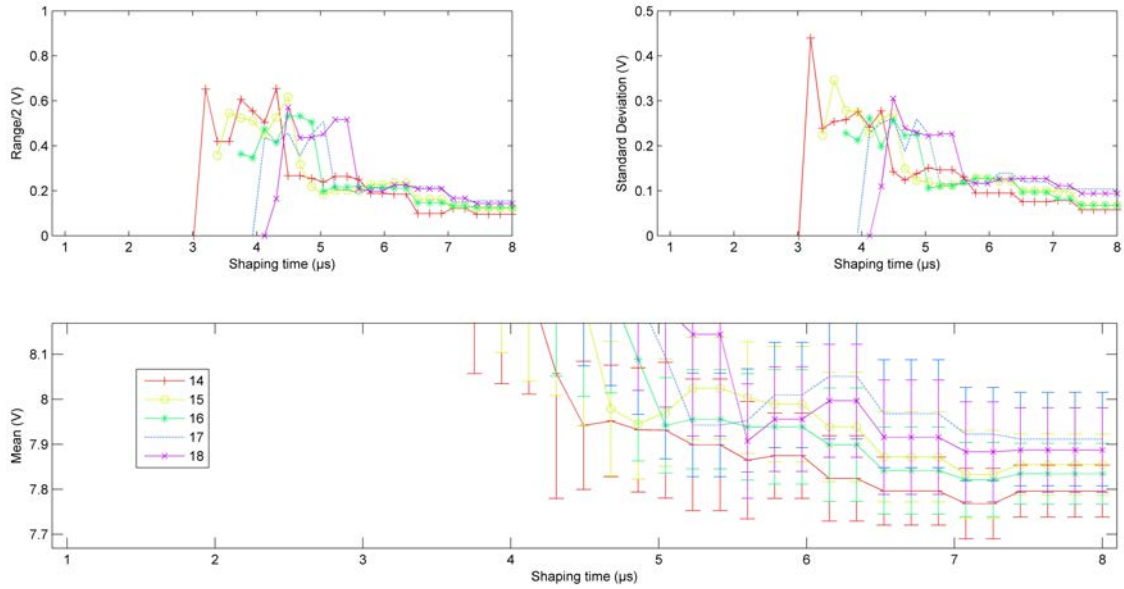
In Fig. 5.4, Fig. 5.5 en Fig. 5.6 zien we de resultaten van onze studies van de amplitudes, toegepast op de testpuls in 5.3. Eerst en vooral zien we dat de waarden van de *range* vergelijkbaar zijn met de waarden van de standaard deviatie. Voor de meeste metingen zijn beide waarden relatief klein ten opzichte van de gemeten gemiddelde amplitude, wat resulteert in een kleine relatieve fout, het vertrouwen in onze metingen bevestigt.



Figuur 5.4: We zien hier de resultaten van de toepassing van onze algoritmes voor 4 tot 8 punten op één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.



Figuur 5.5: We zien hier de resultaten van de toepassing van onze algoritmes oze algoritmes voor 9 tot 13 punten op één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.



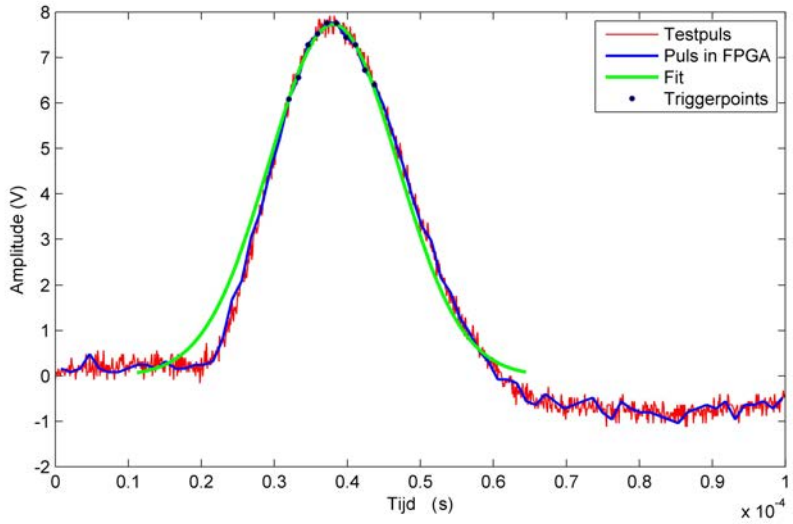
Figuur 5.6: We zien hier de resultaten van de toepassing van onze algoritmes voor 14 tot 18 punten één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.

Als eerste resultaat zien we hier dat we, bij een *shaping time* van 8 µs, voor alle puntenaantallen een gelijkaardige waarde voor de amplitude krijgen, netjes in de buurt van de maximale amplitude gemeten in deze testpuls van 7.92 V. Ons systeem lijkt dus over het hele gebied een resultaat zonder bias op te leveren. De parameter die de kwaliteit van de resolutie indiceert, de standaarddeviatie, ligt rond de waarde van 0.1 V. Dit levert een relatieve fout op van minder dan 2% van de pulshoogte. De beste resultaten voor de standaarddeviatie vinden we voor puntenaantallen tussen 8 en 14 punten. Deze resultaten zijn ook vergelijkbaar met de standaarddeviaties bekomen in onze studies van de invloed van ruis op ons systeem (zie Fig. 5.2). De ruis lijkt dus de voornaamste oorzaak voor de spreiding van de amplitude in ons systeem te zijn.

Verder zien we ook dat deze bevindingen standhouden voor kortere *shaping times* tot op een gegeven moment de gemeten amplitudes beginnen te fluctueren, en de standaarddeviatie de hoogte in schiet. Voor grotere puntenaantallen valt dit bij langere *shaping times* voor dan bij kleinere puntenaantallen. Dit gebeurt op de moment waarop het aantal meetpunten nog maar net in de puls past. Voor nog kortere *shaping times* wordt het onmogelijk om met dit aantal meetpunten de puls te samplen.

5.4 Keuze van de trigger

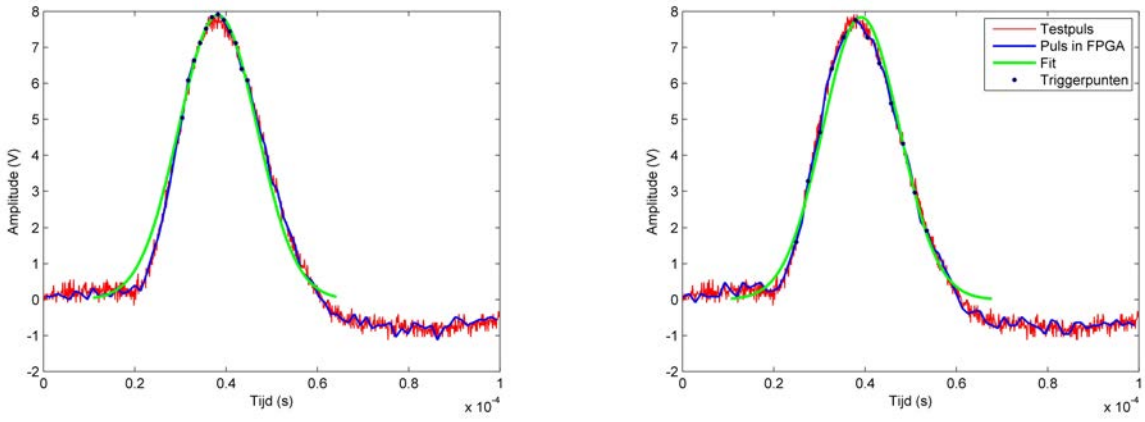
Nu we over het juiste gereedschap beschikken kunnen we ook bekijken wat voor trigger we willen gebruiken. De trigger selecteert de punten die we gebruiken om onze fit uit te voeren, zoals weergegeven in Fig. 5.7.



Figuur 5.7: Dit is het resultaat van een fit van 10 punten die wordt uitgevoerd op de puls van Fig. 5.3. De trigger pikt er 10 punten uit: eerst en vooral het maximum, de 4 punten die het dichtst voor het maximum liggen, en de 5 punten die er het dichtst achter liggen.

5.4.1 De distributie van de triggerpunten over de puls.

In ons systeem moeten we eerst een puls detecteren: de aangewezen manier om dit te doen is door op zoek te gaan naar een maximum in het inkomende signaal. Om foute triggers als gevolg van ruis te onderdrukken kiezen we er tegelijk ook voor om een minimaal triggerniveau van 0.5 V te implementeren. Wanneer een puls gedetecteerd is liggen twee voor de hand liggende opties op ons te wachten: ofwel kiezen we een vast aantal punten rond een maximum in het signaal (dit noemen we de normale of gewone trigger), ofwel kiezen we ervoor om het vaste aantal punten zoveel mogelijk over de puls te verspreiden (dit noemen we de gedistribueerde trigger). Omdat de implementatie van de fit equidistante punten vraagt zijn deze opties niet altijd heel verschillend[3], maar het is de moeite waard om dit te bekijken omdat het gebruik van minder punten minder resources van de FPGA zal vragen. Op Fig. 5.8 kun je het verschil tussen beide methodes bekijken.



Figuur 5.8: Het verschil tussen de normale/gewone trigger die punten zo dicht mogelijk rond het maximum kiest (links), en de gedistribueerde trigger die de punten zoveel mogelijk over de puls verspreid(rechts). Deze triggers en fits gebruiken 12 punten, en de linkse trigger kiest een punt meer na het maximum dan ervoor indien het over een even aantal punten gaat.

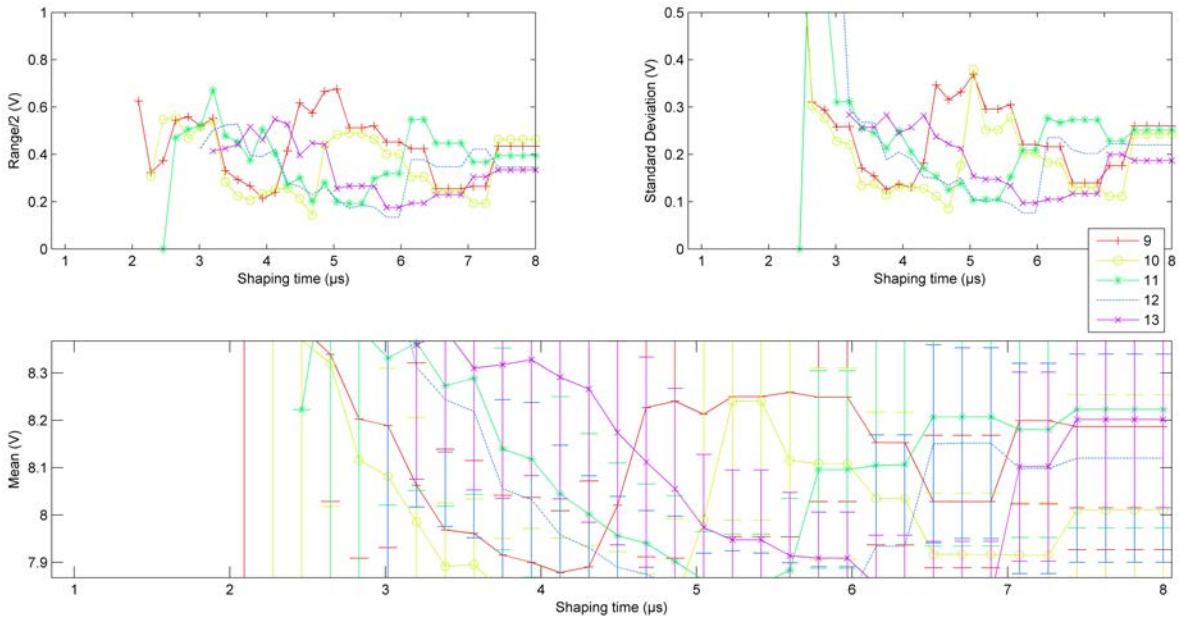
[3]Als het aantal punten dat we gebruiken om te fitten meer dan de helft is van het aantal punten dat boven het triggerniveau ligt is geeft de equidistante optie dezelfde als deze van een gewone trigger. Bij een *shaping time* van 8 µs werken kunnen we een maximum van ongeveer 24 punten per puls boven het triggerniveau vinden, dus vanaf 13 punten lopen deze opties gelijk.

10 Punten	Testpuls 1			Testpuls 2			Testpuls 3			Testpuls 4		
	Mean	Range	StdDev	Mean	Range	StdDev	Mean	Range	StdDev	Mean	Range	StdDev
Normale Trigger	2.632	0.094	0.048	5.812	0.143	0.075	6.388	0.115	0.079	7.751	0.103	0.053
Gedistribueerde Trigger	2.632	0.083	0.054	5.865	0.122	0.077	6.485	0.294	0.165	8.010	0.463	0.243

7 punten	Testpuls 1			Testpuls 2			Testpuls 3			Testpuls 4		
	Mean	Range	StdDev	Mean	Range	StdDev	Mean	Range	StdDev	Mean	Range	StdDev
Normale Trigger	2.632	0.104	0.059	5.819	0.165	0.085	6.392	0.128	0.076	7.743	0.115	0.060
Gedistribueerde Trigger	2.661	0.201	0.101	6.090	0.424	0.233	6.434	0.389	0.219	8.177	0.459	0.289

Figuur 5.9: Het gemiddelde van de amplitude, en de range en de standaarddeviatie hiervan, wordt hier voor verschillende testpulsen voor 7 en voor 10 punten weergegeven. En dit voor een systeem met een gewone trigger, en een trigger die de punten zoveel mogelijk distribueert.

Tabel 5.9 en Fig. 5.5 en Fig. 5.10 illustreren het resultaat dat we vinden over het geheel van alle testpulsen en puntenaantallen: Met als criterium de standaarddeviatie van de amplitude presteert het systeem met de gedistribueerde trigger slechter, en in het beste geval hetzelfde als datgene met de gewone trigger. We nemen ook een tendens waar: voor meer punten wordt dit verschil kleiner, en voor testpulsen met een kleinere amplitude wordt het verschil ook kleiner. Dit is niet onlogisch vermits we in beide gevallen minder punten boven het triggerniveau vinden, waardoor het resultaat van de triggers in meer gevallen hetzelfde zullen zijn. Bovendien presteert de normale trigger ook beter als we naar de gemiddelde amplitude kijken: in functie van de *shaping time* deze levert een resultaat op dat stabieler is, en consistenter is over de algoritmes met verschillende aantal punten. We kunnen concluderen dat een normale trigger een beter resultaat oplevert. De keuze is dus duidelijk, met als bijkomend voordeel dat een normale trigger ook minder logica van de FPGA zal gebruiken.



Figuur 5.10: We zien hier de resultaten van de toepassing van onze algoritmes voor 9 tot 13 punten op één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier is gebruikt is de gedistribueerde trigger.

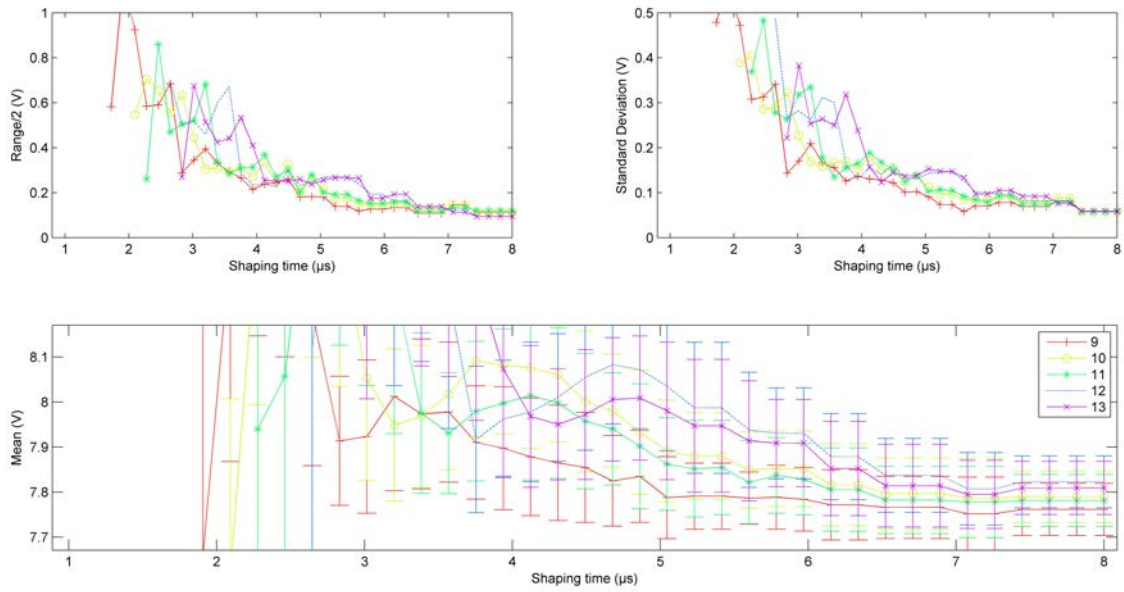
Samengevat blijkt hier –net als uit de laatste paragraaf van sectie 5.3.2– dat je beter punten dicht bij het maximum sampled dan uit de hele puls om een goed resultaat voor de amplitude te verkrijgen. Dit resultaat hoeft niet zo opmerkelijk te zijn: de vervorming van de curve (zie Fig. 5.3) ten opzichte van een perfecte gauss is groter in de staarten dan dichtbij het maximum.

5.4.2 Verdere optimalisaties van de trigger

Naast de belangrijke optimalisatie die we in de vorige paragrafen besproken hebben zijn er nog twee andere situaties die we onderzocht hebben.

Pre- en post-triggering: Een idee om de gedistribueerde trigger beter te doen werken bestond eruit om een vorm van pre en posttriggeren in te voeren. Dit bestaat eruit om –indien nodig– de trigger toe te laten een aantal punten voor het overschrijden van het triggerniveau toch mee te nemen, en een aantal punten toch mee te nemen nadat het signaal terug onder het triggerniveau zou vallen. Het voordeel is dat een puls zo meer punten bevat, en er dus meer equidistante opties ontstaan. Dit is uitgetest voor zowel één als twee punten ervoor (pre) en erachter (post), en de mogelijke combinaties hiervan. De invloed van deze aanpassing op de standaarddeviatie en de gemiddelde amplitude bleken verwaarloosbaar.

De verdeling van de punten rond het maximum: Bij de normale trigger zou pre- en post-triggeren hetzelfde resultaat opleveren als het veranderen van het triggerniveau, en levert ons dus enkel onnodige complicaties op. Er is echter wel nog een ander gegeven dat we moeten bekijken: bij een even aantal punten blijft er na het uitzoeken van het maximum een oneven aantal punten over om te verdelen voor en na het maximum. De trigger moet dus uitmaken of er beter één punt meer voor het maximum gekozen wordt, of erachter. In het geval van een perfecte –en dus symmetrische– gausscurve zou dit geen verschil mogen maken, maar bij de lichtjes verdraaide curve waar wij mee werken is dit niet het geval. Een vergelijkende blik op Fig. 5.5 en Fig. 5.11 leert ons dat de verschillen verwaarloosbaar zijn.



Figuur 5.11: We zien hier de resultaten van de toepassing van onze algoritmes voor 9 tot 13 punten op één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier gebruikt is kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer voor het maximum genomen dan erna.

We maken dan de arbitraire keuze voor een systeem met één punt meer na het maximum. Een argument dat voor dit soort trigger spreekt is het feit dat de pulsen geen perfecte gaussfuncties zijn, maar dat de *rise time* iets korter is dan de *fall time* van de puls. Als we naar zeer korte pulsen kijken wordt het dus ook logisch om meer punten na het maximum te kiezen dan ervoor.

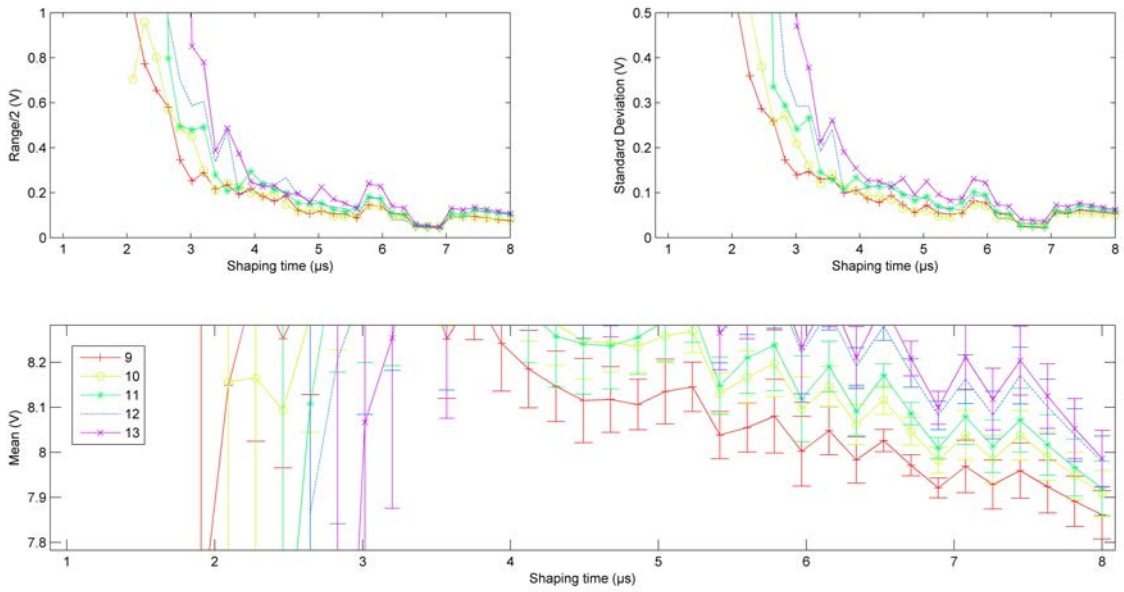
5.5 σ vastleggen?

Vermits we voor onze FPGA in de regel een *shaping amplifier* met een gekende *shaping time* plaatsen kennen we in principe ook de breedte σ van de puls die binnenkomt op onze FPGA. Als we deze waarde nu als vast beschouwen wordt ons algoritme nog eenvoudiger. Het eerste voordeel hiervan is het feit dat meer eenvoud minder *resource usage* op de FPGA betekent. Een tweede potentieel voordeel van het werken met een vaste σ parameter zou een stabielere en potentieel meer precieze bepaling van de amplitude zijn.

Vergelijkende proeven (zoals Fig. 5.5 en Fig. 5.12) met een aangepast algoritme leveren echter een duidelijk resultaat op: de standaard deviatie geeft een gelijkaardig, en voor sommige *shaping times* zelfs een beter resultaat. Dit is niet onverwacht omdat we met deze extra informatie de ruis sterker zullen onderdrukken.

De gemiddelde amplitude is echter niet erg stabiel voor variaties in de *shaping time*[4]. En voor

[4]Hierbij wordt de σ parameter die het algoritme moet gebruiken natuurlijk ook aangepast.



Figuur 5.12: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 7.92 V (Fig. 5.3). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor. Het fit-algoritme maakt hier gebruik van de gekende waarde van σ, die wordt geleverd door de *shaping time* van de *shaping amplifier*.

verschillende aantallen punten varieert de gemiddelde amplitude sterk in vergelijking met het andere fit-algoritme. Omdat de standaard deviatie toch al laag lag voor een systeem met een variabele σ kiezen we dan ook voor de optie waarin de gemiddelde amplitude het meest consistente resultaat levert voor verschillende puntenaantallen.

De vraag die zich nu stelt is waarom de fit waarin we meer juiste informatie meenemen minder consistente waarden voor de amplitude oplevert dan deze waarin meer parameters mogen variëren. Het antwoord is te vinden in het feit dat de informatie over de pulsbreedte niet exact juist is, een *preamplifier* levert geen exact gaussische vorm, maar een gausscurve waarvan het zwaartepunt iets meer naar de kant achter het maximum verschoven is. De beste pulsen die uit onze opstelling komen zijn ook niet volledig 'pole-zero' gecorrigeerd, een factor die extra problemen oplevert. Deze verstoringen van de perfect gaussische vorm zijn nooit onafhankelijk van de amplitude van de puls, wat het gebruik van een vaste waarde voor σ bemoeilijkt.

Het rest ons nog te rapporteren dat pogingen om fits met een lichtjes aangepaste breedte (groter en kleiner) ook geen betere resultaten opleverden.

5.6 Aantal punten

Nu we over een geoptimaliseerd systeem beschikken moeten we nog een laatste beslissing nemen: welk vast aantal punten gaan we gebruiken voor ons eerste systeem?

Een kwadratische fit heeft drie onbekenden, we zullen dus in eender welke methode minstens drie punten nodig hebben om de fit te kunnen uitvoeren. Het aantal punten moet voorktomen uit een evenwicht tussen de gevraagde verwerkingssnelheid, de complexiteit[5] op de FPGA en de benodigde precisie. In het geval dat we in dit proefschrift als eerste willen belichten werken we met een pulsbreedte van 8 µs, wat ons meer dan genoeg tijd geeft om de bewerkingen op de FPGA uit te voeren. De verwerkingssnelheid zal het aantal punten niet limiteren, en de vraag of de FPGA de complexiteit aankan is een gegeven dat we enkel kunnen testen door een implementatie te maken.

Zoals al gebleken in sectie 5.3.2 zijn de verschillen tussen de verschillende aantallen punten niet groot. Toch proberen we de keuze van een initieel aantal punten te doen door te zoeken naar een puntenaantal dat doorheen alle proeven goede resultaten levert. Een eerste eis is een betrouwbare ruisonderdrukking (zie Fig. 5.2). Daarnaast gaan we in eerste instantie op zoek naar een puntenaantal dat een resultaat met lage standaarddeviatie voor 8 µs *shaping time* levert. In tweede instantie kunnen we bekijken welk aantal een stabiele gemiddelde amplitude oplevert voor verschillende *shaping times*, zodat ons systeem in de toekomst makkelijk overgezet kan worden naar systemen met andere *shaping times*. Als we naar Figs. 5.5, 5.4, 5.6 kijken, is tien meetpunten is een aantal waar deze resultaten goed zijn. Bovendien is dit een aantal punten dat in de lagere helft van de mogelijke aantallen punten (4-24) ligt, wat metingen van kortere pulsen ook toelaat.

Dit puntenaantal is een goede basis om mee te beginnen, het zal ook interessant zijn om later het puntenaantal te verlagen tot de kortst mogelijke puls die we op een betrouwbare manier kunnen meten. Op deze manier gaat de verwerkingstijd van één puls op de FPGA omlaag. En ook de kans dat de pulsen van twee deeltjes die vlak na elkaar gedetecteerd worden samenvallen (zie hoofdstuk 10) gaat omlaag. Hiervoor is ons minimaal aantal punten (4) al ineens een goede kandidaat. Dit geeft voor alle testpulsen een iets hogere standaarddeviatie dan het geval van 10 punten, maar lijkt een stabiele weergave te geven tot een *shaping time* rond de 2 µs. Een bijkomend belangrijk nadeel is dat ons systeem van 4 punten de ruis minder onderdrukt zoals gebleken is in Fig. 5.2. Toch is een systeem met vier punten een interessant gegeven vermits we over een *shaping amplifier* met een *shaping time* van 2 µs beschikken.

[5] Meer punten betekent grotere *fixed point* getallen op de FPGA, een gegeven dat zeer snel kan groeien.

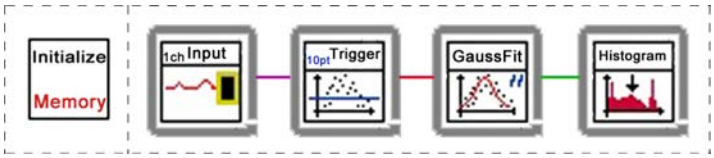
Hoofdstuk 6

Implementatie op FPGA

In dit hoofdstuk beschouwen we de implementatie van de FPGA code in LABVIEW. De implementatie wordt besproken en becommentarieerd, waarbij de nadruk ligt op de vertaling van de eerder besproken structuren voor de fit en de trigger naar de FPGA. Enkele technische details, die voor de werking en de *resource usage* van de implementatie belangrijk zijn, hebben we verschoven naar het volgende hoofdstuk. Dit omdat, hoewel er veel tijd van het ontwikkelingsproces in deze details gestoken is, deze details niet essentieel zijn voor het begrijpen van onze implementaties.

6.1 Globale structuur

De globale structuur van onze implementatie wordt geïllustreerd door Fig. 6.1. Er wordt gewerkt met vier parallel werkende blokken die verschillende functies uitvoeren: De input zorgt ervoor dat de data die de ADC genereert in de juiste volgorde (in de tijd) op de FPGA terecht komt. De resterende drie blokken staan respectievelijk in voor (i) het triggeren van de data, (ii) het bepalen van het maximum, en (iii) het opslaan en aanbieden van de data voor een grafische weergave van het spectrum (het histogrammen).



Figuur 6.1: Een weergave van de structuur van de programmatuur op de FPGA op het hoogste niveau. Na de initialisatie van de geheugens en FIFO's zijn de vier parallel werkende hoofdblokken: Input, Trigger, Gaussische Fit en Histogramming. De gekleurde lijnen die deze verbinden duiden de FIFO's aan die de benodigde data doorsturen en bufferen.

In de hiernavolgende secties behandelen we elk van deze blokken apart, maar vooraleer we hier dieper op ingaan moeten we de communicatie tussen deze verschillende blokken bespreken. Om een maximale verwerkingssnelheid te verkrijgen willen we graag dat deze blokken parallel functioneren, maar deze parallelle werking veroorzaakt een communicatieprobleem. We moeten ervoor zorgen dat elk blok op elk moment een output kan leveren, zonder te moeten wachten totdat het volgende blok dat de data moet verwerken klaar is met zijn bewerkingen.

Om dit te begrijpen moeten we kijken naar de data flow in het systeem: Dit begint bij de input, deze geeft aan een vaste –van de gebruikte ADC(s) afhankelijke– rate data door aan de trigger. De trigger pikt uit dit signaal enkel de pulsen eruit, in eerste instantie zal de trigger de *data rate* (afhankelijk van de intensiteit van de gemeten gammastraling) verminderen. We hoeven van de fit –die de ingewikkeldere, en dus langer durende, berekeningen bevat– dan ook niet te eisen dat deze aan dezelfde *data rate* als de trigger werkt. Maar, vermits de pulsen op onregelmatige tijdstippen toekomen zal de *rate* van de data die de trigger naar de fit stuurt variëren. Het is dus een goed idee om deze variaties uit te middelen door de data te bufferen. Dit wordt op de FPGA geïmplementeerd door het gebruik van *First In First Outs* (FIFOs); dit zijn geheugencomponenten die de data bufferen. Sterker nog, omdat dit een probleem is dat zo vaak voorkomt is het gebruik van FIFOs de standaard manier in LABVIEW voor FPGAs geworden om te communiceren tussen verschillende parallelle systemen. De compiler is zo ingesteld dat een FIFO zelfs bij constante *data rates* in de meeste gevallen de snelste implementatie op de FPGA waarborgt.

Verder is er de mogelijkheid om het aantal elementen dat in een FIFO past te kiezen. Dit is een gegeven dat je best kunt bepalen door te kijken naar de variaties in de *data rate* en de snelheid waarmee de input en output van de FIFO data levert en verwerkt. In ons geval is er een initiële keuze gemaakt voor elk van de gebruikte FIFOs door een klein aantal elementen (100 of 1023, afhankelijk van de te verwerken *data rate*) te kiezen. 1024 is gekozen omdat het een door LABVIEW aangerijkte standaardwaarde is, en 100 wordt gebruikt op plaatsen waar de *data rate* 10 maal lager ligt. Deze waarden kunnen indien nodig nog geoptimaliseerd worden aan de hand van de resultaten van de timing experimenten in hoofdstuk 9. In dit proefschrift hebben wij hier echter geen noodzaak voor gezien, en aldus ook niet uitgevoerd.

Een tweede belangrijk gegeven bij het gebruik van FIFOs is de vraag wat er moet gebeuren als een FIFO vol zit. Hier zijn drie mogelijkheden: (i) ofwel maken we plaats door een element weg te gooien, (ii) ofwel schrijven we ons element niet weg, (iii) ofwel wachten we totdat de FIFO terug plaats heeft om ons element te kunnen opslaan. Deze laatste methode is de methode die gebruikt wordt in alle FIFOs die gebruikt worden bij de communicatie tussen parallel lopende stukken code. Dit omdat dit de enige methode is die garandeert dat er geen punten verloren gaan tussen de verschillende blokken. Als we hierop niet zouden kunnen rekenen zou er een basic *error handling* mechanisme dat fouten in deze *data flow* detecteert en oplost noodzakelijk worden. In de FPGA cursus [12] wordt aangeraden om *error handling* tot het minimum te beperken, omdat dit in de regel veel resources gebruikt. In ons geval is de keuze dan eenvoudig, zeker omdat bij de twee andere alternatieven evenveel data verloren gaat. Ondanks dit alles blijft het beter om deze situaties te vermijden door de verwerkingsstappen snel genoeg te maken, en indien nodig voldoende grote buffers te voorzien.

Als laatste willen we er ook nog even op wijzen dat de *data rate* in de verschillende stappen van het dataverwerkingsproces gereduceerd wordt: Zoals eerder vermeld brengt de trigger de *input rate* -die afhankelijk is van de ADCs- al terug, omdat enkel de punten die deel uit maken van pulsen geselecteerd worden. De *data rate* die het triggeralgoritme levert is gelijk aan tien keer de *pulse rate* vermits de trigger per (getriggerde) puls tien punten levert. Vervolgens distilleert de fit uit deze tien punten één waarde voor het maximum, de *data rate* die naar het histogramming blok wordt gestuurd is dus gelijk aan de *pulse rate*, en één tiende van de *data rate* die de trigger output. Deze laatste reductie van de *data rate* is de reductie waar in

de paragraaf over FIFO's over gesproken wordt.

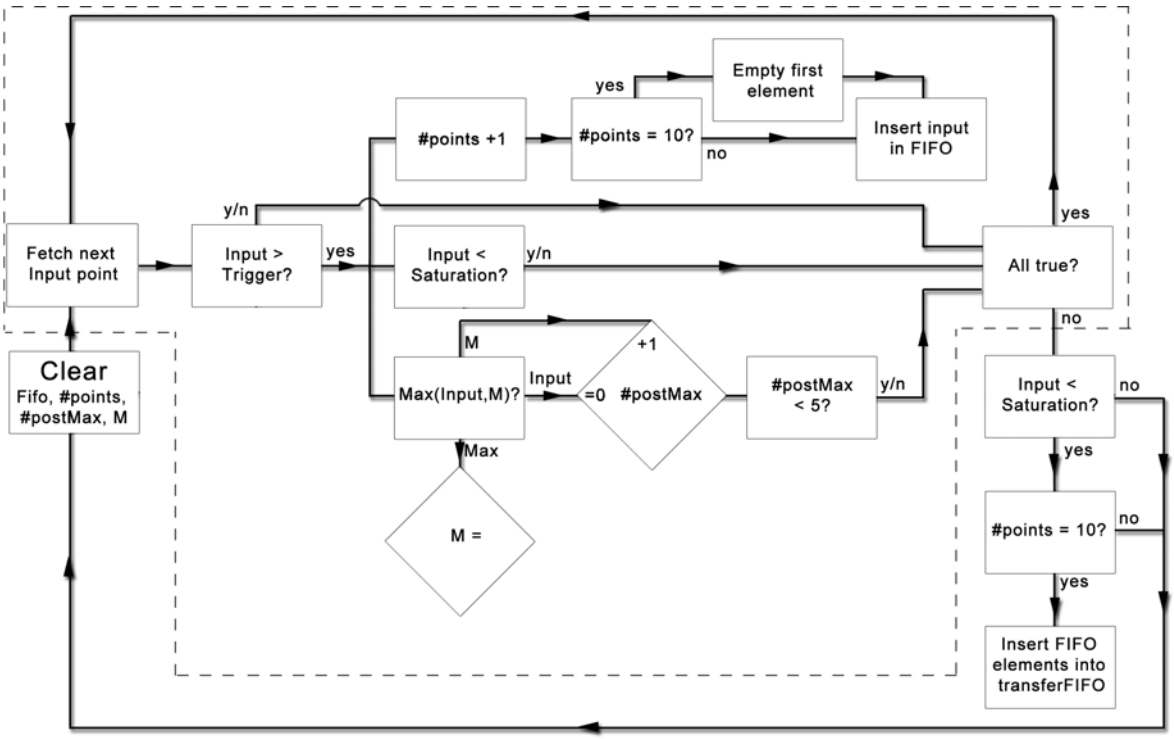
6.2 Input

Onze FPGA bestaat uit acht 750 kHz ADCs, die we voor een maximale *acquisition rate* liefst parallel gebruiken, met een delay van 167 ns tussen de meetpunten die elke opeenvolgende ADC neemt. Het is echter niet mogelijk om de timing van individuele ADC's af te regelen op de FPGA zelf, deze nemen gewoon data volgens een vaste klok die voor alle ADCs hetzelfde is. Een oplossing hiervoor is het aanleggen van een externe *delay line*, waarna de input van de verschillende ADCs op de FPGA zelf enkel nog in de juiste volgorde gezet moeten worden. De programmatie van de FPGA is in dit geval eenvoudig, maar het opzetten van een *delay line* is een technisch gegeven dat niet per se noodzakelijk is voor het ontwikkelen van de spectroscoop en is niet uitgevoerd in het kader van dit proefschrift. Ons input blok bestaat dus enkel uit het uitlezen van één enkele ADC naar de FIFO die deze data naar de trigger aanvoert.

Vermits de ADC aan een constante rate data levert zouden we één FIFO kunnen besparen door de ADC rechtstreeks in het FIFO blok uit te lezen. Er is echter voor gekozen om de Input en Trigger blokken uit elkaar te houden om veranderingen aan de code eenvoudig te maken. Zo is het bijvoorbeeld eenvoudig om het Input blok te vervangen door een blok dat een testsignaal genereert, een gegeven dat zeer handig is tijdens de ontwikkeling. Ook met het oog op een eventuele toekomstige opstelling met acht parallelle ADCs is dit belangrijk, hier zal er immers sowieso een FIFO nodig zijn om de meetpunten van de verschillende ADCs te bufferen.

6.3 Trigger

De eenvoud van de voorgestelde trigger (zie sectie 5.4) doet ook een eenvoudige implementatie vermoeden. De trigger voert het schema dat is weergegeven in Fig. 6.2 uit, dit bevat één enkele loop waarin enkel eenvoudige logische functies ($<, =, ...$) en bewerkingen met kleine counters worden uitgevoerd ($+1$). Dit zijn eenvoudige functies die op de FPGA weinig resources innemen, en vooral zeer snel uitgevoerd zouden moeten kunnen worden. Het is belangrijk dat de snelheid waarmee de trigger werkt hoog is ten opzichte van het fitten en het histogrammen, omdat de trigger de hoogste *data rate* binnenkrijgt. We zijn echter nooit genoodzaakt geweest om de trigger aan te passen omwille van snelheidsproblemen, het ligt in de aard van een FPGA dat logische functies zoals deze op een snelle manier kunnen worden uitgevoerd.



Figuur 6.2: Dit schema geeft de werking van de geïmplementeerde trigger weer. Alles wat zich binnen de stippellijn bevindt zit in één loop. De interne FIFO is een FIFO bestaande uit exact 10 punten, en de transferFIFO is de FIFO die de getriggerde pulsen naar de fit leidt.

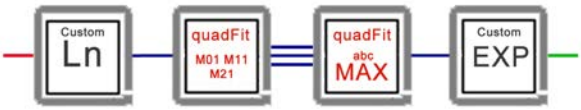
De volledige werking van de trigger kan begrepen worden aan de hand van Fig. 6.2. Het is echter nog belangrijk om te vermelden dat er twee veiligheden zijn ingebouwd: Ten eerste wordt er enkel getriggerd op pulsen die tien punten, of meer, boven het triggerniveau opleveren. Ten tweede zorgt de trigger ervoor dat pulsen die punten bevatten met de saturatiewaarde van de ADC ook niet getriggerd worden. Dit is geen volledig waterdicht systeem om gesatureerde pulsen te vermijden vermits de beperkte *sample rate* van de ADC ervoor kan zorgen dat er geen gesatureerd punt wordt opgemeten. Een goede experimentator dient echter de *preamplifiers* zo af te regelen dat de meerderheid van de gammadeeltjes die opgemeten worden onder de saturatiewaarde liggen. Verder is het nog belangrijk om op te merken dat het triggerniveau *real time* vanaf de host aanstuurbaar is.

6.4 De fit

Het moet ondertussen duidelijk zijn dat de implementatie van de fit de belangrijkste stap van het ontwikkelingsproces is geweest. De trigger stuurt via een FIFO pulsen bestaande uit tien punten naar de Fit. De fit bestaat opnieuw uit vier parallel uitgevoerde bewerkingen die in Fig. 6.3 getoond worden. Eerst wordt van de tien getriggerde punten telkens een logaritme getrokken. Vervolgens worden die 10 logaritmes gebruikt om de eerste stap van de kwadratische fit uit te voeren: het berekenen van de coëfficiënten $M_{01}, M_{11}, en M_{21}$. Deze drie coëfficiënten worden elk via een aparte FIFO doorgestuurd naar het finale deel van het fit algoritme, waar het logaritme van de amplitude wordt bepaald:

$$\ln A = c - \frac{b^2}{4a}.$$

De laatste stap bestaat uit het nemen van de exponent van dit getal, waarna dit via een FIFO naar de histogramming wordt doorgestuurd.

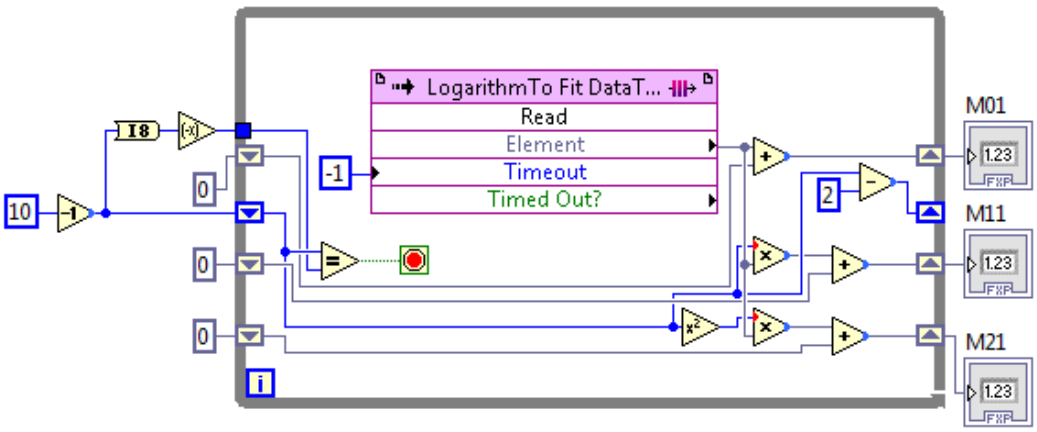


Figuur 6.3: De vier parallel werkende delen van het QuadFitalgoritme zijn: de logaritmische functie, de generatie van de benodigde sommen M_{kl} voor de kwadratische fit, de kwadratische fit, en de exponentiële functie. De gekleurde lijnen duiden de FIFO's aan die de benodigde data doorsturen en bufferen: De rode lijn is de FIFO die de data van de trigger bevat, elke blauwe lijn symboliseert één interne FIFO, en de groene lijn staat voor de FIFO die de amplitudes naar de Histogramming doorstuurt.

De keuze voor parallellisatie van deze stappen is genomen omdat deze stappen *rescource* intensief zijn, en dus ook traag kunnen verlopen. De twee laatste stappen hebben echter een 10x lagere data rate, en zouden dus samen genomen kunnen worden. Omdat een eventuele toevoeging van een methode om de juistheid van de fit te evolueren, en aan de hand van het resultaat hiervan te beslissen of het maximum berekend moeten worden of niet in deze laatste twee stappen terecht zou komen worden ze toch opgesplitst. Deze methodes brengen immers extra complexiteit –en dus benodigde logica– met zich mee.

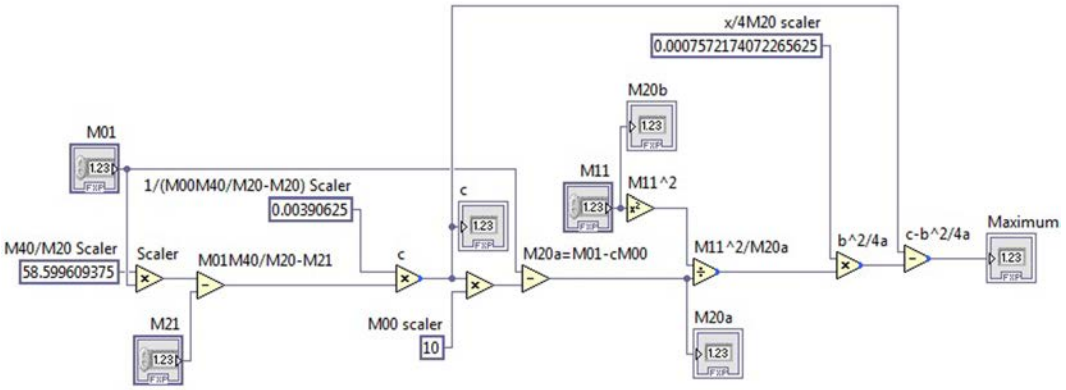
De kwadratische fit is opgesplitst in twee delen die we hier apart overlopen, een gedetailleerde bespreking van de programmatuur is verwerkt in hoofdstuk 7. Het eerste deel bestaat uit de berekening van de benodigde sommen M_{01}, M_{11}, en M_{21}. De berekening hiervan (zie sectie (4.3)) is mathematisch eenvoudig, en laat weinig ruimte voor variatie. De implementatie (zie Fig. 6.4) bestaat dan ook uit een enkele loop waarin de drie sommen parallel worden berekend, de loop loopt over de tien verschillende punten. Voorts springen de x-coördinaten in stappen van twee van -9 tot $+9$ zoals wordt toegelicht in sectie 4.2.1. Voor maximale snelheid is er per som eigen logica voorzien op de FPGA om de berekeningen uit te voeren. Snelheid is hier namelijk belangrijk ten opzichte van het tweede deel van de berekening van de fit omdat de *data rate* hier 10 keer hoger ligt.

Het tweede gedeelte van de kwadratische fit is het gedeelte waarin het logaritme van de amplitude berekend wordt met behulp van de sommen uit het eerste gedeelte. De methode



Figuur 6.4: De LABVIEW code die drie sommen M_{01}, M_{11}, en M_{21} berekent.

van de minste delingen uit sectie 4.2.3 wordt hier op een gewijzigde manier toegepast door het invoeren van scalers voor de sommen die we van te voren kennen, zoals besproken in sectie 4.3.4. De Labview implementatie wordt weergegeven in Fig. 6.5. Hiervoor is ook een soort optimalisatie mogelijk: als een bewerking en een schaling op dezelfde moment uitgevoerd kunnen worden moet er bepaald worden wat de meest efficiënte volgorde is. Omdat in de regel, bij elke bewerking het formaat (de grootte en/of de precisie) van een fixed point zal groeien is het in de regel beter om de schaling zo ver mogelijk naar achter te trekken in de implementatie. Dit is voor alle schalingen uitgevoerd, en is duidelijk te zien in de berekening van de laatste deling. Daar zie je dat er na het uitvoeren van de deling nog geschaald wordt met een factor $1/4M_{20}$. Deze schaling had even goed kunnen gebeuren door M_{20} te schalen met $4M_{20}$, maar dit levert grotere FXP aan de deling ((±,29,13) tegenover (±,39,33)), wat uitmondt in één extra benodigde DSP48E.



Figuur 6.5: De LABVIEW code die het logaritme van de amplitude ($c - \frac{b^2}{4a}$) bepaald. De labels van de verschillende bewerkingen geven de uitkomst van de bewerking weer. In dit diagram zie je ook dat aan dit algoritme ook de (al dan niet geschaalde versies van) coëfficiënten a,b, en c kan gevraagd worden. De aanwezigheid van deze *outputs* in de code vertraagt de code niet zolang deze waarden niet opgevraagd worden omdat de compiler zal beslissen om deze niet te implementeren op de FPGA.

6.5 Histogramming

Het laatste gedeelte van de implementatie is het histogrammen en communiceren van de data. De FPGA is uitgerust met een goede tool om het histogram te realiseren: RAM geheugen dat in één klokcyclus op de FPGA kan worden uitgelezen. Er wordt dus een stuk geheugen op de FPGA gereserveerd dat het histogram representeert. Elke bin wordt vertegenwoordigd door een 32 bits unsigned integer (U32) getal dat onder een bepaald geheugenadres is opgeslagen, en dit getal staat voor het aantal *counts* dat binnen deze bin valt. Er is gekozen voor de U32 representatie opdat ook lange metingen tot de mogelijkheden van de detector zouden behoren. Het systeem aanvaardt op deze manier tot 4 294 967 296 *counts* per bin, terwijl het alternatief U16 tot 65 536 *counts* aanvaardt.

Deze component moet voor de rest geen ingewikkelde berekeningen uitvoeren, en is in die zin te vergelijken met de *trigger*. Het grote verschil met de trigger is echter de veel lagere *data rate*. Waar de trigger aan de *input rate* (voor ons systeem 750 kHz) moet werken moet het histogramming gedeelte enkel aan de *pulse rate* van het signaal[1] werken. Een optimalisatie in de zin van snelheid is dus niet essentieel. We implementeren dan ook een zo simpel mogelijk algoritme om de gebruikte resources te beperken.

Het algoritme bestaat dan ook uit één enkele loop die telkens bekijkt of de amplitude die in een bin gestoken moet worden kleiner is dan de bovengrens van een bin. De loop begint bij de laagste bin te tellen en loopt op, totdat de ongelijkheid voldaan is. Vervolgens wordt de inhoud van deze bin met één verhoogd. Het aantal bins (tot 1024)[2], en de breedte van de bins kan afgeregeld worden door de host, verder is het ook nog mogelijk om alle bins te *clearen*. Het is mogelijk om op snellere manieren de data te verdelen, maar zoals aangehaald is hier geen noodzaak voor. Hier is dan ook geen verdere nadruk op gelegd tijdens de implementatie.

[1] Als we kijken naar de volgende twee hoofdstukken zien we dat we niet hoger dan 10 kHz moeten gaan. Bij deze *pulse rate* is de *pile up* al zo sterk aanwezig dat spectra onleesbaar worden.

[2] Dit kan echter eenvoudigweg verhoogd worden in ons systeem door meer geheugen voor het histogram vrij te houden in de RAM van de FPGA.

Hoofdstuk 7

Details van de implementatie

In dit hoofdstuk worden eerst enkele programmatorische details besproken, en is in eerste instantie interessant voor een lezer die zelf van plan is met een FPGA te werken. Het zwaartepunt ligt hierin in de sectie die de specificatie van de grootte van alle *fixed point* getallen verduidelijkt. De volgende twee secties leggen de noodzaak voor, en de werking van de aangepaste logaritmische en exponentiële functies die geïmplementeerd zijn. Als laatste bespreken we een eerder technisch detail waarmee een programmeur van een FPGA ten allen tijde rekening dient te houden: de communicatie tussen de host en de FPGA.

7.1 Fixed Points

Zoals eerder al veelvuldig vermeld zijn de optimalisaties van de FXP getallen een belangrijke manier om het aantal gebruikte DSP48E componenten te verminderen. Berekeningen met getallen die uit meer bits bestaan zijn voor een FPGA immers meer *resource* intensief om uit te voeren: Voor grotere aantallen bits zijn meer DSP48E componenten per berekening nodig. Omdat LABVIEW en Xilinx over deze DSP48E componenten erg weinig informatie beschikbaar maken lag onze aanpak voor het minimaliseren van de vereiste DSP48E componenten dan ook gewoon in het zoveel mogelijk verminderen van de grootte van de getallen in de module die de fits uitvoert (zie secties 2.3.1, 2.3.3 en 6.1). In de rest van deze sectie wordt de hiervoor gebruikte methodiek toegelicht.

LABVIEW berekend aan de hand van de input van een functie automatisch wat voor formaat de output van de functie moet hebben opdat deze volledig juist weergegeven kan worden (dus groot en precies genoeg). Omdat wij echter in sommige gevallen beter dan de PC weten hoe groot de getallen zijn kunnen we deze waarden verkleinen. We zullen dit illustreren aan de hand van een voorbeeld: we overlopen hoe de som M_{21} wordt berekend. Dit is de onderste berekening in Fig. 6.4, en bestaat uit een kwadraat gevolgd door een vermenigvuldiging en een som.

- Als we eerst kijken naar het kwadraat weten wij iets dat LABVIEW niet weet: LABVIEW neemt aan dat de input van dit kwadraat een *signed integer* met een IWL van 8 is, en dus een maximale waarde van 128 heeft. LABVIEW zal als output dan een *integer* van 16 bits kiezen, omdat dit de kleinste *integer* is waar $128^2 = 16384$ inpast. Wij weten echter dat de maximale waarde van de input van het kwadraat $9^2 = 81$ is[1],

[1]De grootste waarde die onze x coördinaat kan aannemen is immers 9 in ons systeem van 10 punten.

wat past in een integer van 8 bits.

- We nemen deze informatie nu verder mee naar de volgende berekening in de reeks: de vermenigvuldiging. Een input bestaat uit de output van het logaritme, dit is een $(\pm, 18, 3)$ FXP die deze data range volledig gebruikt. Deze FXP loopt dus van -4 tot +4, met een precisie van 3.051175810^{-5}. We rekenen dan de maximale waarde die de vermenigvuldiging zal opleveren uit door $4 * 81 = 324$ te berekenen, dit is groter dan $2^8 = 256$ en kleiner dan $2^9 = 512$, past dus in een *signed* FXP met een IWL van 10. Om de precisie te bepalen moeten we de precisie van ons kwadraat kennen, vermits we over een *integer* praten zal de minimale precisie 1 zijn. De precisie van de FXP die de uitkomst van de vermenigvuldiging is verkrijgen we door middel van de berekening $1 * 3.051175810^{-5}$, wat een FXP van $(\pm, 25, 10)$ oplevert.

- Als laatste berekenen we het formaat dat de som opleveren. De maximale waarde die deze kan bereiken is $(9^2 + 7^2 + 5^2 + 3^2 + 1) * 2 * 4) = 1320$, en de precisie van de som zal de precisie van onze input blijven. Dit levert ons een FXP van $(\pm, 27, 12)$ op.

Deze methode is toegepast voor het hele algoritme, hier is een .vi die deze formaten voor ons berekent geschreven: Format Calc.vi.

Na het uitvoeren van deze methode voor alle getallen hebben we nog een kleine aanpassing aangebracht. Er is namelijk een tweede gegeven dat vastligt: de precisie van de output. De custom EXP functie accepteert namelijk enkel $(\pm, 18, 3)$ FXP getallen. Aan de hand van de precisie van deze getallen kunnen we de precisie (en dus de WL) van de berekeningen ervoor bepalen. Het is immers niet nodig bij een hogere precisie te rekenen dan deze die de output aanvaardt. Op een analoge manier als de aanpak waarbij we kennis hebben van de initiële parameters is de benodigde precisie aangepast.

Ter illustratie vermelden we bijvoorbeeld hoeveel de grootte van de twee FXP die aan de deling die we in de fit uitvoeren deelnemen verminderd zijn via onze methode: Hetgeen LABVIEW voor ons berekend is als input $(,,)$ en $(,,)$. Na onze optimalisatie wordt dit: $(+, 45, 15)$ en $(\pm, 29, 13)$. Verder is het nog belangrijk om op te merken dat bij een aanpassing van het aantal punten deze optimalisatie volledig overgedaan moet worden omdat de input parameters veranderen.

7.2 Custom Ln

Het logaritme dat getrokken moet worden vooraleer we een kwadratische fit kunnen uitvoeren is op een FPGA minder eenvoudig dan het lijkt. LABVIEW FPGA beschikt namelijk wel over een logaritmische functie, maar wel een die enkel $(+, 16, 0)$ FXPs met een waarde tussen $[1/e, 1]$ aanvaardt. Het omvormen van de data die in $(\pm, 16, 16)$ formaat staan naar het $(+, 16, 0)$ formaat is een trivialiteit, dit kan op een zeer efficiënte manier worden uitgevoerd door middel van een *bitshift*. Dit kan je bekijken als een proces waarbij de punt in de FXP gewoon van plaats verschoven wordt. Wat problemen veroorzaakt zijn de grenzen van de aanvaarde getallen $[1/e, 1] = [0.3679, 1]$ die het logaritme aanvaardt. Het $(+, 16, 0)$ FXP getal kan zich immers tussen de grenzen $[0, 1]$ bevinden. We verliezen op deze manier dus 36.79% van ons meetinterval, wat betekent dat we geen meetpunten onder 3.679 V zouden kunnen accepteren. Dit is natuurlijk een onwerkbare situatie, dus hier is een oplossing voor gezocht.

We zouden onze data zo kunnen schalen dat alles in het interval $[1/e, 1]$ zou passen, maar een schaling zou een verlies aan resolutie betekenen. Onze oplossing behoudt de resolutie, en maakt gebruik van eenvoudige wiskunde:

$$\ln(x) = \ln(2x) - \ln(2).$$

We kunnen dus om het logaritme van een getal onder 0.3679 te berekenen dit getal eerst met een factor twee vermenigvuldigen, dan het logaritme hiervan trekken, en vervolgens het logaritme van twee hiervan aftrekken. Omdat een factor twee niet altijd volstaat kunnen we deze procedure in een loop plaatsen, en zo de vermenigvuldiging herhalen tot deze in het interval past. De logaritmes van twee, en de hogere machten van twee kunnen van te voren uitgerekend worden, en op een geheugen in de FPGA geplaatst worden. Er moet een maximaal aantal keren vastgelegd worden waarop de vermenigvuldiging herhaald mag worden om de snelheid van het systeem hoog genoeg te houden. Wij hebben hier geen studie van gedaan, maar gewoon een keuze gemaakt voor een maximum van drie iteraties, dit komt overeen met een minimale inputwaarde van 0.4599 V. Het triggerniveau zal dus altijd boven deze waarde moeten liggen, wat in ons geval geen probleem oplevert omdat dit in de buurt lijkt te komen van het ruisniveau (zie hoofdstuk 3). Indien lagere triggers toch vereist zijn kan dit aangepast worden.

De implementatie op de FPGA is zoals deze hierboven beschreven wordt, maar achteraf moet er toch opgemerkt worden dat dit zeker niet de efficiëntste manier is. Het aanpassen van de schalingsfactor van 2 naar $e = 2.719$ zal de minimale waarde van de trigger voor eenzelfde aantal loops verlagen.

7.3 Custom EXP

De exponentiële functie die na de kwadratische fit moet worden uitgevoerd bevat eenzelfde soort van probleem als het logaritme. De factor waar we een exponent van willen nemen bevindt zich tussen $[-4, 0]$, en de exponentiële functie accepteert enkel getallen binnen $[-1, 1]$. Gebruik makend van $e^x = e^{2x}e^{-2}$ kunnen we een analoge oplossing bedenken. Als ons getal zich buiten het bereik $[-1, 1]$ bevindt kunnen we hier één of twee keren het getal twee bij optellen of aftrekken, zodat het zich wel binnen het interval bevindt. Vervolgens kan de exponent genomen worden, en via een vermenigvuldiging met de waarde gecorrigeerd worden. De implementatie hiervan werkt –net als het custom logaritme– met een geheugen dat de correctiefactoren bevat.

7.4 Communicatie

De communicatie van een FPGA met de host(PC) is iets dat omzichtig aangepakt moet worden. De moeilijkheid is het feit dat een PC, die serieel data aanvaard en verwerkt, niet telkens op het moment dat een FPGA ergens data aanbiedt dit kan uitlezen, er is immers geen mechanisme dat de PC en het programma dat op de FPGA loopt synchroniseert. Andersom, in de situatie waar de host data naar de FPGA wil sturen, geldt hetzelfde probleem. Een eenvoudige oplossing voor dit probleem is het gebruik van een buffer; de FPGA is dan ook voorzien van twee speciale buffers die dit verkeer kunnen regelen.

Het gebruik van deze buffers is echter niet altijd eenvoudig omdat er maar 2 van deze buffers aanwezig zijn. Wat betekent dat we, als we meer dan twee kanalen hebben waarin belangrijke informatie gecommuniceerd wordt er telkens een custom interleaving algoritme geschreven wordt dat twee –of meerdere– soorten data door dezelfde buffer communiceert. Dit legt dan weer extra eisen op aan de parallel lopende code in termen van timing en synchronisatie. Kortom, een scenario dat we indien mogelijk best vermijden.

In onze finale implementaties gebruiken we dan ook geen buffers bij de communicatie tussen FPGA en host, *data rates* en volumes zijn zo beperkt dat dit niet noodzakelijk is. Voor het aansturen van de FPGA[2] wordt gewoon rechtstreeks een signaal gestuurd naar de FPGA. Deze signalen *setten* een variabele op een FPGA, die deze waarde zal behouden tot op het moment waarop een nieuw signaal naar de FPGA gestuurd wordt. Andersom wordt de inhoud van één enkele bin ook rechtstreeks gecommuniceerd naar de PC. Het is echter wel belangrijk om deze buffers te vermelden, omdat het tijdens het ontwikkelingsproces vaak nodig is gebleken om te *proben* wat er precies in verschillende facetten van het programma precies gebeurt, en omdat de *data rates* hier vaak zeer hoog liggen, moet hier dan ook gebruik gemaakt worden van deze buffers.

[2]De aansturing bestaat uit 5 zaken: van het triggerniveau, het aantal bins, en de grootte van de bins, de bin die uitgelezen dient te worden, en een optie om de bin te *clearen*.

Hoofdstuk 8

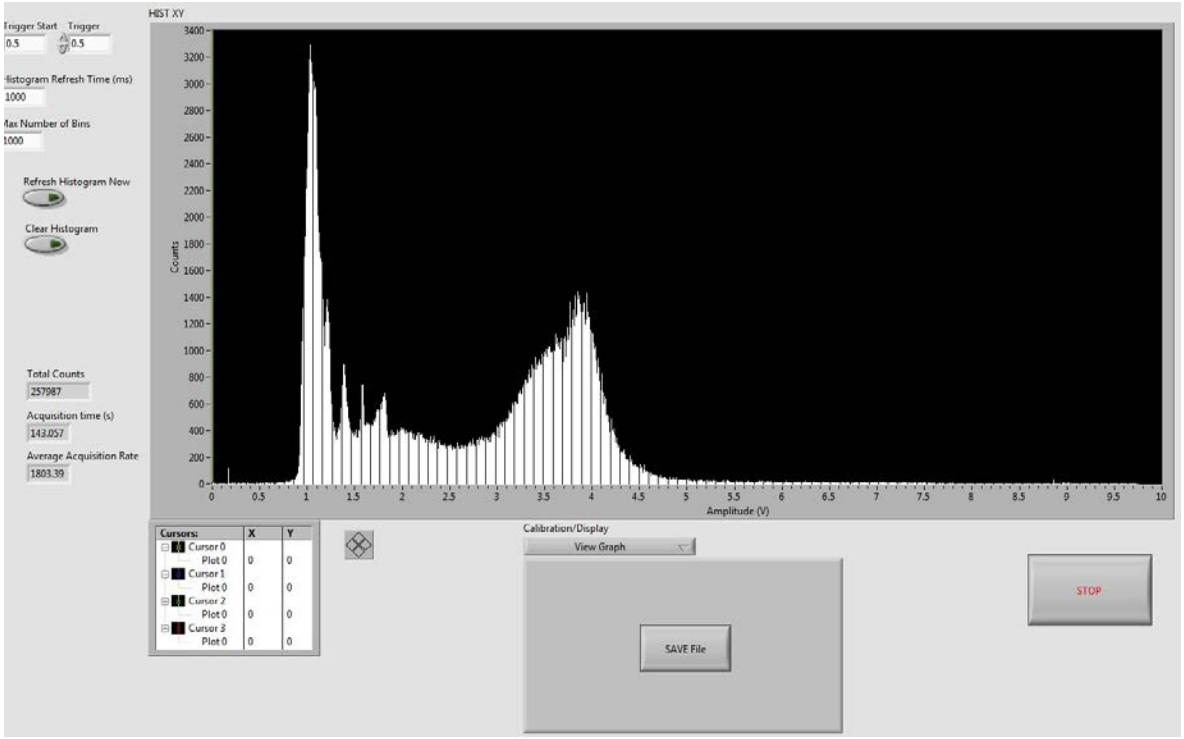
Implementatie op Host

Om een complete gammaspectroscopieopstelling te maken hebben we naast de detector, de elektronica en de FPGA die als *multi channel analyser* werkt nog een laatste sluitstuk nodig: een systeem dat de FPGA aanstuurt en het spectrum grafisch weergeeft. Dit kan een zeer eenvoudig systeem zijn vermits de FPGA onafhankelijk van de PC werkt. De enige zaken die aangestuurd moeten worden zijn: triggerniveau, maximaal aantal bins, breedte van een bin, en een clear functie van het interval. Er kan verder nog aan de FPGA gevraagd worden om het aantal counts in een bepaalde bin door te sturen. Deze taken kunnen gerealiseerd worden met een display en een beperkte hoeveelheid eenvoudige elektronica, iets wat interessant is als je naar een praktische detector toewerkt. Wij hebben echter een PC ter beschikking, en hebben zodoende op de PC in LABVIEW een gebruiksinterface geschreven om de spectroscopie uit te voeren. Wanneer er een mobiel apparaat ontwikkeld moet worden kunnen deze taken door een eenvoudige processor worden overgenomen .

In deze toepassing laat LABVIEW zijn sterke kanten zien: het is relatief eenvoudig om een duidelijke en intuïtieve gebruikersinterface te programmeren. Omdat het zwaartepunt van dit proefschrift op het programmeren van de FPGA ligt gaan we niet teveel uitweiden over deze component. Deze is vooral geïmplementeerd om zelf proeven uit te voeren om te beoordelen hoe goed ons volledige systeem werkt, en in mindere mate om aan te tonen dat we over een volledige spectroscopietoepassing beschikken.

We overlopen dan ook gewoon de functies die uitgevoerd kunnen worden met behulp van ons programma zoals in Fig. 8.1 weergegeven:

- Het programma kan spectra uitlezen en weergeven aan een gewenst verversingstempo.
- Het programma kan het opgeslagen spectrum op de FPGA verwijderen en resetten.
- Het programma laat de gebruiker toe het triggerniveau in te stellen.
- Het programma kan het totaal aantal opgenomen counts, de duur van de acquisitie, en de gemiddelde acquisitie snelheid weergeven.
- Er kan met behulp van cursoren data aangeduid en afgelezen worden van het spectrum.
- Er kan een gaussische fit uitgevoerd worden aan pieken uit het spectrum die je selecteert door middel van cursoren.



Figuur 8.1: Een opname van de gebruikersinterface die op de hostPC draait. We zien een opname van een Ba133 bron.

- Om de achtergrond ten gevolge van Compton verstrooiing uit het gemeten spectrum te verwijderen kan er ook een meer geavanceerde fit uitgevoerd worden, waarbij er eerst een polynoom aan een geselecteerd gebied rond de puls wordt gefit, om vervolgens een gaussische fit uit te voeren van een piek die gecorrigeerd is met het resultaat van de polynomiale fit.
- Het spectrum kan manueel gecalibeerd worden.
- Het spectrum kan gecalibreerd worden door twee pieken op het spectrum aan te duiden en een energie te geven. Dit is een lineaire schalingsmethode.
- Het spectrum kan gecalibreerd worden door drie pieken op het spectrum aan te duiden en een energie te geven. Dit is een kwadratische schalingsmethode.
- In beide vorige methodes kunnen de kalibratiepieken ook door middel van de gaussische en meer geavanceerde fit bepaald worden.
- Als laatste kan een spectrum ook opgeslagen worden in een textfile. Alle bins en counts worden hierin genoteerd, alsook informatie zoals de start en eindtijd van de acquisitie, het totaal aantal opgenomen counts, de duur van de acquisitie, en de gemiddelde acquisitie snelheid.

Het is misschien interessant om nog eens op te merken dat de FPGA onafhankelijk van dit programma zal werken. Dus terwijl de PC bezig is met berekeningen zoals een calibratie blijft de FPGA data nemen.

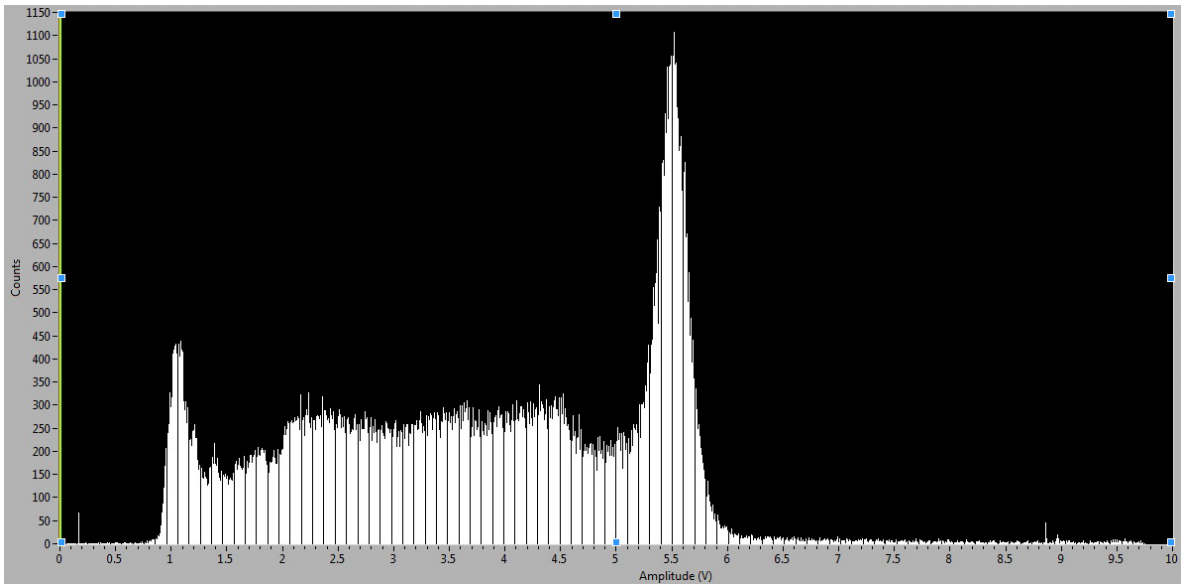
Hoofdstuk 9

Resultaten

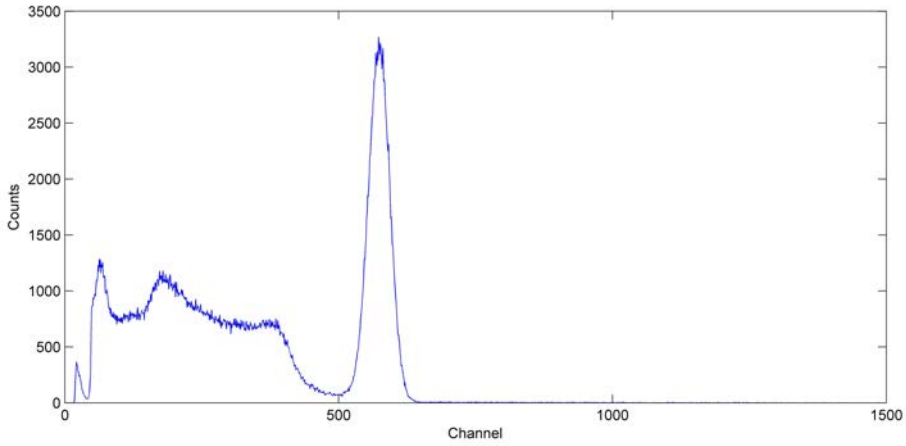
In dit hoofdstuk gebruiken we onze gammaspectroscopieopstelling om radioactieve bronnen op te meten. Aan de hand van de resulterende spectra in Fig. 9.1 en Fig. 9.3 kunnen we zaken zoals de lineariteit en resolutie van de detector becommentariëren. Verder bespreken we kort de performantie van het systeem op de FPGA, we bespreken zowel de benodigde resources op de FPGA als de snelheid van ons algoritme.

9.1 De referentieopstelling

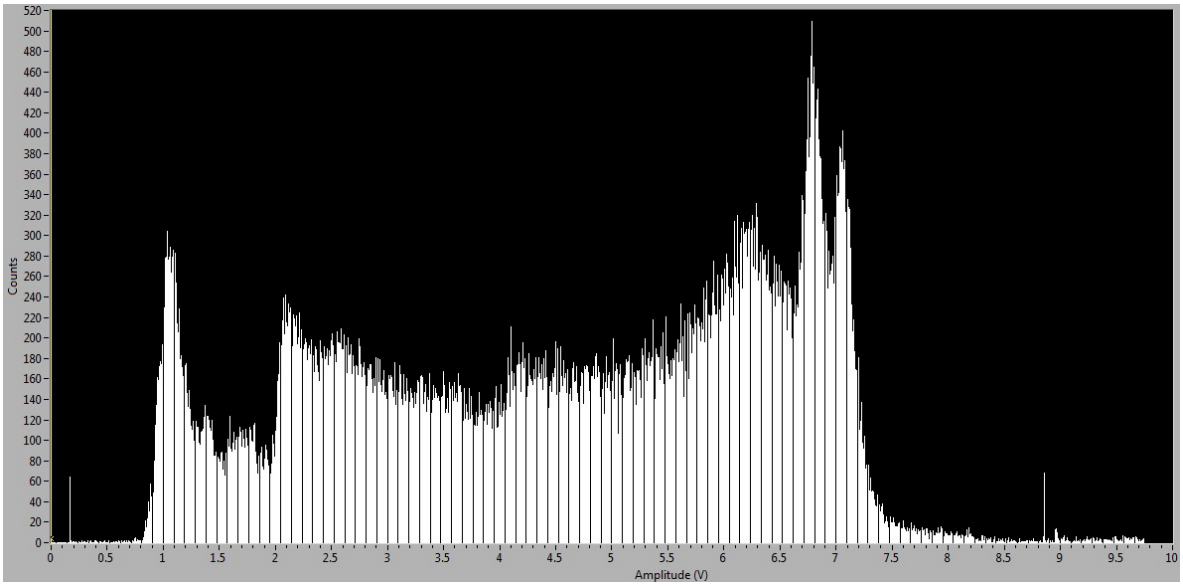
Vooraleer we de resolutie en de lineariteit van onze resultaten kunnen bespreken moeten we een referentiekader vastleggen. De scintillator en de PMT die we in onze opstelling gebruikt hebben behoren eigenlijk tot een gammaspectroscopiepakket van Spectrum Techniques [13]. Naast deze twee onderdelen bestaat dit pakket uit een hoogspanningsbron, een multi channel analyser (inclusief de benodigde elektronica om de signalen te verwerken), en een computerprogramma waarmee het geheel aangestuurd, uitgelezen, en verwerkt kan worden. De interne werking van de MCA is jammer genoeg informatie die Spectrum Techniques niet vrijgeeft. Het belangrijkste is echter dat het geheel dezelfde functies kan uitvoeren als onze detector (inclusief de toepassing op de Host PC), en een uitgelezen referentieopstelling is omdat het gebruik van dezelfde scintillatiedetector eventuele verschillen op dat niveau elimineert. Opnames van twee bronnen zijn te zien in Fig. 9.2 en Fig. 9.4.



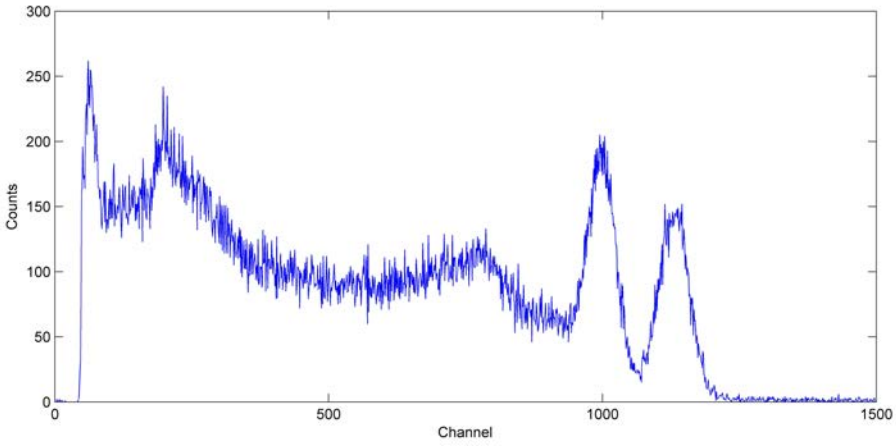
Figuur 9.1: Een spectrum opgenomen aan de hand van een 30 kBq Cs137 bron. De acquisitie tijd is 100 s, met een acquisitie rate van 1461.5 /s. Als we vergelijken met het referentiespectrum in Fig. 9.2 zouden we kunnen zeggen dat we rond 5.5 V de karakteristieke gammapiek van 662 keV waarnemen. Verder vinden we ook de *backscatter* terug bij 2.1 V en de *Compton edge* bij 4.5 V.



Figuur 9.2: Een referentiespectrum van een 30 kBq Cs137 bron. Gemeten met dezelfde scintillator en PMT, maar andere elektronica. De twee pulsen in de laagst energetische kant zijn x straling, een gegeven dat door deze detector niet optimaal gemeten wordt. Daarnaast zien we van laag naar hoog: een uitgesproken *backscatter* piek, de *Compton edge*, en de karakteristieke gammapiek van 661 keV.



Figuur 9.3: Een spectrum opgenomen aan de hand van een 12 kBq Co60 bron. De acquisitie tijd is 184 s, met een acquisitie rate van 672.7 /s.Als we vergelijken met het referentiespectrum in Fig. 9.4 zouden we kunnen zeggen dat we rond 7 V de twee karakteristieke gammapieken van 1.17 MeV en 1.33 MeV waarnemen. Verder vinden we ook de *backscatter* terug bij 2.1 V en de *Compton edge* bij 6.2 V.



Figuur 9.4: Een referentiespectrum van een 12 kBq Co60 bron. Gemeten met dezelfde scintillator en PMT, maar andere elektronica. De laagst energetische puls is x straling, een gegeven dat door deze detector niet optimaal gemeten wordt. Daarnaast zien we van laag naar hoog: een uitgesproken *backscatter* piek, de *Compton edge*, en de karakteristieke gammapieken van 1170 keV en 1330 keV.

9.2 Resolutie

De NaI scintillatiedetector verbreedt de energie van een invallend deeltje volgens een gauss-verdeling. De breedte van deze gauss is een maat voor de resolutie van de detector. De meest gebruikte maat voor resolutie in de spectroscopie is dan ook de fractionele energieresolutie die in % wordt uitgedrukt [14]. Dit is niet anders dan een relatieve maat voor de breedte van een piek (ΔE):

$$\frac{\Delta E}{E}.$$

Voor de breedte van de piek (ΔE) wordt meestal de *Full Widt at Half Maximum* (FWHM) van de curve genomen. Een makkelijk te meten eenheid die recht evenredig is met de σ van de gauss. We hebben dan ook de breedtes van verschillende pieken opgemeten in ons eigen systeem en het referentiesysteem, met als resultaat Tabel 9.1. We kunnen hier zien dat ons systeem op het vlak van resolutie toch nog slechter presteert dan het commerciële systeem.

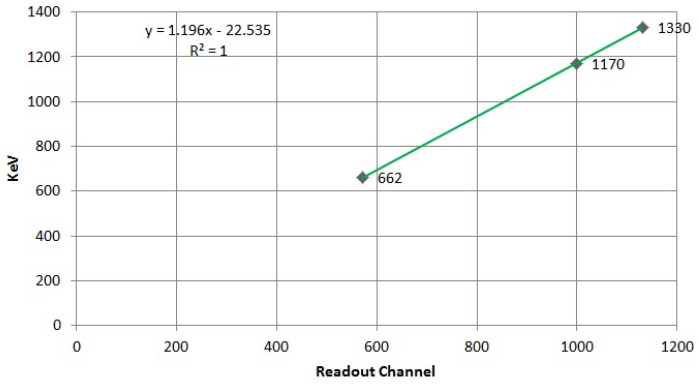
Bron	Piek (keV)	Kanaal Ref.	Resolutie Ref.(%)	Piek positie (V)	Resolutie (%)
Cs137	662	572	7.69	5.52	14.3
Ba133	356	geen data	geen data	3.75	16.7
Co60	1170	998	5.21	6.79	14.1
Co60	1330	1130	4.78	7.05	16.1

Tabel 9.1: Een tabel die literatuurwaarden van pieken gemeten met een NaI scintillatiedetector van verschillende bronnen naast de waarden die we gemeten hebben in ons referentiesysteem (Ref.) (Fig. 9.2 en 9.4), en ons eigen systeem (Fig. 9.1 en 9.3). Ook de resoluties bepaald met behulp van de FWHM zijn toegevoegd.

Als we de resolutie echter vergelijken met de resolutie van $\approx 2\%$ die we verkregen uit onze studie van de witte ruis door middel van MATLAB simulaties (zie sectie 5.2.2) zien we dat de (slechtere) resolutie niet het gevolg kan zijn van de witte ruis op het signaal. Dit doet vermoeden dat het systeem van de *shaper* en *preamplifier* niet optimaal presteert.

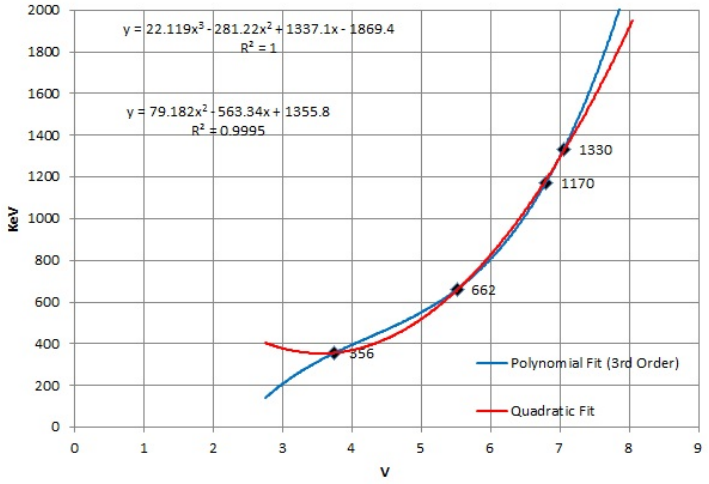
9.3 lineariteit

Om de lineariteit van onze detector te bepalen nemen we opnieuw waarden uit Tabel 9.1. Door deze waarden uit te zetten ten opzichte van de literatuurwaarden van de pieken moeten we de detector kunnen kalibreren. Omdat NaI detectors een zeer goede lineariteit bezitten verwachten we dat deze kalibratie louter met een lineaire functie moet gebeuren. Voor onze referentiedetector is dit correct, zoals in Fig. 9.5 te zien is.



Figuur 9.5: Een afbeelding die de lineariteit van de referentiedetector weergeeft aan de hand van de pulsen uit tabel 9.1.

Als we dezelfde test uitvoeren voor onze eigen detector krijgen we echter Fig. 9.6. Het is hier duidelijk dat een lineair verband niet volstaat; en daarom zijn twee andere fits uitgevoerd: een 2e en 3e graadspolynoom. De redenen hiervoor zullen we moeten zoeken in het systeem van de *charge sensitive amplifier* en de *shaping amplifier*. Het lijkt immers erg onwaarschijnlijk dat de fit hiervoor verantwoordelijk is, we hebben in sectie 5.1.2 al eerder opgemerkt dat de amplitudes in de simulaties goed overeen lijken te komen met de amplitude die we visueel kunnen afschatten van de pulsen.



Figuur 9.6: Twee fits van de lineariteit van onze detector; aan de hand van de pulsen uit tabel 9.1.

Dit probleem lijkt dus –net zoals het geval was bij de resolutie– bij de *shaper* en *preamplifier* te liggen, één van de startpunten van dit proefschrift. Het is dus ook niet de bedoeling om deze elektronica aan te passen. Een mogelijkheid om dit op te lossen is het gebruik maken van de applicatie op de Host PC die het schalen met polynomen toelaat. Hoewel deze polynomen in Fig. 9.6 niet volledig soelaas lijken te brengen kan dit wel een werkbare oplossing bieden. Een groot ongemak is natuurlijk het feit dat je voor het calibreren met hogere graad polynomen ook meerdere pieken nodig hebt, en een simpele calibratie met bijvoorbeeld een enkele Cs137 bron zoals in een lineair systeem onmogelijk wordt.

9.4 Resource usage

Fig. 9.7 geeft de finale voetafdruk van ons algoritme gecompileerd op de FPGA. Er wordt dus maar ongeveer 42 % van de logica op de FPGA gebruikt. We zien hier ook weer dat het aantal DSP48E *multipliers* het gegeven is dat het hoogste oploopt. Toch hebben we dit aantal door middel van alle keuzes en optimalisaties kunnen beperken tot de helft van het beschikbare aantal. Als laatste kan je zien dat er één ram blok gebruikt wordt. Dit is het RAM blok waar het histogram in wordt opgeslagen.

Device Utilization	Used	Total	Percent
Total Slices	2002	4800	41.7
Slice Registers	4035	19200	21.0
Slice LUTs	4944	19200	25.8
DSP48s	16	32	50.0
Block RAMs	1	32	3.1

Figuur 9.7: De voetafdruk van het volledige systeem op de FPGA.

Als conclusie kunnen we zeggen dat het misschien mogelijk geweest was om het algoritme op een kleinere FPGA te implementeren. Ik ben het hier echter niet mee eens. Voor de ontwikkeling van een systeem op een FPGA is het niet slecht om wat teveel logica te beschikking te hebben. Je hebt namelijk logica nodig om zaken te kunnen *proben* op je FPGA, en sommige optimalisaties voer je ook alleen maar uit door een werkend systeem lichtjes aan te passen. Na deze ontwikkelingsstap is het wel mogelijk om een kleinere FPGA te kiezen voor de uiteindelijke implementatie.

9.5 Timing

Tot slot bekijken we de verwerkingssnelheden die we met ons systeem halen. Omdat dit bestaat uit parallel werkende stukken zal de traagste parallelle stap de verwerkingssnelheid van ons systeem bepalen. Vermits we in onze testopstelling enkel over één 750 kHz ADC beschikken moeten we voor onze snelheidstesten een andere manier gebruiken om ze uit te voeren. Een goede methode hiervoor is door de FPGA zelf de data van de inputpuls te laten simuleren aan een rate die veel hoger is dan deze van de ADC. Dit heeft nog een voordeel: op deze manier kunnen we voor elke module apart de input simuleren, en dus de snelheid meten onafhankelijk van de andere modules.

Blok	Snelheid 1 input punt (klokpulsen)	Snelheid 1 puls (klokpulsen)
Trigger	10/15	202-218
Custom Ln	27-39	270-390
Quadfit $M_{01}M_{11}M21$	-	42
Quadfit abc MAX	-	53
Custom EXP	27-39	27-39
Histogram	10	10

Tabel 9.2: De timing van de verschillende parallel werkende blokken op de FPGA is hier uitgedrukt in het aantal klokpulsen dat de blokken nodig hebben om een bewerking uit te voeren. Zowel de timing voor één enkele input voor elke parallel werkend blok als voor een volledige puls zijn weergegeven. De meest interessante waarde is deze die de verwerkingstijd voor één volledige puls weergeeft, omdat we hiermee de verwerkingssnelheid van ons systeem kunnen afschatten.

Er is dus voor elk parallelle blok een implementatie op de FPGA gecompileerd waarbij de FPGA zelf de input van enkele verschillende pulsen simuleert voor dat specifieke blok. De input wordt gegenereerd in een parallel werkend blok dat de data zeer snel simuleert –deze snelheid is afhankelijk van de aard van de input– maar er is altijd nagekeken of dit blok sneller werkt dan de snelheid van het blok dat we opmeten. In Tabel 9.2 zien we de resultaten van deze metingen uitgedrukt in het aantal klokpulsen dat elk blok nodig heeft. Een klokpuls op onze FPGA is standaard 40 MHz, maar kan voor onze implementatie indien nodig opgevoerd worden tot 160 MHz.

Vooraleer we de individuele blokken kort bespreken moet er wel nog een kanttekening bij gemaakt worden: door elke stap apart op de FPGA te implementeren zal de logica niet op dezelfde manier geconnecteerd zijn over de FPGA. Het is mogelijk dat doordat er een complexere *routing* nodig is in een implementatie van het volledige systeem dat een blok één klokpuls trager werkt.

Nu overlopen we de timing van elk blok even:

- Het triggerblok verwerkt punten aan twee verschillende snelheden: 10 of 15 klokpulsen. De lagere snelheid heeft te maken met het feit dat het blok iets langer duurt voor punten na het maximum dan de punten voor het maximum. Achter het maximum moet namelijk een element uit een FIFO gehaald worden, en dit duurt 5 klokpulsen. De 10 klokpulsen die de trigger er sowieso over doet zijn ook het gevolg van FIFO's: in dit blok moet namelijk een opeenvolgend een FIFO uitgelezen, en een andere FIFO terug beschreven worden. Verder is het nog interessant op te merken dat voor punten die niet boven het triggerniveau komen er een verwerkingstijd van 7 klokpulsen nodig is.

- Het Custom Ln blok is de bottleneck van ons systeem. De verwerkingssnelheid varieert naargelang het aantal keer dat het algoritme een loop uitvoert (zie sectie 7.2). Voor kleinere inputwaarden zal dit langer duren, per triggerpunt doet dit blok er 27 tot 39 klokpulsen over, en per puls moet dit dus tien keer uitgevoerd worden.

- Quadfit $M_{01}M_{11}M21$, het tweede gedeelte van het fit algoritme heeft 42 klokpulsen nodig per puls. Dit blok kan moeilijk verder geoptimaliseerd worden, maar loopt al snel

ten opzichte van de rest. Dit komt omdat de tijdrovende componenten de wiskundige bewerkingen zijn, die we naar ons inziens niet verder kunnen optimaliseren.

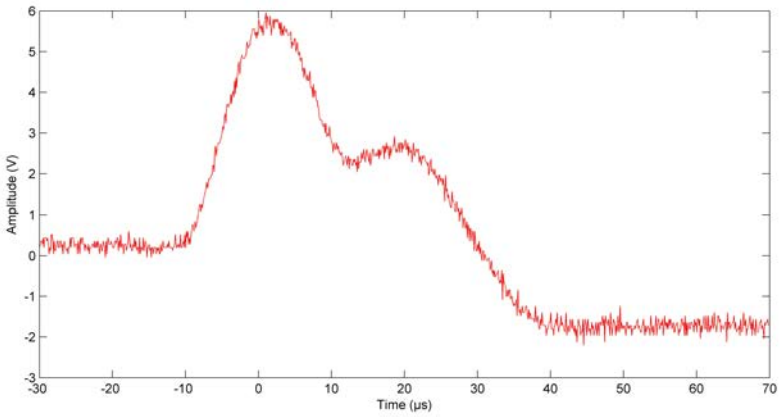
- Het volgende blok van de fit is Quadfit Max, en heeft 53 klokpulsen per fit nodig. Ook hier geldt de opmerking dat dit blok vooral bestaat uit wiskundige bewerkingen waar wij niet direct verdere optimalisatiemogelijkheden voor zien.
- Het Custom EXP blok heeft niet toevallig precies dezelfde tijd nodig voor één punt als het blok dat het logaritme uitvoert. Het is immers een zeer gelijkaardige bewerking. Het is echter geen bottleneck omdat de *data rate* hier al tien maal lager ligt.
- Tot slot het histogram: het schrijven van één amplitude naar het histogram duurt 10 klokpulsen. De tijdrovende processen zijn hier wederom de twee geheugencomponenten: het duurt 5 klokpulsen om de FIFO die de data levert uit te lezen, en 4 klokpulsen om het geheugen dat het histogram bevat te schrijven. De eenvoudige logica die de amplitude in de juiste bin plaatst doet er maar één klokpuls over, en een ingewikkeldere implementatie is zoals aangehaald in sectie 6.5 overbodig.

We kunnen dus concluderen dat de Custom Ln de bottleneck van ons systeem is. Deze module verder optimaliseren, en eventueel dupliceren, kan de snelheid van het systeem opvoeren. Als we dit wegwerken komt de trigger als volgende kandidaat voor optimalisatie in aanmerking. De enige manier om de snelheid hier op te drijven is door de FIFO's op een snellere manier te implementeren. In onze implementatie is deze geïmplementeerd in het block RAM; maar er zijn snellere implementaties mogelijk die in de logica van de FPGA zelf geïmplementeerd worden.

Hoofdstuk 10

Pile Up rejection

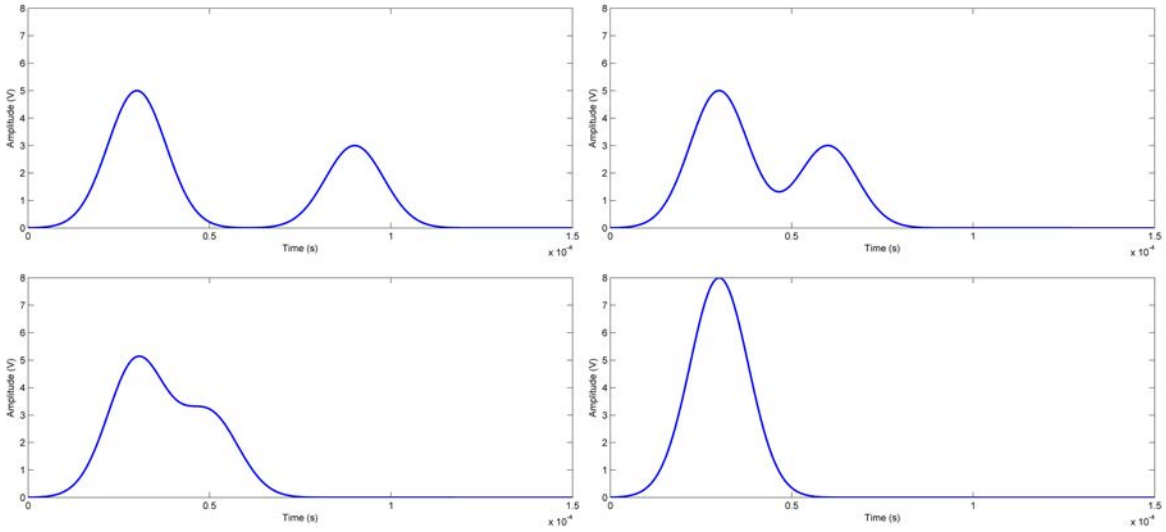
In dit laatste hoofdstuk presenteren we een uitbreiding op het onderzoek dat voor dit proefschrift was voorgesteld. Omdat de meeste commerciële gammaspectroscopen ook over een systeem beschikken om *pile-up* om te gaan hebben we een poging gedaan om een eigen versie hiervan op de FPGA te implementeren. Deze studie moet vooral als een aanvulling op de rest van dit proefschrift beschouwd, en is zodoende ook niet in alle detail uitgewerkt.



Figuur 10.1: Een opname van een geval van *pile up* in onze opstelling bij een *shaping time* van 8 µs.

10.1 Pile Up

Vermits radioactief verval een stochastisch proces is kan het gebeuren dat twee gamma's kort na elkaar op de detector invallen en gedetecteerd worden. In dit geval zal het signaal –nadat *shaping* heeft plaatsgevonden– bestaan uit een som van twee individuele pulsen. Fig. 10.2 illustreert hoe, als deze pulsen dicht genoeg bij elkaar komen te liggen, in eerste instantie de vorm van beide pulsen zal veranderen omdat bij het centrale gedeelte van de pulsen de staart van de andere puls wordt opgeteld. Als de pulsen vervolgens nog dichter bij elkaar komen te liggen zal de vorm sterker aangepast worden, en als twee deeltjes perfect gelijktijdig invallen krijgen we opnieuw een gaussische vorm, maar nu bestaande uit twee opgetelde gausscurves. Omdat we de energieën van aparte vervalkanalen willen meten om bronnen te identificeren is dit een ongewenst effect. Dit komt in ons systeem natuurlijk ook voor, Fig. 10.1 is hier een voorbeeld van.



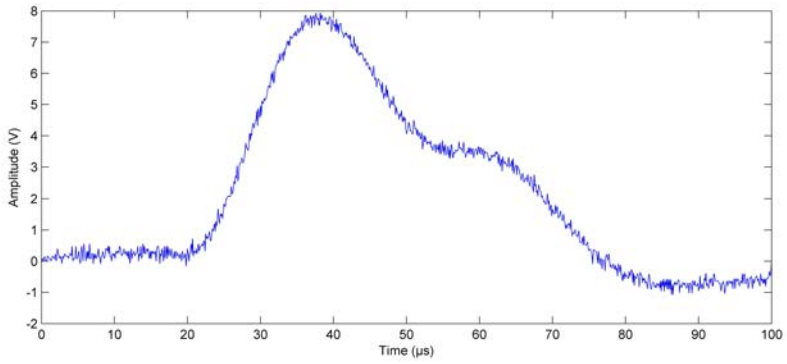
Figuur 10.2: In deze figuur zien we een simulatie van de situatie waarbij twee gausscurves elkaar dicht naderen. Dit komt in de gammaspectroscopie courant voor, en wordt *pile up* genoemd. Het centrum van de meest linkse puls ligt altijd op 3.8 µs, en de pulsen hebben een σ van 8 µs.

Dit effect manifesteert zich logischerwijs vaker bij hoge *count rates*. Vermits een van de eisen van onze detector een hoge snelheid is moeten we een manier om deze *pile up* te behandelen invoeren. Er zijn nu twee methoden om dit op te lossen: ofwel implementeren we een fit die de invloeden van pile-up registreert en corrigeert, ofwel gooien we de pulsen die onderhevig zijn aan *pile up* gewoon weg (*pile up rejection*). De eerste methode zou echter moeten bestaan uit een fit van twee gaussen, en zoals in dit proefschrift al duidelijk is gebleken is het niet eenvoudig om complexe algoritmes op de FPGA te implementeren. We kiezen dus in eerste instantie voor de tweede methode omdat deze eenvoudig is. *Pile up rejection* is geen slechte aanpak omdat in situaties waar we veel *pile up* hebben, we ook over een hoge *count rate* spreken. En in deze gevallen zou de acquisitie zo snel moeten verlopen dat het niet erg is om een paar gemeten pulsen te verliezen.

10.2 Simulatie

Ook in dit hoofdstuk ontkomen we niet aan de ontwikkelingsfase waarin simulaties noodzakelijk zijn. Vooraleer we aan een oplossing beginnen denken moet er eerst eens goed bekeken worden hoe dicht pulsen bij elkaar moeten liggen om een verandering in de amplitude te veroorzaken die boven de standaarddeviatie ligt. We bekijken dit door ons in MATLAB gekloonde systeem de eerste puls van een signaal van twee opeenvolgende pulsen te laten fitten. Als we nu bekijken wat er gebeurt met de gefitte amplitude als we deze twee pulsen (in de tijd) steeds dichter bij elkaar brengen kunnen we zien vanaf wanneer de *pile up* een noemenswaardige invloed heeft. Het is echter onmogelijk om dit uit te voeren met twee opgenomen pulsen, omdat deze ruis bevatten. Bij een optellen van twee pulsen zal deze ruis namelijk ook opgeteld worden, wat de puls sterk beïnvloedt. We zijn dan ook uitgegaan van een model waarbij we bij één testpuls een ruisloze gesimuleerde gauss optellen, zoals in Fig. 10.3. Het nadeel van deze methode is dat deze tweede puls niet de lichtjes aangepaste gauss zal zijn die onze *shaping amplifier* levert (zie sectie 3.2.3), en dat deze simulatie ook de sterke daling van de *baseline* die kan voorkomen bij twee opeenvolgende pulsen (zie Fig. 10.1) niet in rekening brengt.

In ons gesimuleerde systeem kunnen we ook weer dezelfde statistiek invoeren als in hoofdstuk 5. Door gebruik te maken van opnames van onze pulsen, die met een oscilloscoop met een hogere *sample rate* dan onze ADCs opgenomen zijn, kunnen we dezelfde puls meerdere malen door ons algoritme onder handen laten nemen, door andere sets punten te gebruiken.

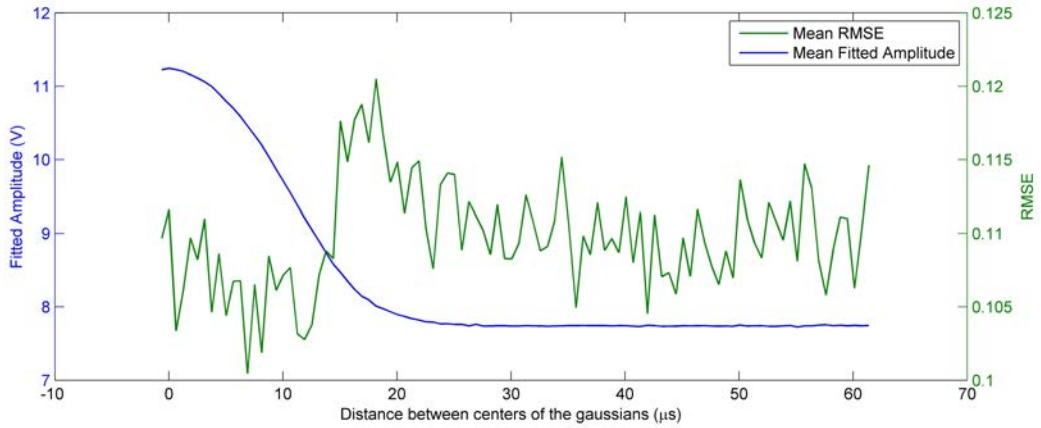


Figuur 10.3: Deze figuur bestaat uit de puls weergegeven in Fig. 3.6 en een gesimuleerde ruisloze gauss die hierbij opgeteld is. De gesimuleerde gauss heeft een σ van 8 µs, een amplitude van 3.5 V, en het centrum (parameter μ) komt 25 µs na de het centrum van de opgenomen puls.

Fig. 10.4 en 10.5 zijn voorbeelden van deze simulaties. Voorlopig kijken we enkel naar de blauwe lijn. Hieraan kunnen we zien wat dat pas vanaf een tijdsverschil kleiner dan $20 - 25$ µs de amplitude van de eerste piek echt zal verschuiven onder invloed van *pile up*. Deze grens komt overeen met het beeld in Fig. 10.3. Met het oog op deze afbeelding kunnen we alvast een conclusie trekken: het wordt erg moeilijk om in het triggerblok *pile up* te registreren en eventueel te weren. Dit omdat de pulsen in de situatie waar *pile up* invloed heeft op onze resultaten de pulsen al zo dicht bij elkaar liggen dat het voor een trigger bijna onmogelijk wordt om dit te detecteren. Een andere aanpak is dus aangewezen.

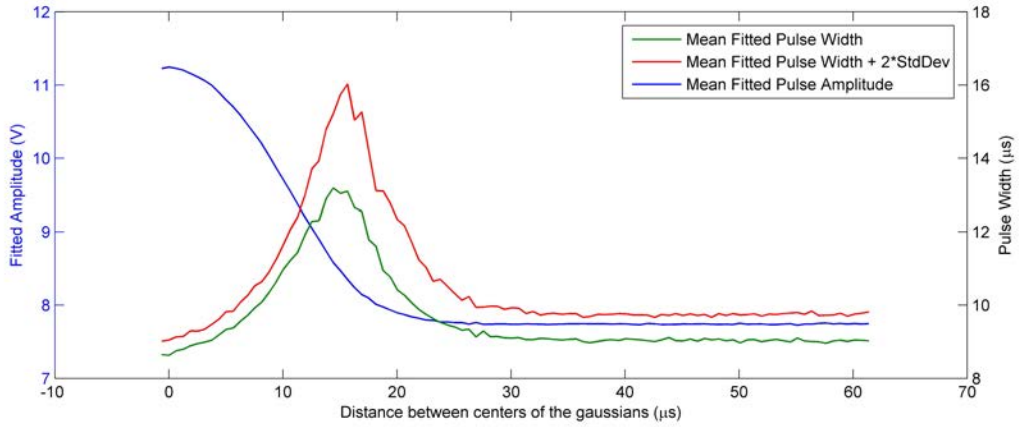
De volgende benadering ligt voor de hand: laat ons kijken of we geen gebruik kunnen maken van de informatie die de fits van de pulsen ons kunnen leveren om te oordelen of deze gefitte pulsen al dan niet beïnvloed zijn door *pile up*. Om dit mogelijk te maken moeten we op zoek naar een parameter die een correlatie vertoont met de invloed van *pile up*. Hiervoor komen in ons algoritme (zie hoofdstuk 4) twee parameters in aanmerking:

De RMSE van de fit: Onder invloed van *pile up* wijkt de curve af van een gauss, daardoor verwachten we dat de fit een slechter resultaat zou opleveren. Om de kwaliteit van de fit af te schatten kunnen we de RMSE (uitdrukking 5.1) berekenen. Door middel van simulaties in Matlab zoals in Fig. 10.4 is de RMSE bekeken bij gesimuleerde gevallen van *pile up*. De resultaten van de meting van één puls wordt weergegeven in de situatie waarin we een tweede puls steeds dichter bij deze puls brengen. Deze pulsen zijn te zien in Fig. 10.3. We zien zowel de gemiddelde amplitude als de gemiddelde RMSE (zie uitdrukking 5.1) afgebeeld zijn ten opzichte van de afstand tussen de centra van de twee gaussen. Deze simulaties zijn net als in hoofdstuk 5 uitgevoerd voor een set van verschillende testpulsen, en verschillende pulshoogtes van de tweede puls, met conforme resultaten. Hier duidelijk zien dat de RMSE te veel fluctueert om *pile up* aan af te schatten, een andere parameter is dus vereist.



Figuur 10.4: Deze figuur is een MATLAB simulatie van de invloed van *pile up* op de RMSE in ons systeem. De resultaten van de meting van één puls wordt weergegeven in functie van de afstand van tussen de centra van de twee pulsen.

De breedte van de fit: Een tweede parameter die we zouden kunnen gebruiken is de σ van de gefitte puls. Deze waarde hebben we niet vastgelegd in onze fit (zie sectie 5.5); en we zouden verwachten dat de breedte onder invloed van *pile up* vergroot. Verder kunnen we deze ook op een relatief eenvoudige manier berekenen (zie uitdrukking 4.6), omdat we dezelfde parameters toch al moeten berekenen voor de fit van de amplitude. In Fig. 10.5 is voor dezelfde simulatie als in de vorige paragraaf de evolutie van σ bekeken. We zien hoe de gemiddelde amplitude, de gemiddelde σ, en de gemiddelde $\sigma + 2$ maal de standaard deviatie hierop (95% betrouwbaarheidsinterval) van de fits (σ) afgebeeld zijn ten opzichte van de afstand tussen de centra van de twee gaussen. Deze simulaties zijn weer net als in hoofdstuk 5 uitgevoerd voor een set van verschillende testpulsen, en verschillende pulshoogtes van de tweede puls, met conforme resultaten.



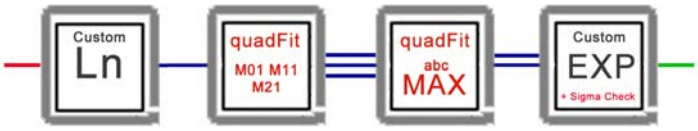
Figuur 10.5: Deze figuur is een MATLAB simulatie van de invloed van *pile up* op de pulsbreedte σ in ons systeem.De resultaten van de meting van één puls wordt weergegeven in functie van de afstand van tussen de centra van de twee pulsen

In deze simulaties wordt zoals in Fig. 10.5 duidelijk dat deze parameter geschikt kan zijn voor het detecteren van *pile up*: de parameter σ is groter in een groot deel van gebied waar *pile up* voorkomt dan in het gebied waar de pulsen ver uit elkaar liggen. Door een *threshold* te installeren op deze sigma zouden we een deel van de *pile up* kunnen weren.

We moeten wel twee belangrijke opmerkingen maken: Ten eerste valt direct op dat het op deze manier onmogelijk is om *pile up* te detecteren in de situatie waarbij er maar een klein tijdsverschil ($< 7\,\mu$s) tussen de twee opeenvolgende pulsen zit. Dit hoeft niet zo te verbazen, het systeem nadert daar immers de situatie waarin de twee gaussische pulsen opnieuw samen een gausscurve vormen. Ten tweede zie we dat er voorzichtig moet worden omgesprongen met de *threshold* die we instellen, we moeten oppassen dat er ook niet teveel goede pulsen geweerd worden uit ons systeem. Om dit te illustreren is aan Fig. 10.5 ook een plot van de waarde van σ waaronder we 97.5% van de waarden van sigma moeten vinden toegevoegd. Er zal een evenwicht gevonden moeten worden tussen de hoeveelheid *pile-up* die we tegenhouden, en het aantal juiste pulsen dat er ook uit gefilterd worden. Dit is een gegeven dat best bekeken wordt aan de hand van de kwaliteit van spectra die we met onze opstelling meten, onder invloed van verschillende *count rates*.

10.3 Implementatie

De implementatie van een systeem dat *pile up* filtert op basis van de σ van de fit is niet zo complex. Het algoritme dat de fit uitvoert moet lichtjes aangepast worden zoals te zien is in Fig. 10.6. In het laatste blok, wordt parallel met de berekening van de exponent (die de amplitude geeft) ook de parameter σ berekend. Daarna wordt op basis van deze σ beslist of de amplitude naar het histogram wordt gecommuniceerd, of wordt weggegooid.



Figuur 10.6: De structuur van het gedeelte in ons programma dat de fit uitvoert (zie hoofdstuk 6, en Fig. 6.3) moet nauwelijks veranderd worden.

De veranderingen in implementatie zijn minimaal: Eerst en vooral moet door het blok dat de kwadratische fit uitvoert een extra stuk informatie doorgegeven worden: de parameter a van de kwadratische functie. Vervolgens moet door het laatste blok van de fit als extra berekening de berekening van σ uitgevoerd worden (uitdrukking 4.6). De logica hiervoor kan ook beperkt worden gehouden: Door in plaats van a de factor $\frac{1}{a}$ (die in het vorige blok toch uitgerekend wordt) door te communiceren moet er geen deling, maar een vermenigvuldiging uitgevoerd worden. Ook de vierkantswortel in deze uitdrukking moeten we niet uitvoeren als we de *threshold* aanpassen en deze op het kwadraat van σ sigma leggen. De resources die deze implementatie op de FPGA nodig hebben worden in Fig. 10.7 weergegeven.

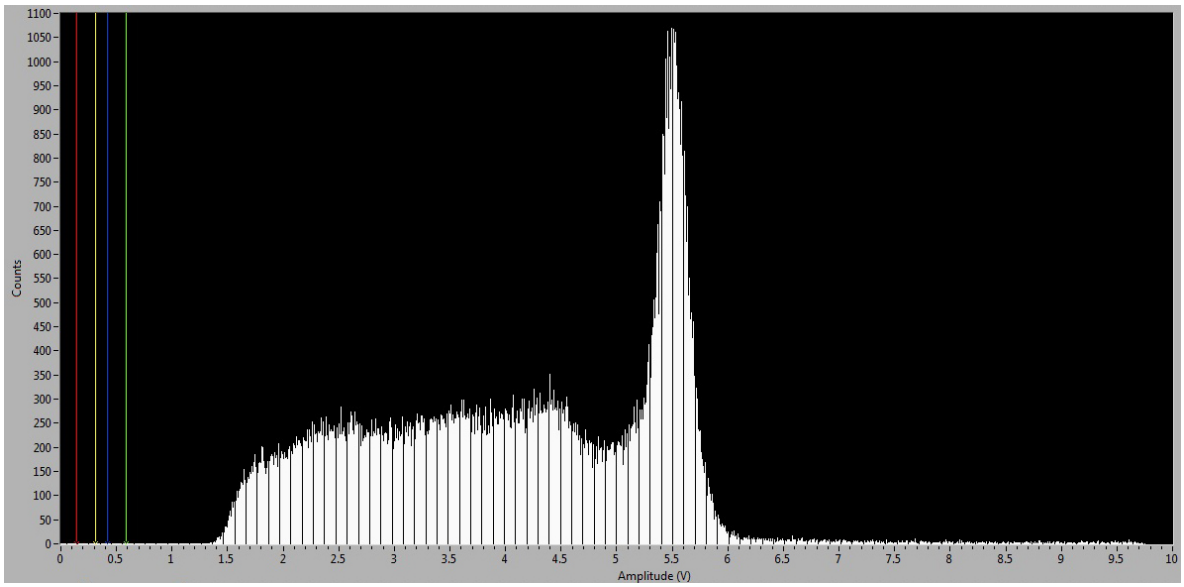
Device Utilization	Used	Total	Percent
Total Slices	2342	4800	48.8
Slice Registers	4989	19200	26.0
Slice LUTs	6012	19200	31.3
DSP48s	18	32	56.2
Block RAMs	11	32	34.4

Figuur 10.7: De voetafdruk van het volledige systeem, inclusief de *pile up rejection* op de FPGA. Het aantal gebruikte block RAMs moet met een korrel zout genomen worden. Omdat we voor het afregelen van het algoritme nog een aantal zaken in de FPGA monitoren met de PC wordt het merendeel gebruikt als buffer voor communicatie met de host. Als we deze schrappen uit de implementatie blijft enkel het block RAM dat het histogram bevat over.

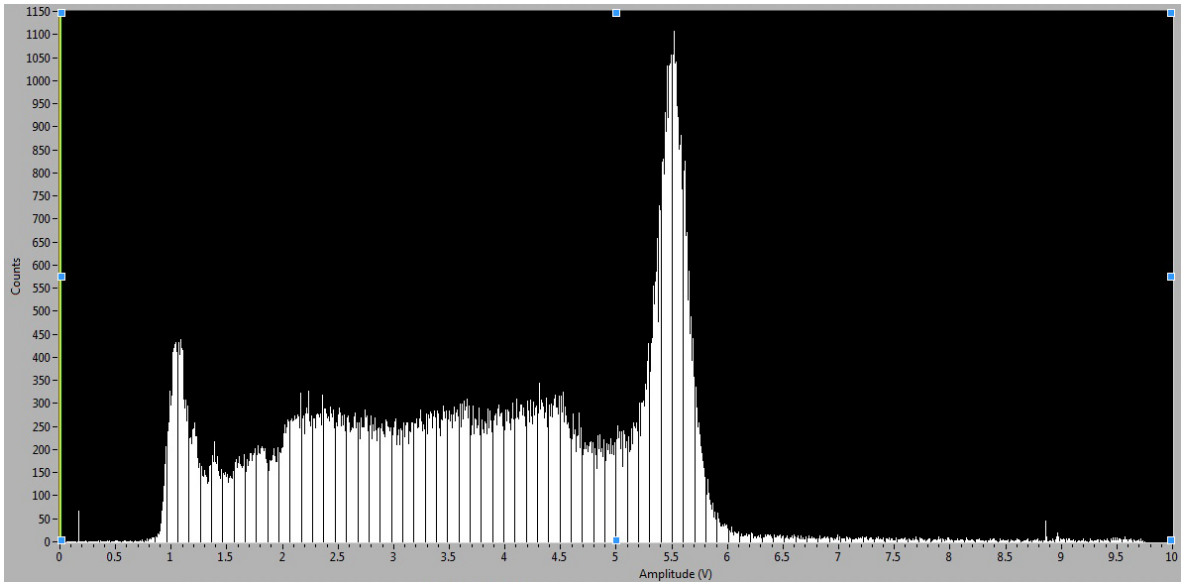
10.4 Resultaten

Om de aanpak van *pile up* te evalueren voeren we twee proeven uit: Ten eerste kijken we aan de hand van een bron met een relatief lage activiteit na of de resultaten gelijkaardig zijn aan het systeem zonder *pile up rejection*. Het systeem zou namelijk in situaties waar nauwelijks *pile up* voorvalt hetzelfde moeten presteren als het basisalgoritme besproken in het vorige hoofdstuk. Ten tweede bekijken we het verschil tussen de situatie met en zonder *pile up rejection* op de metingen van een meer actieve bron. Hierbij verwachten we veel pile-up, en zou onze filter een groot verschil moeten maken.

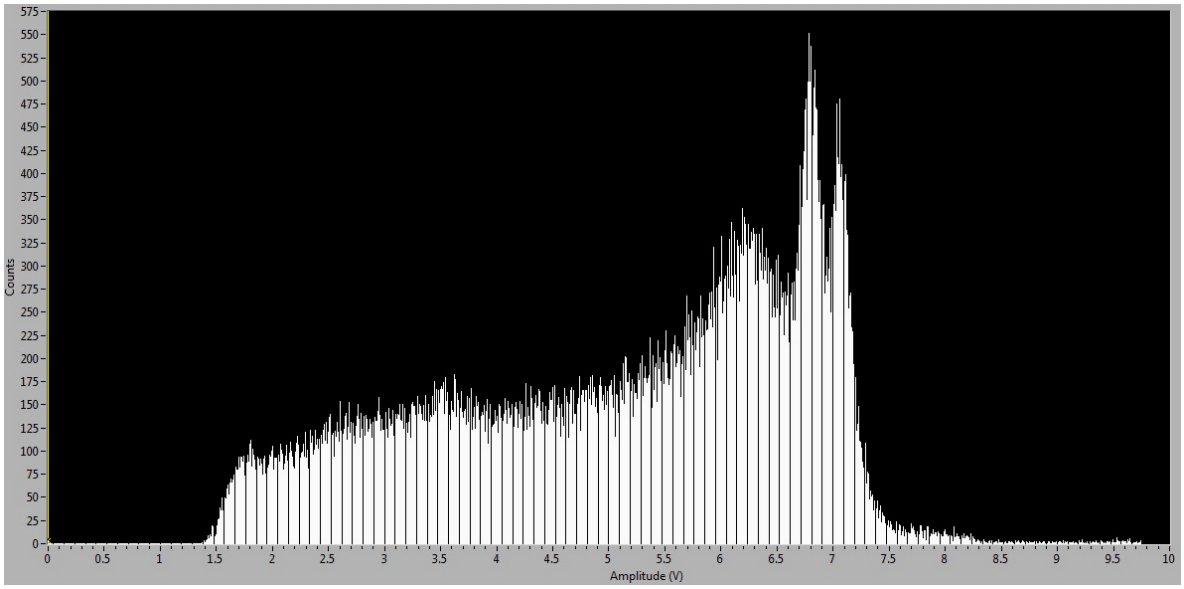
Nauwelijks pile-up Om een beperkte hoeveelheid *pile up* te waarborgen is een bron met een relatief lage activiteit gebruikt voor de metingen. Dit is precies dezelfde opstelling als in het vorige hoofdstuk, om een vergelijking van beide resultaten mogelijk te maken. In Fig. 10.8 en Fig. 10.10 zijn zulke spectra te zien. Een vergelijking met de spectra uit het vorige hoofdstuk leert ons dat de spectra in wezen zeer gelijkaardig zijn. De verschillen vinden we in het lager energetische gebied: hier lijkt de versie met *pile up rejection* een deel van het spectrum weg te filteren. Dit is -hoewel de accuraatheid van de detector bij lagere energieën twijfelachtig is- een ongewenst resultaat. De exacte redenen hiervoor zijn moeilijk aan te halen, maar de ruis zou hier de spelbreker kunnen zijn. Bij lage amplitudes zal de ruis op het signaal meer domineren, waardoor het fit-algoritme, en dus ook de test op σ erg inaccuraat kunnen worden. Het zou dan ook interessant zijn om deze test te herhalen voor pulsen die minder ruis bevatten.



Figuur 10.8: Een opname van een spectrum van een 30 kBq Cs137 bron met *pile up rejection* met een *threshold* van 10 µs op σ. De acquisitie tijd is 100 s. De *pile up rejection* heeft 15 % van de gefitte pulsen tegengehouden, wat leidt tot een acquisitie rate van 1233.2 /s. Vergelijk met Fig. 10.9 voor het geval zonder *pile up rejection*.



Figuur 10.9: Een spectrum opgenomen aan de hand van een 30 kBq Cs137 bron. De acquisitie tijd is 100 s, met een acquisitie rate van 1461.5 /s. Als we vergelijken met het referentie-spectrum in Fig. 9.2 zouden we kunnen zeggen dat we rond 5.5 V de karakteristieke gammapiek van 662 keV waarnemen. Verder vinden we ook de *backscatter* terug bij 2.1 V en de *Compton edge* bij 4.5 V.



Figuur 10.10: Een opname van een spectrum van een 12 kBq Co60 bron met *pile up rejection* met een *threshold* van 10 µs op σ. De acquisitie tijd is 207 s. De *pile up rejection* heeft 21 % van de gefitte pulsen tegengehouden, wat leidt tot een acquisitie rate van 531.7 /s. Vergelijk met Fig. 9.3 voor het geval zonder *pile up rejection*.

In het hoger energetische gebied kunnen we verder twee waarnemingen doen, die we kunnen analyseren aan de hand van tabel 10.1: Ten eerste zien we dat we den pieken van onze bronnen op dezelfde plaatsen vinden als in het vorige hoofdstuk. Het *pile up rejection* systeem introduceert dus geen bijkomende bias. Ten tweede lijkt het erop dat de resolutie van de pieken toch iets verbetert. Een mogelijke verklaring voor dit laatste ligt in het feit er een klein aantal gevallen van *pile up* gefilterd worden.

Er rest ons nog te zeggen dat deze testen zijn uitgevoerd voor verschillende waarden voor de *threshold* op σ tussen 8 µs en 12 µs, met gelijkaardige resultaten tot gevolg. We kunnen concluderen dat hoewel het algoritme niet optimaal is bij lage energiën het *pile up rejection* algoritme wel gelijkaardige resultaten levert als het algoritme zonder. We kunnen er nu van uitgaan dat dezelfde bespreking als in het vorige hoofdstuk zal gelden voor dit algoritme, zolang er niet veel pile-up voorvalt.

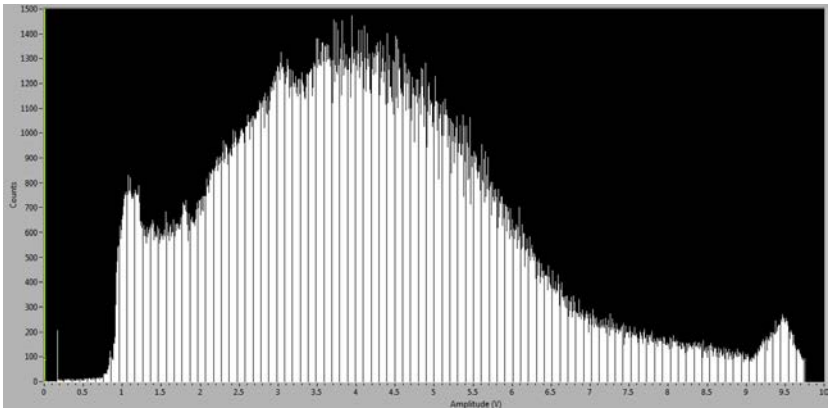
Bron	Piek (keV)	Kanaal Ref.	Resolutie Ref.	Piek positie (V)	Resolutie (%)
Cs137	662	572	7.69	5.52	16.6
Ba133	356	geen data	geen data	3.82	13.9
Co60	1170	998	5.21	6.80	13.9
Co60	1330	1130	4.78	7.07	10.6

Tabel 10.1: Een tabel die literatuurwaarden van pieken gemeten met een NaI scintillatiedetector van verschillende bronnen naast de waarden gemeten met het referentiesysteem (Ref.) (zie hoofdstuk 9), en de waarden waargenomen in ons systeem plaatst. Vergelijk ook met de situatie zonder *pile up rejection* in tabel 9.1

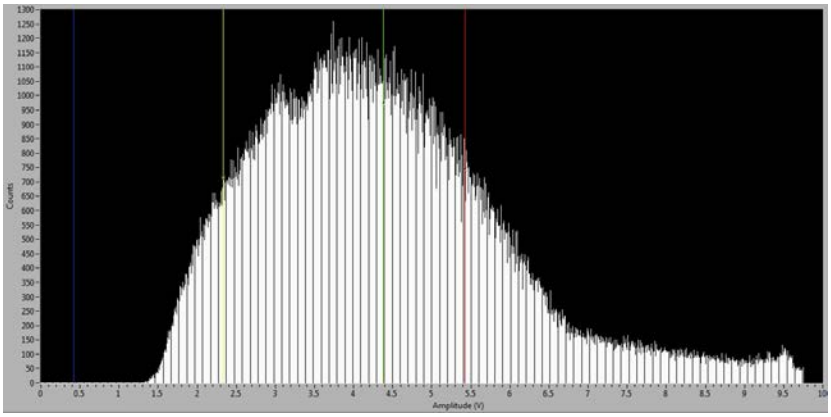
.

Gedrag bij sterke pile-up De echte test voor het systeem is natuurlijk in een situatie waar *pile-up* het spectrum gemeten met het basisalgoritme volledig verandert. Deze situatie verkrijgen we door meer actieve bron op te meten, en een voorbeeld hiervan is te zien in Fig. 10.11. In deze figuur is de structuur van de bron verdwenen (vergelijk met Fig. 9.1). *pile up rejection* zou dit probleem moeten oplossen, of toch op zijn minst een verbetering van het spectrum moeten opleveren. Als we echter naar dezelfde meting met *pile up rejection* in Fig. 10.12 zien we dat er niets gebeurt.

Deze proeven zijn uitgevoerd voor verschillende *threshold* waarden voor σ, met allemaal een gelijkaardig resultaat. We kunnen dus concluderen dat -hoewel de simulaties een veelbelovend resultaat voorspellen- deze aanpak van *pile up* niet werkt. De eerste reden die wij hiervoor kunnen aanhalen is dat de ruis op de pulsen het moeilijk maakt om accuraat *pile up* te filteren. Een tweede reden vinden we in het feit dat bij *pile up* de *baseline* van de pulsen sterk omlaag getrokken kunnen worden. Bij signalen waar veel *pile up* in zit kan de *baseline* dan ook sterk variëren, wat pieken sterk kan verplaatsen en uitsmeren.



Figuur 10.11: Een opname van een spectrum van een 873 kBq Cs137 bron zonder *pile up rejection*. De acquisitie tijd is 63 s, met een acquisitie rate van 9685.8 /s.



Figuur 10.12: Een opname van een spectrum van een 873 kBq Cs137 bron met *pile up rejection* met een *threshold* van 10 µs op σ. De acquisitie tijd is 61 s. De *pile up rejection* heeft 26 % van de gefitte pulsen tegengehouden, wat leidt tot een acquisitie rate van 7119.4 /s.

Hoofdstuk 11

Conclusies

In dit proefschrift hebben we het gebruik van een High Level Synthesis tool voor een FPGA applicatie onderzocht. Dit heeft als doel om te evalueren hoe toegankelijk de FPGA technologie met deze tools wordt voor wetenschappers die geen sterke achtergrond in de elektronica hebben. Praktisch werd dit uitgevoerd door een fysicus die voor de eerste keer met deze technologie in aanraking komt, de FPGA te laten benaderen met behulp van de FPGA module van de grafische programmeeromgeving NI LABVIEW.

Om dit te onderzoeken is een testcase met bestaande gammaspectroscopie hardware opgezet. Hierin krijgt de FPGA de opdracht om de taak van de multi channel analyser over te nemen. Om dit te bereiken werd op de FPGA een niet-triviaal Pulse Height Analysis algoritme geïmplementeerd dat via fits aan de signalen van gedetecteerde gammastraling de energieën van deze straling afschat.

Omwille van de beperkte logica waar een FPGA over beschikt was het eerste belangrijke zwaartepunt in onze ontwikkelingsstrategie het streven naar zoveel mogelijk mathematische eenvoud. Dit streven bestond vooral uit het kiezen voor een eenvoudige kwadratische fitmethode en het optimaliseren van de precisie van de mathematische bewerkingen. Voor deze optimalisaties bleek de kennis van fixed point arithmetic, en een diepgaand begrip van de FPGA technologie noodzakelijk.

Omdat een FPGA weinig flexibele programmatuur toelaat, en het compileren van FPGA firmware een tijdrovend proces is komen we bij het tweede belangrijke zwaartepunt in de ontwikkelingsstrategie: Om de prestaties van ons systeem a priori te evalueren en de optimale configuratie uit verschillende variaties op ons systeem te selecteren hebben we de performantie hiervan geëvalueerd middel van computersimulaties.

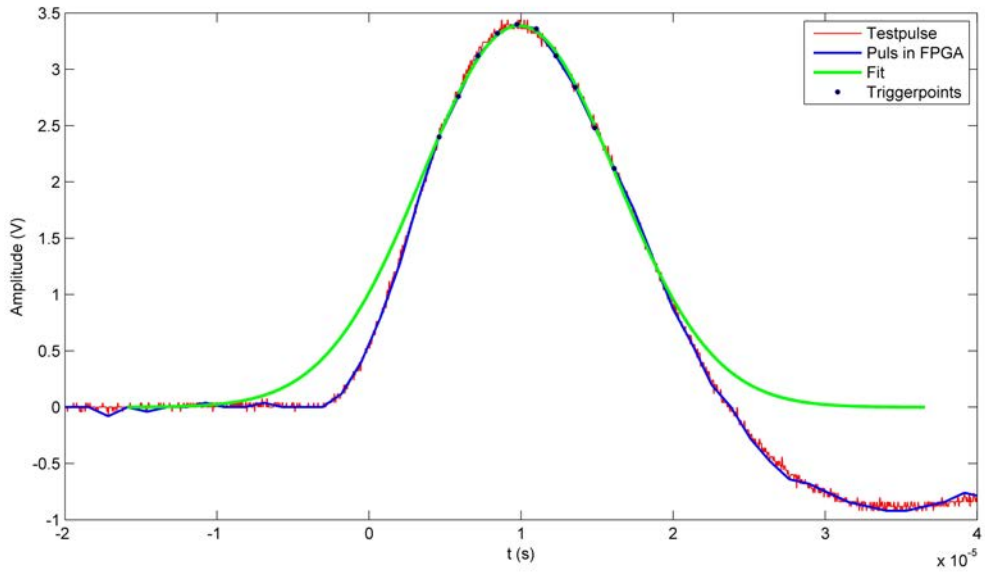
Aan de hand van deze eenvoudige strategieën zijn we erin geslaagd om een werkend systeem op te zetten dat de performantie van een commercieel systeem benadert.

Hieruit kunnen we concluderen dat de FPGA een krachtige technologie is die met behulp van High Level Synthesis Tools toegankelijk wordt voor niet-specialisten. Toch is deze toegankelijkheid niet onvoorwaardelijk: een goed begrip van de FPGA technologie is een vereiste.

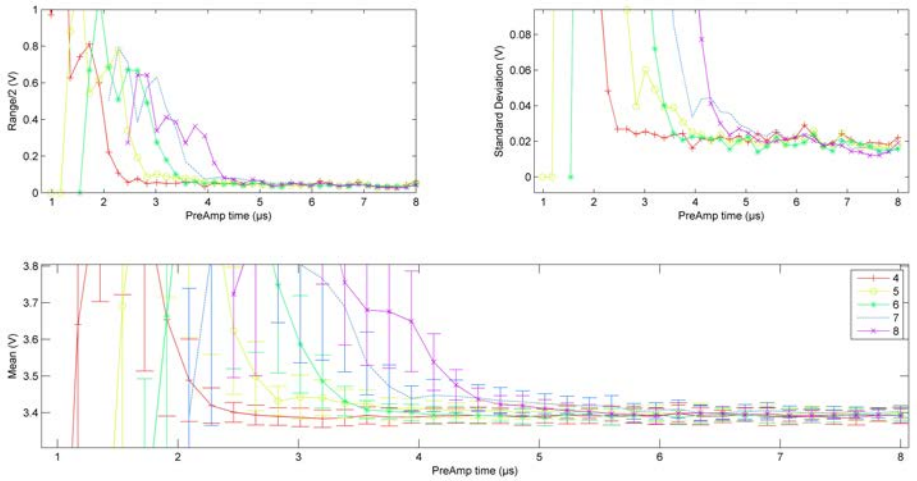
Bijlage A

De studie van een tweede testpuls

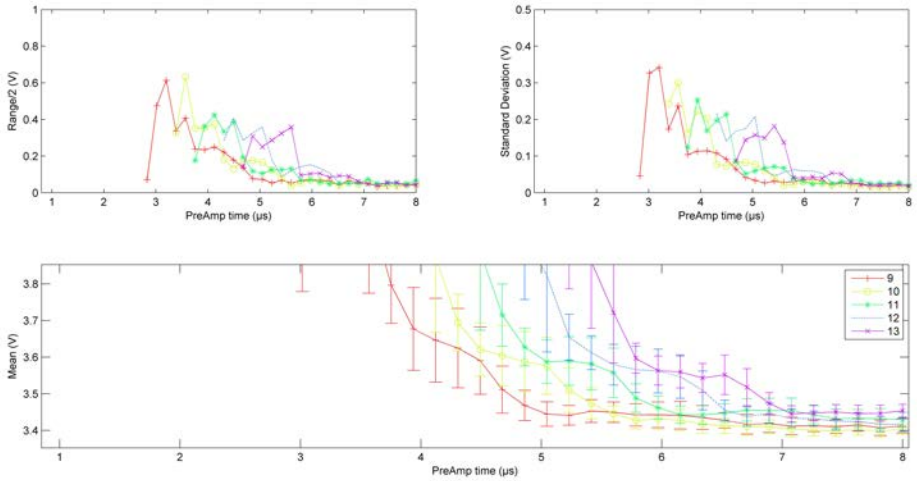
In deze bijlage geven we de resultaten van simulaties zoals beschreven in sectie 5.3 voor een tweede testpuls. Dit om de kritische lezer ervan te overtuigen dat onze proeven eenzelfde resultaat geven voor de 10 verschillende testpulsen die we gebruikt hebben. De motivatie om een puls weer te geven die zich in de lagere regionen van de te meten amplitude bevind is tweeledig: Enerzijds is dit om aan te tonen dat het algoritme vergelijkbaar presteert over de hele band van waar te nemen amplitudes (0 V - 8 V). Anderzijds is dit om met de oscilloscoop een lager ruisniveau op de gemeten testpuls te verkrijgen.



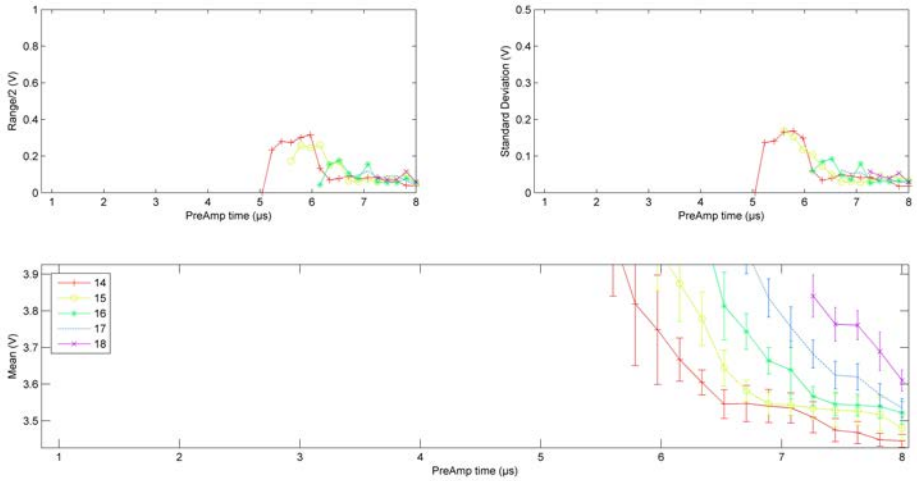
Figuur A.1: Een testpuls opgenomen met onze opstelling, met een *shaping time* van 8 µs. De maximale amplitude bepaalt met de oscilloscoop is 3.44 V.



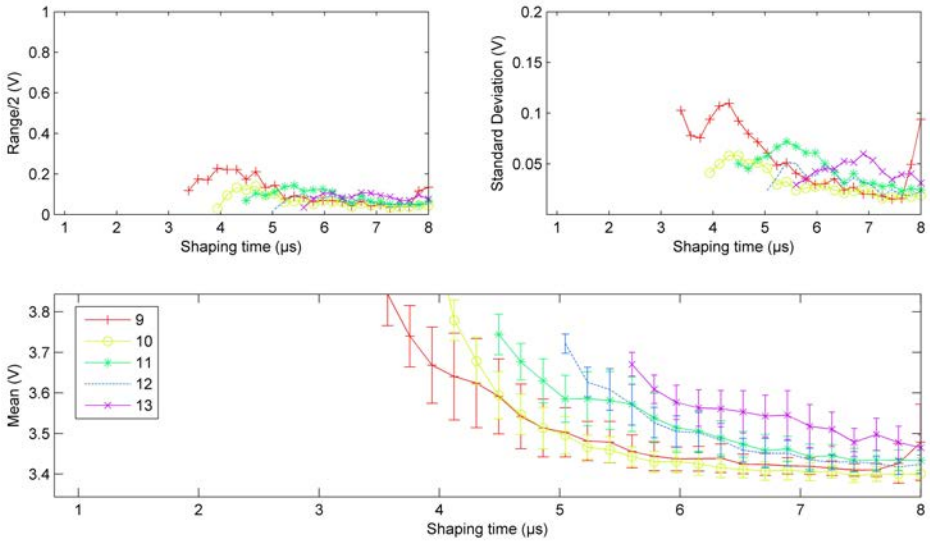
Figuur A.2: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.



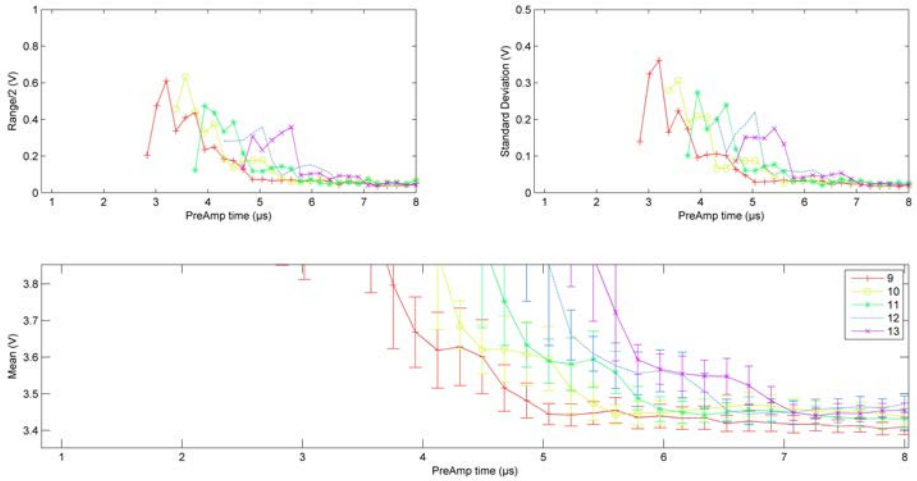
Figuur A.3: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.



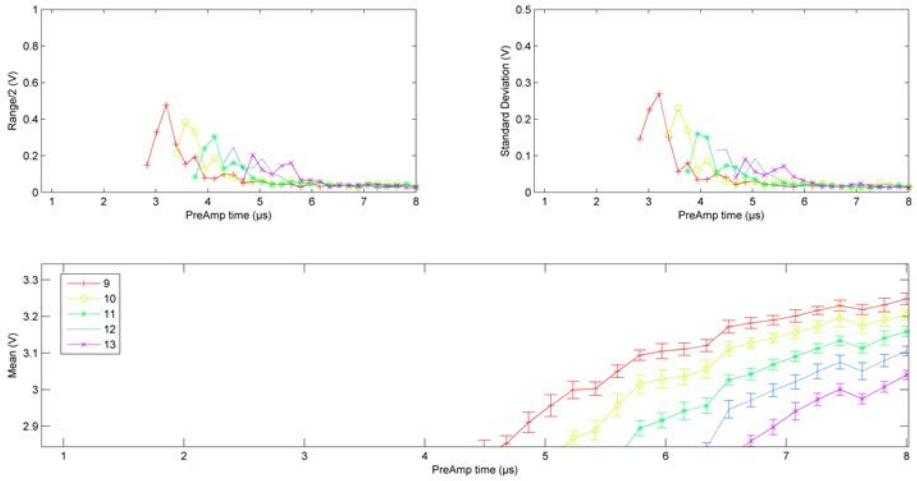
Figuur A.4: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor.



Figuur A.5: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier is gebruikt kiest een vast aantal punten van de puls op zo'n manier dat een zo groot mogelijk deel van de puls gesampled wordt met equidistante punten.



Figuur A.6: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier gebruikt is kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer voor het maximum genomen dan erna.



Figuur A.7: We zien hier de resultaten van de toepassing van ons algoritme voor verschillende puntenaantallen op één testpuls met een maximale amplitude van 3.44 V (Fig. A.1). De trigger die hier is gebruikt kiest een vast aantal punten rond het maximum, en voor even puntenaantallen wordt er één punt meer na het maximum genomen dan ervoor. Het fit-algoritme maakt hier gebruik van de gekende waarde van σ, die wordt geleverd door de *shaping time* van de *shaping amplifier*.

Bibliografie

[1] R. S. Peterson, Experimental γ ray spectroscopy and investigations of environmental radioactivity. The Unisversity of the South Sewanee, Tennessee, 1996.

[2] W. R. Leo, Techniques for Nuclear and Particle Physics Experiments. Springer-Verlag, 2nd ed., 1994.

[3] S. N. Ahmed, Physics and Engineering of Radiation Detection. Academic Press, 1st ed., 2007.

[4] U.S. NIM Committee, Standard NIM Instrumentation System, Mei 1990.

[5] FAST ComTec GmbH, CSA4 user manual, 1.0 ed., December 2009.

[6] H. Andrianiaina, R. Andriambololona, J. Rajaobelison, G. Rambolamanana, and H. Röngen, "Fpga-based technology for pulse height analysis in nuclear spectrometry system," in Proceedings of the Fourth High-Energy Physics International Conference, (Antananarivo), Augustus 2009.

[7] P. Tsao and H. Chou, "Nuclear pulse height measurement using fpga techniques," in Nuclear Science Symposium Conference Record, 2008 NSS '08, pp. 2015–2017, Oktober 2008.

[8] S. Stanko, B. Klein, and J. Kerp, "A field programmable gate array spectrometer for radio astronomy," Astronomy and Astrophysics, vol. 436, pp. 391–395, June 2005.

[9] I. Elhanany, S. Jacobi, M. Kahane, E. Marcus, D. Tirosh, and D. Barak, "A novel architecture for digital pulse height analysis with application to radiation spectroscopy," in Proocedings of the 7th Mediterranean Conference on Control and Automation, (Haifa, Israel), pp. 2143–2151, June 1999.

[10] H. Andrianiaina, R. Andriambololona, J. Rajaobelison, G. Rambolamanana, and H. Röngen, "Fpga-based technology for pulse height analysis in nuclear spectrometry system," HEPMAD'09 Conference, August 2009.

[11] W. Jinyuan, M. Wang, E. Gottschalk, and Z. Shi, "Fpga curved track fitter with very low resource usage," Nuclear Science Symposium Conference Record, 2006. IEEE, vol. 2, pp. 1290–1295, Oktober 2006.

[12] LABVIEW FPGA Course Exercises, october 2010 edition ed., 2010.

[13] Spectrum Techniques, Inc., Spectrum Techniques UCS-20 Universal Computer Spectrometer.

[14] H. Spieler, Introduction to Radiation Detectors and Electronics. UC Berkerley, 198.