Exact Inner Product Processor in FPGA

Luís Filipe Azenhas Fiolhais

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. Horácio Cláudio De Campos Neto

Examination Committee
Chairperson: Prof. António Manuel Raminhos Cordeiro Grilo
Advisor: Prof. Horácio Cláudio De Campos Neto
Members of the Committee: Prof. Pedro Filipe Zeferino Aidos Tomás

November 2017
Abstract

The objective of this work is to develop an efficient hardware processor to perform the inner product operation with full accuracy. The floating-point inner product is ubiquitous in industrial and scientific applications. However, general purpose processors split the inner product into two independent operations, rounding the full precision results of the multiplication and accumulation to the format-precision, which may significantly affect the overall accuracy. Therefore, many systems must rely on software approaches to achieve numerical accuracy, which imposes a significant performance overhead.

This work uses a long-accumulator with full fixed-point precision to achieve full accuracy. The accumulation is split into small segments (adders and registers) to break the wide critical delay path. Two segmented accumulator architectures are proposed, stalling and non-stalling. The stalling architecture has a lower resource consumption, but stops the accumulation process whenever there is a carry to propagate between segments. The non-stalling architecture uses an autonomous carry propagation unit to maximize performance, but requires extra resources. Both architectures use a Generalized Signed-Digit redundant numeric representation to support signed addition without propagating the sign between segments every cycle.

The proposed processor has been implemented in a Zynq 7010-1 FPGA. A single-precision non-stalling core can execute with a clock frequency of 80 MHz and occupies about 6K LUTs. Full accuracy has been demonstrated using a set of hard to correctly solve benchmarks. The results obtained with a real-world credit risk analysis example also confirm that the processor provides a significantly better accuracy than traditional arithmetic for the same operand-precision.

**Keywords:** exact inner product, floating-point, long-accumulator, segmented accumulator, FPGA, Generalised Signed Digits
Resumo

O objectivo deste trabalho é desenvolver um processador eficiente em hardware para executar o produto interno com exactidão total. O produto interno com vírgula-flutuante é ubíquo em aplicações industriais e científicas. Contudo, processadores de uso geral dividem o produto interno em duas operações independentes, arredondando os resultados da acumulação e multiplicação com precisão total para a precisão do formato, o qual pode afectar a exactidão global. Assim sendo, muitos sistemas usam software para obter precisão numérica, o que afecta o desempenho.

Este trabalho usa um acumulador-longo com precisão total de vírgula-fixa para obter exactidão total. A acumulação é dividida em segmentos pequenos (somadores e registos) para quebrar o grande caminho crítico. Duas arquitecturas de acumuladores segmentados são propostas, stalling e non-stalling. A arquitectura stalling consome poucos recursos, mas para a acumulação sempre que tem carries para propagar entre segmentos. A arquitectura non-stalling maximiza o desempenho com uma unidade autónoma de propagação de carry, mas requer mais recursos. Ambas as arquitecturas usam uma representação numérica redundante (Generalized Signed-Digit) para suportar adição com sinal sem propagar o sinal entre segmentos todos os ciclos.

O processador proposto foi implementado na Zynq-7010-1-FPGA. Um núcleo de precisão-simples non-stalling opera com uma frequência de relógio de 80 MHz e ocupa 6K LUTs. Exactidão total foi demonstrada utilizando um conjunto de testes difíceis de resolver correctamente. Os resultados obtidos com um exemplo de análise de crédito de risco também confirmam que o processador fornece uma exactidão significativamente melhor que aritmética tradicional para a mesma precisão de operandos.

Palavras-Chave: produto interno exacto, vírgula flutuante, acumulador longo, acumulador segmentado, FPGA, Generalised Signed Digits
# Contents

**List of Tables** viii  
**List of Figures** x  
**Acronyms** xiii  

## 1 Introduction 1  
1.1 Motivation .................................................. 1  
1.2 Goals .......................................................... 4  
1.3 Thesis Outline ................................................. 4  

## 2 Inner Product Computation 6  
2.1 Floating-Point Arithmetic .................................... 6  
2.1.1 IEEE 754 Format ............................................. 6  
2.1.2 Basic Floating-Point Operations ............................ 7  
2.1.2.1 Fused Multiply-Add ..................................... 8  
2.2 Inner Product .................................................. 9  
2.2.1 Accumulation Solutions .................................... 10  
2.2.1.1 Addition with Remainder ................................ 10  
2.2.1.1.1 Pichat and Kahan Accuracy Tests ................. 12  
2.2.1.1.2 Kahan’s Algorithm Improvements .................. 12  
2.2.1.2 Long Accumulator ....................................... 13  
2.2.2 Hardware Implementation Analysis ......................... 15  
2.3 Long Accumulator Architectures .............................. 16  
2.3.1 Non-Segmented Long Accumulator .......................... 16  
2.3.2 Segmented Long Accumulator ............................... 17  
2.4 Base Inner Product Core Architecture ....................... 19  
2.5 Conclusion ................................................... 20  

## 3 Segmented Accumulator Stalling Architecture 22  
3.1 Basic Stalling Architecture .................................. 22  
3.2 Carry Processing Block Architecture ......................... 23  
3.3 Carry Processing Optimisation ................................ 25  
3.4 Architecture Analysis ......................................... 28  
3.5 Conclusion ................................................... 30
# Segment Non-Stalling Architecture

## Segment Non-Stalling Architecture

### Basic Non-Stalling Architecture

### Hazard Identification & Solutions

#### Read After Write Hazard Analysis

#### Read After Write Hazard Solution

#### Carry Write Conflict Hazard Analysis

#### Carry Write Conflict Hazard Solution

## Conclusion

## Segmented Accumulator Addition/Subtraction Architectures

### Generalised Signed-Digit Number Representations

### Using GSD Numbers in the Segmented Accumulator

#### Conversion from GSD to IEEE 754

### Adding GSD Support to the Exact Inner Product Core

#### Stalling Segmented Accumulator

#### Non-Stalling Segmented Accumulator

## Conclusion

## Exact Inner Product Core Results Analysis

### FPGA Implementation Results

#### Stalling Segmented Accumulator Architecture

#### Non-Stalling Segmented Accumulator Architecture

### Stalling and Non-Stalling Architectures Comparison

### Performance and Resources Analysis

### Evaluation Results

#### Accuracy Tests Results

#### Credit Risk Analysis Example

## Conclusion

## Future Work

## Bibliography

## Programs used for Accuracy Tests
## List of Tables

1. **Vectors** $x$ and $y$ ................................................................. 2
2. **Vectors** $a$ and $b$ ................................................................. 2
3. **Step by step results of the inner product of the vectors** $a$ and $b$ using single-precision arithmetic ................................................................. 2
4. **Correct step by step results of the inner product of the vectors** $a$ and $b$ ................................................................. 3
5. **IEEE 754 32, 64 and 128 bit formats** ................................................................. 7
6. **Relative error obtained for the tests data set, from [2]** ................................................................. 12
7. **Comparison between all algorithms described using resources, critical path and accuracy** ................................................................. 16
8. **Start Position relative to register selection for each combination of carries** ................................................................. 26
9. **Minimum Number of Cycles for each carry combination with three carries** ................................................................. 27
10. **Minimum Number of Cycles for each carry combination with four carries** ................................................................. 27
11. **GSD variables description** ................................................................. 46
12. **Addition steps for** $u_i = [−8, 7]$ and $c_i = [−2, 2]$ ................................................................. 50
13. **Carry code decode for carry digit set** $c_i = [−2, 2]$ ................................................................. 50
14. **Addition steps for** $u_i = [−16, 15]$ and $c_i = [−4, 4]$ ................................................................. 52
15. **Carry code decode for carry digit set** $c_i = [−4, 4]$ ................................................................. 53
16. **Conversion to radix-32** ................................................................. 55
17. **Clock Frequency for the Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the clock frequency for three different segment sizes $y$: 8 bits, 16 bits and 32 bits** ................................................................. 65
18. **Resource Usage for the Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the resource usage for three different segment sizes $y$: 8 bits, 16 bits and 32 bits** ................................................................. 65
19. **Maximum and minimum carries, and their register size to avoid stalling** ................................................................. 66
20. **Clock Frequency for the Non-Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the clock frequency for three different segment sizes $y$: 8 bits, 16 bits and 32 bits** ................................................................. 66
21. **Resource Usage for the Non-Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the resource usage for three different segment sizes $y$: 8 bits, 16 bits and 32 bits** ................................................................. 66
22. **Resources used by the proposed single-precision architectures (stalling and non-stalling) and the single-precision Multiply-Accumulator (MAC) unit** ................................................................. 68
23. **Result comparison between Arbitrary Precision, the Single-Precision Exact Inner Product Core (EIPPC) and Traditional Single-Precision Arithmetic** ................................................................. 69
6.8 Coefficients obtained for the Logistic Regression using traditional single-precision arithmetic, the single-precision exact inner product core (EIPC) and traditional double precision arithmetic, and number of correct digits (NCD) for the traditional single-precision arithmetic and the single-precision inner product core
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>IEEE 754 Single-Precision format for 1.5</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Block Diagram of a Standard Addition Floating-Point Unit, based in [7]</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Block Diagram of a Standard Floating-Point Fused Multiplier-Adder Unit, based in [10]</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>LA register [3]</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>LA blocks</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Segmented Accumulator with 64 bit segments</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Proposed accumulator in [33]</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>k bit critical path</td>
<td>17</td>
</tr>
<tr>
<td>2.9</td>
<td>Single-Precision Segmented Accumulator with 32 bit registers</td>
<td>18</td>
</tr>
<tr>
<td>2.10</td>
<td>Single-Precision Segmented Accumulator with 32 k Memory</td>
<td>19</td>
</tr>
<tr>
<td>2.11</td>
<td>Multiplication and Accumulation Stages of the Inner Product Core</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>Fixed-Point to IEEE 754 Single Precision Conversion stages of the Inner Product Core</td>
<td>20</td>
</tr>
<tr>
<td>3.1</td>
<td>Execution Stage of the processor with full-width adder</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Segmented Accumulator with three adders and a CPB</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>First three stages of the CPB</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Segmented Accumulator with one adder in the CPB</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Relationship between the segment size (bits) and number of adders, for four floating-point with 16, 32, 64 and 128 bits</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>Datapath for the Non-Stalling architecture with three adders in the EX stage and one in the FFA (depicted inside the dashed box)</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>RAW hazard, where S1 is summand 1 and S2 is summand 2</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>CWC hazard</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>Datapath for the Segment Fetch Stage with three adders in the Execute Stage and one FFA</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Final Datapath with three adders in the Execute Stage and one FFA (depicted with a dashed box)</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Relation between the number of bits in the carry register and segment size, for Floating-Point sizes of 16, 32, 64 and 128 bits</td>
<td>40</td>
</tr>
<tr>
<td>4.7</td>
<td>Relation between the carry register size and number of stalls, with four segment sizes: 8, 16, 32 and 64 bits</td>
<td>41</td>
</tr>
<tr>
<td>4.8</td>
<td>Relation between the number of bits in the carry register and the number of adders in the EX stage, for Floating-Point sizes of 16, 32, 64 and 128 bits</td>
<td>43</td>
</tr>
<tr>
<td>4.9</td>
<td>Relation between the carry register size and number of stalls, with four Number of Adders: 2, 3, 4 and 5</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Block Diagram of an addition and subtraction using GSD numbers in the segmented architecture</td>
<td>54</td>
</tr>
</tbody>
</table>
5.2 Block Diagram of the conversion unit .......................... 56
5.3 Shift and Accumulation result of the three digits, \(d_0, d_1\), and \(d_2\) .......................... 57
5.4 Shift and Accumulation result of the two digits, \(d_1\) and \(d_2\). The final significand is underlined on the accumulation result .......................... 58
5.5 Block Diagram of the Multiplication and Accumulation Stages in the Inner Product Core .......................... 61
5.6 Block Diagram of the Conversion Stages in the Inner Product Core .......................... 62

6.1 Clock frequency and LUT's ratio between the non-stalling and stalling architectures using a 32-bit segment size, for three floating-point sizes: 16 bits, 32 bits and 64 bits .......................... 67
6.2 The single-precision Floating-Point Multiply-Accumulate unit is divided in two blocks, one multiplier and one accumulator .......................... 67
6.3 System used to test each vector .......................... 70

7.1 Optimised Conversion Unit Digit Selection .......................... 75
Acronyms

BRAM  Block Random Access Memory. 75

CPB  Carry Processing Block. 22, 26, 28, 30, 31, 62, 65, 73–75

CWC  Carry Write Conflict. 33, 34, 39, 44, 45

EX  Execute. 31, 32, 34–36, 40–45, 62

FFA  Free Flow Adder. 31–33, 36–39, 44, 66, 67, 73–75

FPGA  Field Programmable Gate Array. 4, 5, 16, 64–66, 74, 75

GLM  Generalized Linear Model. 71

GPP  General Purpose Processor. 1–9, 68

GPU  Graphical Processing Unit. 9

GSD  Generalized Signed-Digit. 46, 47, 53–57, 61–63, 74, 75

IEEE  Institute of Electrical and Electronics Engineers. 1–3, 6, 7, 12, 19, 23, 28, 29, 39, 41–43, 54–57, 63, 64, 68, 71, 74, 75

IRLS  Iteratively Reweighted Least Squares. 71

LA  Long Accumulator. 13, 14

LSB  Least Significant Bit. 17, 47, 58

LUT  Look Up Table. 67, 75

LZD  Lead Zero Detector. 19, 56, 75

MSB  Most Significant Bit. 17, 47, 54, 58

NaN  Not a Number. 7, 74, 75

RAW  Read After Write. 32–34, 45

SF  Segment Fetch. 31–36, 44, 62, 66

SIMD  Single Instruction, Multiple Data. 9

WB  Write Back. 31, 34–36, 44, 45, 62, 66
Chapter 1

Introduction

The objective of this work is to develop an efficient hardware processor to perform the inner product operation with full accuracy. This chapter describes the motivation to study the inner product operation, the goals of the project and the report outline.

1.1 Motivation

In scientific computations high performance floating-point arithmetic is frequently required. The Institute of Electrical and Electronics Engineers (IEEE) 754 standard in [1] defines five binary floating-point formats to choose from (16-bit to 128-bit sizes), each with specific degrees of precision. In fact, the most used operation sequence is a multiplication followed by an addition, which justified the IEEE standardisation of a fused multiply-add operation [1]. However, there is still no standard for the inner product operation:

\[ s = \sum_{i=0}^{n-1} x_i \times y_i. \]

A unit optimised to compute the inner product will process a multiplication on the elements of two vectors followed by a continuous accumulation of the multiplication results. To maintain accuracy the result of the multiplication must be used directly, without any rounding, in the accumulation. The same must occur for the accumulation, when adding, all digits of the accumulator must be used. Since the inner product operation is ubiquitous this is a current relevant research subject in computer architectures.

In a General Purpose Processor (GPP) the inner product is split into two operations: one multiplication and one addition. This separation causes errors in the computation because of the intermediate roundings.

Example 1.1.1. From [2], consider the following two vectors \( x \) and \( y \), of size \( 10^{16} + 1 \) whose elements are represented using the IEEE 754 single-precision format.
Table 1.1: Vectors $x$ and $y$

<table>
<thead>
<tr>
<th>ID</th>
<th>$x$</th>
<th>$y$</th>
<th>Incorrect Result</th>
<th>Correct Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>$10^{-16}$</td>
<td>1.0</td>
<td>1.0 + $10^{-16}$</td>
</tr>
<tr>
<td>2</td>
<td>1.0</td>
<td>$10^{-16}$</td>
<td>1.0</td>
<td>1.0 + $2 \times 10^{-16}$</td>
</tr>
<tr>
<td>10^{16}</td>
<td>1.0</td>
<td>$10^{-16}$</td>
<td>1.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

The correct result of the inner product is 2.0 but the result of the computation is 1.0. This error occurs because in each accumulation the smallest element must be aligned with the accumulation result. Since the accumulator holds a value which is much greater than the element, the smallest value after the alignment is incorrectly inferred as zero. Furthermore, this issue is also present using double-precision arithmetic.

Example 1.1.2. From [3], consider two vectors $a$ and $b$ of size five whose elements are represented using the IEEE 754 single-precision format.

Table 1.2: Vectors $a$ and $b$

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27 182 818 280</td>
<td>1 486 249 700 000</td>
</tr>
<tr>
<td>2</td>
<td>−31 415 926 540</td>
<td>878 366 987 900 000</td>
</tr>
<tr>
<td>3</td>
<td>14 142 135 620</td>
<td>−22 374 920 000</td>
</tr>
<tr>
<td>4</td>
<td>5 772 156 649</td>
<td>4 773 714 647 000 000</td>
</tr>
<tr>
<td>5</td>
<td>3 010 299 957</td>
<td>185 049</td>
</tr>
</tbody>
</table>

The results of the inner product step by step using single-precision arithmetic are in Table 1.3

Table 1.3: Step by step results of the inner product of the vectors $a$ and $b$ using single-precision arithmetic

<table>
<thead>
<tr>
<th>OP</th>
<th>Multiplication Result</th>
<th>Accumulation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0 \times b_0$</td>
<td>40 400 454 688 051 390 578 688</td>
<td>40 400 454 688 051 390 578 688</td>
</tr>
<tr>
<td>$a_1 \times b_1$</td>
<td>−27 594 712 738 320 030 418 075 648</td>
<td>−27 554 312 062 957 597 286 342 656</td>
</tr>
<tr>
<td>$a_2 \times b_2$</td>
<td>−316 429 137 452 910 772 224</td>
<td>−27 554 627 963 447 859 562 414 080</td>
</tr>
<tr>
<td>$a_3 \times b_3$</td>
<td>27 554 625 657 604 850 348 720 128</td>
<td>−2 305 843 009 213 693 952</td>
</tr>
<tr>
<td>$a_4 \times b_4$</td>
<td>557 052 996 878 336</td>
<td>−2 305 285 969 135 271 936</td>
</tr>
</tbody>
</table>

The correct results of the inner product step by step are in Table 1.4.
Table 1.4: Correct step by step results of the inner product of the vectors \( a \) and \( b \)

<table>
<thead>
<tr>
<th>OP</th>
<th>Multiplication Result</th>
<th>Accumulation Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 \times b_0 )</td>
<td>40 400 455 513 804 516 000 000</td>
<td>40 400 455 513 804 516 000 000</td>
</tr>
<tr>
<td>( a_1 \times b_1 )</td>
<td>(-27 594 712 767 027 468 866 000 000)</td>
<td>(-27 554 312 311 513 664 350 000 000)</td>
</tr>
<tr>
<td>( a_2 \times b_2 )</td>
<td>(-316 429 153 126 650 400 000)</td>
<td>(-27 554 628 740 666 791 000 400 000)</td>
</tr>
<tr>
<td>( a_3 \times b_3 )</td>
<td>27 554 628 740 109 737 903 000 000</td>
<td>(-557 053 097 400 000)</td>
</tr>
<tr>
<td>( a_4 \times b_4 )</td>
<td>557 052 996 742 893</td>
<td>(-100 657 107)</td>
</tr>
</tbody>
</table>

The correct result of the inner product is \(-1.00657107 \times 10^8\) but the result of the computation is \(-2.305286 \times 10^{18}\). This error occurs because the result of the multiplication is rounded, truncated and stored before the accumulation takes place. Furthermore, using double precision does not solve the problem, in fact it is even worse: \(4.328386285 \times 10^9\). All digits, exponent and sign are wrong.

These two examples show that the inner product operation should be computed as a single fused operation, and not as two independent operations. Furthermore, the usage of a [IEEE] format with higher precision does not necessarily guarantee more accurate results.

When programming a numeric system which uses floating-point arithmetic, the programmer must select one specific precision. However, the precision selected may not adequately fit the computation to be performed if one intermediate result exceeds the precision limits of the representation. For example, in both described examples, the inputs and results are within the limits of the single-precision floating-point format, but accumulation and multiplication errors are generated during the computation because one or more intermediate results exceed the limits of the format. To fix this problem programmers typically use a larger floating-point format to cope with the intermediate computation results, even though there is no precision requirement or justification for its usage. As such, choosing an over adequate floating-point can impose a significant overhead when working with floating-point arithmetic, and, most importantly, does not guarantee a correct final result. Furthermore, the current methodology to achieve accuracy in floating-point arithmetic requires the usage of specialised software arithmetic libraries, which do not guarantee full accuracy and add more overhead to the application.

The requirement for an accurate inner product floating-point operation is present in research areas such as financial models, machine learning and 3D graphics, just to name three.

- **In 3D graphics** the inner product operation is used to apply shaders through, for example the Phong algorithm, to 3D objects. Another useful application would be in ray tracing, specifically to compute a light ray intersection or reflection with an object in 3D space. In both cases, a better precision will have a better overall effect in the final object, especially when using complex objects. Improving 3D graphics fidelity is one of the most important aspects in the video game industry which emphasises speed, and special effects studios which emphasise quality.

- **In financial engineering**, more specifically in high frequency trading, the monte carlo method is often used to predict a market’s behaviour and then to buy or sell stock based on the prediction made. In high frequency trading, speed is critical and can mean the difference between making a profit or a loss. Furthermore, due to the nature of the monte carlo method, traditional floating-point arithmetic may have an issue when processing an accumulation of random elements. As a result, the use of an exact inner product operation would be beneficial both in speed and precision.
In **machine learning**, an exact inner product operation would improve the learning mechanism. Having a more precise learning mechanism would potentially yield more precise predictions and would perform better when there is a large amount of variables in the training set. The methodologies which would benefit the most from the exact inner product core are: neural networks, bayesian networks, decision trees, clustering and linear regression.

Therefore, the availability of a fully-accurate hardware implementation of an inner product floating-point operation can add a meaningful value to a significant number of application areas, both in the correctness of the final results and their speed. Furthermore, since there are no rounding errors when processing intermediate results, programmers can make better decisions about which floating-point format should be used in a given situation.

### 1.2 Goals

The goal of the project is to implement a fused inner product hardware architecture that computes accurately the inner product of two vectors, with only one rounding at the end.

The work has four partial goals:
1. Design an architecture which is able to process the inner product accurately;
2. Improve the architecture’s critical path and carry propagation;
3. Support subtractions;
4. Implement the architecture in a **Field Programmable Gate Array (FPGA)** and demonstrate that the inner product is processed accurately.

### 1.3 Thesis Outline

The thesis is organised in the following chapters:

- **Chapter 2**: introduces the state of the art on inner product solutions using floating-point arithmetic. Floating-point representation and its operations are presented, and the inner product operation and its main problems are formally introduced. Two main algorithms are presented and discussed to solve these problems. One of the algorithms is selected for the following hardware architecture analysis. Finally two previously proposed hardware architectures to solve the described problems are discussed and analysed. A base non-segmented inner product core architecture is described, which will be the basis for the following chapters;

- **Chapter 3**: a Stalling Segmented Accumulator Architecture is proposed which improves the accumulation stage. A dedicated block to handle all carry propagation tasks is proposed, which predicts where carries should propagate and how many carries that can be generated during the propagation;

- **Chapter 4**: builds on Chapter 3 conclusions and proposes a Non-Stalling Segmented Accumulator Architecture. A new carry processing block is proposed which allows the architecture to operate without stalling when the accumulator generates carries. Two hazards are analysed which may prevent the architecture to not stall, and solutions are proposed for them;

- **Chapter 5**: adds subtraction support to both previously proposed architectures using redundant signed representation. Modifications are proposed to both architectures and an analysis is performed;
• **Chapter 6**: presents and analyses FPGA implementation results on the inner product core for both proposed segmented accumulator architectures using three floating-point formats. The inner product core is tested using vectors with which traditional floating-point arithmetic has trouble with. Finally, the inner product core is evaluated using a real world example of credit risk analysis;

• **Chapter 7**: a conclusion to the work is presented along with future work to be done.
Chapter 2

Inner Product Computation

This chapter introduces the state of the art on inner product solutions using floating-point arithmetic. The first section describes the floating-point representation, and introduces the IEEE 754 floating-point format, the rounding modes available and the main components of the basic architecture to compute the floating-point addition.

The second section introduces the inner product operation in current architectures and its implementation difficulties. The main problems are detailed and two possible algorithms are presented and analysed, and one is selected as a basis for hardware architecture analysis.

The third section presents several architectures that have been proposed to solve the problems previously described. These architectures are analysed and one is selected as a basis for the hardware to be developed.

The fourth and final section describes a base architecture to process the inner product.

2.1 Floating-Point Arithmetic

Floating-Point arithmetic has been used since the early forties [4]. It was created as a means to have an approximation of real numbers supporting a dynamic range and precision with a limited number of bits. A floating-point number is represented as

\[
\text{significand} \times \text{base}^{\text{exponent}}. \tag{2.1}
\]

Over the years, the floating-point format has been expanded and improved by the IEEE who proposed the IEEE 754 standard in 1985. The standard defines: rounding rules, operations, exception handling and encodings. The standard describes two formats: binary and decimal. Herein only the binary format is considered. The IEEE 754 standard defines half (16 bits), single (32 bits), double (64 bits) and, more recently, quadruple precision (128 bits) formats.

2.1.1 IEEE 754 Format

A floating-point number represented using the IEEE 754 standard is organised in the following order: sign, exponent and significand [1]. The first bit is always the sign, the following bits are the exponent and the remaining bits are the significand (Figure 2.1). Table 2.1 shows some statistics about the standard formats.
The significand is represented as a fixed-point number and the exponent is represented using an offset (bias), i.e., the value stored in the exponent needs to be subtracted by the bias. Normalized numbers are represented as

\[ (-1)^{sign} \times \text{significand} \times 2^{exp}, \quad (2.2) \]

where \( k \) is the number of bits of the exponent, \( bias = 2^{k-1} - 1 \), and \( exp = \text{exponent} - bias \). The 1. denotes an hidden bit which is not stored. The standard also defines special values, for example: infinity, Not a Number (NaN) and subnormal numbers. Subnormal numbers allow the floating-point format to represent numbers smaller than \( 2^{exp_{\min}} \).

There are four rounding modes defined: round to nearest even (default), rounding to zero (truncation), round toward positive infinity and round toward negative infinity. The default rounding, round to nearest even will round the result to the nearest number—e.g. in 3.3 → 3 and 3.8 → 4, and when the result is halfway between two numbers it will be rounded to the closest even number, e.g. 3.5 rounds to 4. This is done to avoid inserting rounding bias in the computation (if the halfway result is constantly rounded up the computation will slowly drift upwards) [5].

### 2.1.2 Basic Floating-Point Operations

A floating-point addition/subtraction must follow these steps [6], [7]:

1. **Align Significands**: shift the significand \(|\text{Highest Exponent} - \text{Lowest Exponent}| \) number of times, this is required because two numbers can only be added if they have the same exponent;
2. **Add/Subtract**;
3. **Normalize**: when adding two floating point numbers whose significand is in the range \([1, 2]\), the result is in the range \([1, 4]\); so, normalization is the process of returning the number back to the range \([1, 2]\) and adjusting the exponent accordingly;
4. **Round the significand**;
5. **Normalize**: the second normalization stage is needed because after rounding nearest to even the result may become out of range \([1, 2]\);
6. **Store the data**.

**Example 2.1.1.** Consider the addition of the following two floating-point numbers, \(1.1001 \times 2^3\) and \(1.0101 \times 2^4\), and the final result is rounded to positive infinity. Following the steps described:

1. **Align Significands**: the number with the smallest exponent \((1.0101 \times 2^4)\) will shift

\[ |\text{Highest Exponent} - \text{Lowest Exponent}| = |3 - 1| = 2 \text{ bits.} \]
Continued

Thus, \(1.1010 \times 2^1\) after shifting is \(0.011010 \times 2^3\);

2. Add/Subtract:

\[
1.1001 \times 2^3 + 0.011010 \times 2^3 = 1.111110 \times 2^3;
\]

3. Normalize: The result does not need to be normalized because it is already in the \([1, 2]\) range;

4. Round the significand: \(10.0000 \times 2^4\);

5. Normalize: \(1.0000 \times 2^4\);

6. Store the data.

Figure 2.2 shows the block diagram for a standard floating-point addition unit.

![Block Diagram of a Standard Addition Floating-Point Unit](image)

Figure 2.2: Block Diagram of a Standard Addition Floating-Point Unit, based in [7]

The multiplication/division steps to perform the operation are simpler, they are: addition/subtraction of the exponents, multiplication/division of the significands, normalization, round, process signal and store the data. The multiplication/division operations are simpler because the alignment stage is not needed and the signal can be simply processed by a XOR gate.

2.1.2.1 Fused Multiply-Add

In 2008, a standard fused multiplier-add operation was defined as \((A \times B) + C\), which should be processed without a range and precision limit and with only one final rounding [1]. The steps necessary to complete this operation are the following:

1. Multiply significands
2. Add Exponents
3. Align the result of the multiplication without rounding or truncating with \(C\): this procedure works exactly as in the addition/subtraction operation;
4. Add the result of the multiplication to \(C\) using the adder with the full length of the multiplication result;
5. Normalize;
6. Round;
7. Store the data.

Figure 2.3 shows the block diagram for a standard fused multiply-add unit. Presently, Graphical Processing Units (GPUs) and GPPs implement this operation in hardware, compliant with this specification \cite{8}, \cite{9}.

![Block Diagram of a Standard Floating-Point Fused Multiplier-Adder Unit, based in \cite{10}](image)

**2.2 Inner Product**

The inner product is defined by a multiplication followed by an accumulation of all resulting terms:

\[
    s = \sum_{i=0}^{n} x_i \times y_i.  \tag{2.3}
\]

As the inner product is a single operation it should be done in a single fused operation. However, general processors in common implementations split it into two operations: one multiplication and one accumulation. Therefore, the result of the multiplication needs to be rounded, truncated and stored before the accumulation is performed, which may affect the desired accuracy of the operation.

The accumulation operation has difficulties performing the sum of a large number with a small one, because the numbers must be aligned before the accumulation takes place. For example, \(1.15 \times 10^{20} + 1.99 \times 10^{-5}\) requires a shift of 25 digits resulting in \(1.15 \times 10^{20} + 0.00 \times 10^{20} = 1.15 \times 10^{20}\). Most implementations assume that such a big difference in alignment can be safely inferred as an addition of zero. This assumption can be dangerous when adding the same small number multiple times to a large number (e.g. \(10^3 + 10^{-6} + 10^{-6} + \cdots + 10^{-6}\)).

Modern processors that use Single Instruction, Multiple Data (SIMD) instructions \cite{11}–\cite{13} also apply the same sequence of operations, multiplication followed by addition, but in a vector of scalars instead of per scalar. Therefore, they have the same accuracy issues. All in all, the only advantage of using SIMD instructions to process the inner product is speed.

The standard fused multiply-add operation can only maintain accuracy for one addition, while for improved accumulation accuracy solving the full accuracy problem requires maintaining accuracy for \(n\) accumulations, not just one.
2.2.1 Accumulation Solutions

There are many published solutions about floating-point accumulation [3], [4], [14]–[31]. The goal is always to compute the result with minimal deviation from the exact result where the deviation should not depend on the number of addends [19]. Two main alternatives can be identified: Addition with Remainder and Long Accumulator. Since these two concepts are the most prevalent and ubiquitous they are detailed in this section.

2.2.1.1 Addition with Remainder

Addition with Remainder designates a set of algorithms that store the small errors in each computation (remainder) and then correct the result at a later time.

The first of these algorithms was presented in [17] by Pichat (algorithm 1). Pichat proposed to store each remainder in an array \( x' \) where the first element is an approximation of the final result \( x'_1 \)—the standard accumulation with no corrections—and the remaining elements are the errors of the accumulation for that term [21]. In the first iteration \( x'_1 \) holds the result of \( x_1 + x_2 \); in the second iteration \( x'_1 \) holds the result of the accumulation between the first accumulation result and \( x_3 \), i.e., \( x'_1 + x_3 \) or \( (x_1 + x_2) + x_3 \); etc. The final result of \( x'_1 \) can be written as \( \sum_{i=0}^{n} x_i \). Each accumulation error is processed and stored in the following manner: \( x'_2 \) holds the summation error of \( x_1 + x_2 \); \( x'_3 \) holds the summation of error of \( x'_1 + x_3 \) where \( x'_1 = x_1 + x_2 \); etc. The final result is obtained by subtracting the remainders to \( x'_1 \), i.e., \( x'_1 - \sum_{i=2}^{n} x'_i \).

The following algorithm shows its pseudo-code:

```
Data: \( x'_1 = x_1 \)
Result: Sum \( s \)
for \( i = 2 \) to \( n \) do
    \( t = x_i + x'_1 \);
    \( x'_2 = (t - x'_1) - x_i \);
    \( x'_1 = t \);
end
```

**Algorithm 1:** Pichat’s algorithm

where \( t \) is the local sum, \( x \) is the input vector and \( x' \) is the remainder vector. This algorithm can achieve a better accuracy because the computation of remainders will generate smaller errors.

---

**Example 2.2.1.** Consider a floating-point arithmetic unit which supports 8 bit floating-point numbers (4 bits for the significand and 3 bits for the exponent), which is used to process the accumulation of the following vector \( x = [2.625, -10.5, 0.5, 5.0] \). Pichat’s algorithm is used to process the accumulation (algorithm 1). The following list enumerates the steps taken by the algorithm to obtain the second result of the summation:

1. Write \( x' = [2.625, 0, 0, 0] \);
2. Process \( t = -10.5 + 2.625 = -7.75 \);
3. Process the summation error \( x'_2 = (-7.75 - 2.625) + 10.5 = 0.125 \). The updated summation vector is \( x' = [2.625, 0.125, 0, 0] \);
4. Store the summation result in the vector: \( x' = [-7.75, 0.125, 0, 0] \)

In step two, there is a rounding error because the 8 bit floating-point format does not have enough precision to store the result of the addition. The correct result is \(-7.875\), but the stored value is \(-7.75\).

The remaining iterations of the algorithm have no error, thus they are not described. The final
remainder vector is \( x' = [-2.25, 0.125, 0, 0] \). Performing a final error subtraction of the vector yields the correct result \(-2.375\).

Furthermore, a new vector of remainders can be created \( (x'') \) with the same procedure as the one described above. For example, \( x''_2 \) holds the summation error of \( x'_1 + x'_2 \), etc. This process can be repeated according to the desired accuracy requirements [21].

Kahan also presented a similar algorithm in [22] (algorithm 2). The difference lies in the way the remainder variable is stored and merged into the final sum [23]. In Pichat’s case the remainders are stored in a vector that can be fed back into the loop (replace the previous \( x_i \) with the remainder vector, \( x'_i \)). On the other hand, Kahan’s algorithm stores the remainder in a single variable (and not in a vector). More iterations of the loop do not yield a better result. This slight difference results in different tradeoffs in accuracy/resources consumed for each algorithm. The second requires less resources but has a slight accuracy penalty.

The following algorithm shows Kahan’s addition with remainder pseudo-code:

```
Data: s = x_1; c = 0
Result: Sum s
for i = 2 to n do
    y = x_k - c;
    t = s + y;
    c = (t - s) - y;
    s = t
end
```

Algorithm 2: Kahan’s Algorithm [22], [23], [25]

where \( t \) and \( y \) are the local sums, \( x \) is the input vector, \( c \) is the remainder term, and \( s \) is the sum.

Example 2.2.2. Using the same data vector and floating-point format as Example 2.2.1, Kahan’s algorithm is used to process the accumulation (algorithm 2). The following list enumerates the steps taken by the algorithm to obtain the second result of the summation:

1. Write \( s = 2.625 \);
2. Process \( y = -10.5 - 0 = -10.5 \);
3. Process \( t = 2.625 - 10.5 = -7.75 \);
4. Process the summation error \( c = (-7.75 - 2.625) + 10.5 = 0.125 \);
5. Store the processed summation \( s = -7.75 \).

In step four, there is a rounding error because the 8 bit floating-point format does not have enough precision to store the result of the addition. The correct result is \(-7.875\), but the stored value is \(-7.75\). In the next iteration of the loop, \( i = 3 \), the lost portion \( c \) in the previous iteration will be subtracted to \( x_k \).

The remaining iterations do not yield an accuracy error, and thus are not described. Performing the remaining iterations will yield a final result of \(-2.375\).

Comparing Pichat’s and Kahan’s examples shows that Pichat’s requires a final accumulation of the remainders to obtain the final result and perform the corrections, whereas Kahan’s performs the correction every iteration per summand \( (y = x_k - c) \). The latter causes Kahan’s algorithm to lose accuracy throughout the processing.
2.2.1.1 Pichat and Kahan Accuracy Tests  Both Kahan’s \footnote{algorithm 2} and Pichat’s \footnote{algorithm 1} algorithms were tested with problem sets that, traditionally, floating-point arithmetic has difficulty dealing with, \cite{2}. Seven tests were devised:

- \(x_i\) is the \(i\)th term in the Taylor series expansion for \(e^{-2\pi}\), for 64 terms. This vector tests rounding errors throughout the accumulation;
- \(x_1 = x_2 = \cdots = x_{2047} = 1.0, x_{2048} = x_{2049} = 1.0 \times 10^{-18}, x_{2050} = x_{2051} = \cdots = x_{4096} = -1.0\) (Heavy cancellation). This set tests whether small numbers are inferred as zero when added to a large number;
- \(x_i\) equally spaced in \([1, 2]\) for \(n = 4096\). Easy set to process because all numbers are represented exactly by the IEEE 754 single and double precision formats;
- random numbers from Normal(0,1) distribution, for \(n = 4096\). This set serves the same purpose as the Heavy Cancellation set, but is more difficult to process because the numbers vary in sign;
- \(x_i = 1/i^2\), for \(n = 4096\); Used to be consistent with Higham’s work \cite{32}.
- \(x_i = \pm 10^{p_i}\) where \(p_i\) is chosen from Normal(0,35) distribution randomly but with values greater than 35 in magnitude replaced by +35 or -35. This is similar to the Heavy Cancellation test and the Normal(0,1) test, but with large differences in magnitude between numbers;
- \(x_1 = 1.0, x_2 = x_3 = \cdots = x_{10^{n+1}} = 1.0 \times 10^{-16}\) (One Large, Many Small). This set was selected because it gives an error even in double-precision.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & \(e^{-2\pi}\) & Heavy Canc & equal & normal & \(\sum(\frac{1}{n})\) & random \\
\hline
Kahan & 0.0013 & 1.0 & 0 & 0.00000001 & 0 & 0.0000002 \(10^{-35}\) \\
Pichat & 0 & 0 & 0 & 0 & 0 & 0 \(10^{-35}\) \(35\) \\
\hline
\end{tabular}
\caption{Relative error obtained for the tests data set, from \cite{2}}
\end{table}

As expected Pichat’s method achieves a better result at a cost of requiring a memory the size of the vector, due to storing each remainder separately.

2.2.1.1.2 Kahan’s Algorithm Improvements  Two new algorithms were proposed to improve Kahan’s algorithm accuracy. To enhance the algorithm, a condition was added to further improve it in cases where the sums magnitude is greater than the summand \cite{22, 24, 26}. The improved algorithm is in \footnote{algorithm 3}. The remainder is only added at the end of the loop and not in between each iteration. Depending on the resource usage and accuracy requirements the remainder can be added inside the loop (\(s = t\) is now \(s = t + c; c = 0\)).

Data: \(s = x_1; c = 0\)
Result: Sum \(s\)
\begin{algorithm}
\For {\(i = 2\) to \(n\)} {
\hspace{1cm} \(t = s + x_i;\)
\If {\(\text{abs}(s) \geq \text{abs}(x_i)\)} {
\hspace{1cm} \(c = c + ((s - t) + x_i)\)
\Else {
\hspace{1cm} \(c = c + ((x_i - t) + s)\)
\}
\}
\hspace{1cm} \(s = t\)
}\hspace{1cm} \(s = s + c\)
\end{algorithm}

\textbf{Algorithm 3:} Kahan-Babuška Algorithm

If the remainder is only added at the end, the term may have the same accumulation problem as \(s\).
Fixing the issue requires another remainder term to correct the previous remainder. This can be done until the desired accuracy is reached at a higher resource usage. A Kahan-Babuška algorithm with two remainder terms—or second order—is in algorithm 4.

**Data:** \( s = 0; c_0 = 0; c_1 = 0; c_2 = 0; c_3 = 0 \)

**Result:** Sum \( s \)

**for** \( i = 1 \) to \( n \) **do**

\( t = s + x_i; \)

if \( \text{abs}(s) \geq \text{abs}(x_i) \) **then**

\( c_0 = (s - t) + x_i \)

else

\( c_0 = (x_i - t) + s \)

end

\( s = t; \)

\( t = c_1 + c_0; \)

if \( \text{abs}(c_1) \geq \text{abs}(c_0) \) **then**

\( c_2 = (c_1 - t) + c_0 \)

else

\( c_2 = (c_0 - t) + c_1 \)

end

\( c_1 = t; \)

\( c_3 = c_3 + c_2 \)

end

\( s = s + c_1 + c_3 \)

**Algorithm 4:** Second Order Kahan-Babuška Algorithm

The addition of another remainder term adds an extra four floating-point operations to the loop. If the inputs are ordered, the accuracy of the result improves [27], [29]. Then the largest error is associated with the smallest input [29].

Furthermore, the partial sums should be also inserted in the ordered list to minimise errors. Thus, the accumulation should operate over a sorted list in the following way [20], [25], [29]:

1. Choose two numbers from the list and add them. Reinsert the result back into the list keeping the order;
2. Repeat until there is only one item in the list. That last element will be \( s \).

However, an ordering is required, which at best has a complexity of \( O(n \log n) \), and therefore its run time and resource usage may be too high [21], [26].

### 2.2.1.2 Long Accumulator

The **Long Accumulator (LA)** algorithm computes an accumulation with full accuracy, using full-width fixed-point accumulations [3].

The result of the multiplication of two operands with \( l \) bit significands and exponents \( e_1 \) and \( e_2 \), has a \( 2l \) bit significand and exponent \( e_1 + e_2 \). Thus, the largest possible multiplication result will have a \( 2e_{\text{max}} \) exponent, while the smallest possible value will have a \( 2e_{\text{min}} \) exponent. To allow for the full operand size and including \( k \) bits to guard against possible overflows, the fixed-point accumulation must have a size [3] (Figure 2.4):

\[
L = k + 2e_{\text{max}} + 2l + 2|e_{\text{min}}|.
\]  

(2.4)

The \( k \) digits added at the left end of the accumulator can tolerate \( 2^k \) overflows [3]. The large size of the accumulator is the main disadvantage of this approach.
Example 2.2.3. For the IEEE 754 single-precision format and \( k = 86 \) bits the accumulator must have a size of \( L = 86 + 2 \times 127 + 2 \times 24 + 2 \times 126 = 640 \) bits; for double precision with \( k = 92 \) bits the size is \( L = 4288 \) bits.

| \( k \) | \( 2e_{\text{max}} \) | \( 21 \) | \( 2|e_{\text{min}}| \) |
|-------|----------------|------|--------------|

Figure 2.4: LA register

A direct implementation of the long accumulator would use a long shifter and a long adder, as pictured in Figure 2.5. Performing the accumulation with a full width ripple-carry adder is unpractical as a carry propagation across it would be too expensive. However, the individual accumulations to the register do not need to access all of its bits at once. Therefore, to achieve an acceptable performance these long operations may be split into smaller ones.

Instead of doing the operation with the entire register, an adder with the size of the summand may be used, turning the long adder into a small one. The register segment with which the accumulation is computed is selected according to the exponent value, e.g. \( x \times 2^7 \) will select the segments \([128, 128 + \text{size of segment}]\). In the end, the results of every register segment are concatenated forming the final result. So, the large register is split in multiple and smaller registers. This type of addition is generally referred to as segmented accumulator. It is based on the premise that there should be an accumulator for every summand range (Figure 2.6).

Since the register is split into small registers the carries still need to be propagated from one segment to another. The propagation may be done as follows: when the register capacity is exceeded (overflow),
the carry is propagated to the next segment register [3]. This carry does not need to be absorbed immediately in the next cycle. The carries can be stored for a certain segment and added at a later time (carry save).

![Segmented Accumulator Diagram](attachment:diagram.png)

Figure 2.6: Segmented Accumulator with 64 bit segments

### 2.2.2 Hardware Implementation Analysis

Pichat’s [algorithm 1](#) accumulation can be implemented using a memory to store the remainder vector and three floating-point adders/subtractors. Kahan's [algorithm 2](#) accumulation can be implemented using four floating-point adders/subtractors and two registers, one more floating-point operation than Pichat's but with no memory requirements. The extra floating-point operation is less significant because the biggest difference is the need of a memory the size of the vector to store each remainder. New algorithms based on Kahan’s initial proposal were presented to further reduce arithmetic errors and increase its accuracy. However, its improved version [algorithm 4](#) requires an ordering, implying a memory is necessary to keep all operands in order, and two comparisons, which would increase the hardware usage [25]. This increase is not justified when Pichat’s algorithm reaches the same accuracy without the ordering and comparisons.

The long accumulator main problem is the carry propagation through the large adder, which can be solved when using segmented accesses. This adds more complexity to the design, specifically the carry propagation, but it speeds up the system since each addition to the accumulator can be, ideally, independent and done in parallel. Yet this requires a specific carry mechanism.

Each algorithm addresses different requirements. Therefore, the choice between the three is dependent on the requirements of the system. Kahan’s should be selected when there is a low resource requirement and Pichat’s should be selected when there is memory present in the system that fits the size of the input vectors. The long accumulator should be selected whenever full accuracy is required. In this project full accuracy is targeted, hence the long accumulator is selected. Table 2.3 summarises the differences between all algorithms described.
Table 2.3: Comparison between all algorithms described using resources, critical path and accuracy

<table>
<thead>
<tr>
<th></th>
<th>Pichat</th>
<th>Kahan</th>
<th>Kahan-Babuška (2nd Order)</th>
<th>Long Accumulator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resources</strong></td>
<td>3 FP Units</td>
<td>4 FP Units</td>
<td>4 FP Units</td>
<td>Large Shifter and Accumulator</td>
</tr>
<tr>
<td></td>
<td>1 Memory</td>
<td></td>
<td>Ordering Unit</td>
<td></td>
</tr>
<tr>
<td><strong>Critical Path</strong></td>
<td>FP Unit</td>
<td>FP Unit</td>
<td>FP Unit</td>
<td>Accumulator</td>
</tr>
<tr>
<td><strong>Accuracy</strong></td>
<td>Very Good</td>
<td>Good</td>
<td>Very Good</td>
<td>Full</td>
</tr>
</tbody>
</table>

2.3 Long Accumulator Architectures

In this section two long accumulator architectures are described. One is the basic version which uses a big shifter and a big adder. The other uses the segmented adder technique to avoid the large critical path created by the big adder.

The first long accumulator architectures were proposed by Kulisch in [3] for both the standard long accumulator and segmented accumulator. Dinechin recently proposed a standard long accumulator architecture for FPGAs that can be configured for a specific application’s precision [33].

In the first section Dinechin’s proposal will be described followed by Kulisch’s segmented accumulator architectures. The final section analyses both architectures.

2.3.1 Non-Segmented Long Accumulator

The long accumulator architecture proposed in [33] by Dinechin allows the configuration of the width of the shifter and the adder for a specific application.

![Figure 2.7: Proposed accumulator in [33]](image)

The block diagram of this architecture is shown in Figure 2.7, where \( w_S \) is the size of the word after the shift and \( w_A \) is the width of the adder. To configure the architecture for the specific application three variables need to be set: the maximum expected value of the summand’s exponent (\( \text{MaxMSB}_X \)), the amount of bits for the integer part of the result (\( \text{MSB}_A \)) and the amount of bits for the fractional part of the result (\( \text{LSB}_A \)). \( \text{MSB}_A \) determines the maximum value for the accumulator (\( 2^{\text{MSB}_A} \)) and
$\text{LSB}_A$ determines the final precision of the result. Therefore, the size of the accumulator is $w_A = \text{MSB}_A - \text{LSB}_A + 1$, the number of bits shifted is $n\text{bits}$\_shift = $\text{MaxMSB}_X - \text{exp}$ (where $\text{exp}$ is the summand’s exponent) and the size of word after the shift is $w_S = \text{MaxMSB}_X - \text{LSB}_A + 1$. Increasing the amount of tolerable overflows ($\text{MSB}_A$) requires the width of the adder ($w_A$) to increase but, the size of the word after the shift ($w_S$) does not increase.

Example 2.3.1. Consider $\text{MSB}_A = 16$, $\text{LSB}_A = -15$ and $\text{MaxMSB}_X = 10$. Then, the width of the adder is $w_A = 16 + 15 + 1 = 32$ bits, and the shift will return a word of size $w_S = 10 + 15 + 1 = 26$ bits.

To reduce the critical path of the accumulator a register can be inserted every $k$ bits [34]. As a result the critical path is bound by a $k$ bit addition (Figure 2.8).

![Figure 2.8: k bit critical path](image)

2.3.2 Segmented Long Accumulator

Kulisch in [3] proposed to divide the long accumulator in several registers (segments), such that in each addition step only a few number of segments are involved in the accumulation. By selecting the size $y$ of the registers as a power of two (and greater than two), the segment selection and the number of bits to be shifted can be directly obtained from the summand’s exponent. The number of segments involved (and therefore the number of adders) in each accumulation directly depends on the size of the operand after shifting (which must be padded to a multiple of $y$).

Example 2.3.2. The multiplication of two single-precision (32 bits) operands generates a result with an exponent of 9 bits and a significand of 48 bits. Considering a 640 bits long accumulator, from 2.2.1.2 and assuming $y = 32$ bits there are $640/32 = 20$ segments.

The shift of the summand and segment selection is performed using the Least Significant Bits (LSBs) and Most Significant Bits (MSBs) of the exponent, respectively. If each segment is 32 bits then the alignment shift requires

$$\text{shift} = \log_2 32 = 5 \text{ bits},$$

which leaves $\text{select} = \text{exp} - \text{shift} = 9 - 5 = 4$ bits to select the registers. However, four bits cannot select twenty registers. The most significant registers that can be selected are $R_{15}$, $R_{16}$ and $R_{17}$, which means two registers will never be selected, $R_{18}$ and $R_{19}$. From 2.2.1.2 $k$ was set to 86 bits to achieve $L = 640$ bits. Thus, two registers will never be selected because they are used to prevent overflows.

Having a 5 bit shift results in a summand after shift of $\text{Word After Shift Size} = 2l + y - 1 =$
Continued

48 + 32 − 1 = 79 bits. Therefore the number of adders in the segmented accumulator is

\[ N_{Adders} = \lceil \frac{\text{Word After Shift Size}}{y} \rceil = \lceil \frac{79}{32} \rceil = 3 \text{ adders} \]

of 32 bit width. After the shift, the summand needs to be padded to fit the three adders. The padding required is \( padding = N_{Adders} \times y - \text{Word After Shift Size} = 3 \times 32 - 79 = 17 \) bit.

Concluding, with 32 bit registers: twenty registers, three 32 bit adders and a five bit left shift are required, as shown in Figure 2.9.

![Figure 2.9: Single-Precision Segmented Accumulator with 32 bit registers](image)

Kulisch also proposed an alternative segmented accumulator using a memory instead of a register file. A block diagram with 32 bit words and 10k addresses is shown in Figure 2.10. Using a memory to store the segments is more area efficient, but the carry resolution logic takes more cycles because the contents of the memory are not accessible all in the same cycle.
2.4 Base Inner Product Core Architecture

An inner product unit consists of three basic stages: multiplication, accumulation, and conversion of the accumulation result to IEEE 754 floating-point. A base non-segmented architecture of the inner product core can directly use the described long accumulation in subsection 2.3.1.

The multiplication and accumulation stages are shown in Figure 2.11 and the final conversion stage is shown in Figure 2.12. The inputs are split into three components: sign, exponent and significand. The multiplier multiplies both significands and the first adder adds both exponents. The result of the multiplication is then aligned (shifter), using the result of the exponent addition, and expanded to fit the width of the adder. Then, the result of the shift is accumulated and the contents of the register are stored in two’s complement representation.

After the accumulation is completed, the result is converted back and truncated into floating-point. The two’s complement block converts the contents of the accumulator from two’s complement to sign-magnitude. The Lead Zero Detector (LZD) (implemented as in [35]) counts the number of zeros until the first bit is set to one. The result of the accumulation is shifted according to the result of the LZD so that only the significant digits are used in the final floating-point result. The exponent is computed in the following manner: subtracting the size of the register by the amount of zeros given by the LZD yields the significant bits, subtracting by \(2l - 1\) removes the number of bits until the hidden bit, and to bias encode the exponent another subtraction of \(bias\) suffices. Therefore, the final biased exponent is

\[
\text{Final Biased Exp} = L - (2l - 1) - \text{bias} - \text{LZD} - 1.
\]  

Then, the significand is truncated and the final floating-point number is built from the result of the normalization, the result of the exponent computation and the sign.
2.5 Conclusion

In this chapter the state-of-the-art on inner product computation was analysed. The inner product analysis identified the difficulties in keeping accuracy when accumulating $n$ numbers. Two solutions were analysed: addition with remainder and long accumulator. Of the two solutions only the long accumulator supports full accuracy, and so it was chosen as a basis for the architecture analysis.

A non-segmented long accumulator architecture can use a carry-save mechanism to split the critical path into smaller $k$ bit paths, such that there is a dedicated adder for every $k$ bits. However, having an adder for every $k$ bits is not always needed as it is rare for a single summand to use the entire register.

The proposed segmented accumulator only loads into the adders the segments affected by the summand. This allows the adder to focus on the local sum while keeping the entire processing accurate. There is a direct relationship between the size of the segment registers and the size of the shift. The smaller the size of the registers is, the smaller the alignment shift is. However, the amount of adders needed increases, which increases the size of the register file decoder. As a result, a compromise needs to be made between resources usage and size of critical path. The segmented adder requires special carry logic as each local sum is processed independently. The carries are stored adjacent to the registers that will absorb them. At the end of the computation, extra cycles are required to propagate the remaining carries in each register.

There are tradeoffs between the selection of the long accumulator architectures. The non-segmented architecture is straightforward to implement, whereas the segmented accumulator is more difficult. Furthermore, the amount of resources required by the segmented accumulator is greater than the standard long accumulator, but has a shorter critical path. Therefore, the selection of the long accumulator architecture is dependent on the design requirements.
A base inner product architecture is built using a conventional multiplication stage, followed by the accumulator stage and by a final conversion stage. In the following chapters, two efficient segmented accumulators are proposed with different carry propagation strategies, and a resource-saving final conversion stage is proposed.
Chapter 3

Segmented Accumulator Stalling Architecture

This chapter proposes a segmented accumulator architecture using a register file, which stalls the accumulation process whenever remain carries to propagate. The full-width adder and shifter are replaced by smaller adders, and by a segment selector, respectively. Thus, this chapter focuses on improving the execution stage of the architecture.

Using selected local (segment) additions, sets constraints on how and when carries should propagate. To solve the constraints imposed by the segmented architecture, a Carry Processing Block (CPB) is introduced to handle all carry propagation tasks. The first version of the CPB has a dedicated adder per register. Its second optimised version requires only one adder to propagate carries.

3.1 Basic Stalling Architecture

A non-segmented accumulator is composed of a full-width adder and a large shifter, as shown by Figure 3.1.

The adaptation of the non-segmented accumulator to a segmented accumulator replaces the full width adder by several segment sized adders, and the large alignment shifter by a pre-shifter and register selector—the design follows the guidelines from subsection 2.3.2. The basic architecture is in Figure 3.2.

Using a segmented accumulator does not guarantee the propagation of each register's carry because it only operates on a subset of registers. Hence a dedicated carry propagation block is required, the CPB.
to propagate all carries generated by the segmented accumulator to all registers.

![Segmented Accumulator with three adders and a CPB](image)

The CPB propagates carries whenever there is at least one carry generated by the segmented accumulator. Then, the accumulator process stalls while the CPB processes all carries, and the segments and carries are updated with the results of the CPB. The number of cycles required by the CPB to propagate all carries is equal to the number of segments. At the end of the carry processing the contents of the carry registers are reset.

**Example 3.1.1.** This example describes the steps required to design a segmented accumulator architecture using a register file, which stalls in every carry propagation. The architecture supports the IEEE 754 single-precision format and uses 16-bit segments.

1. Calculate the size of $L$ (subsubsection 2.2.1.2). For 32 bit floating-point numbers $\Rightarrow L = k + 554$ bits (Equation 2.4). Using 46 guard bits ($k$) $\Rightarrow L = 600$ bits.

2. Select the segment size $y$, calculate the number of registers in the register file, calculate the portion of the exponent used to perform the shift and the selection of the registers in the register file (subsection 2.3.2). $y$ is 16 bits $\Rightarrow$ Registers = 38, pre-shift = 4 bits, and select = 5 bits.

3. If the architecture uses 4 bits to shift, the size of the summand after the shift is 48 + 15 = 62 bits. Since the size of the summand is not a multiple of the segment size, padding is required. The padding size is $\lceil 62/16 \rceil \times 16 - 62 = 2$ bits. Therefore, the minimum number of adders is $\lceil 64/16 \rceil = 4$.

### 3.2 Carry Processing Block Architecture

Figure 3.3 shows a simplified block diagram of the CPB for the first three stages, where all_register_out and all_carries_out are all the register segments and carries from the register file, respectively. The
first register does not require an adder because it will never absorb or generate a carry during the carry processing state.

When the CPB is operating over the registers, the architecture cannot be accumulating any other summands, otherwise there are two results writing to the same register. As a result, the accumulator must stall while carries are propagating, i.e., while the CPB is operating over the register file. The accumulator stalls when any of the segmented accumulators generate a carry out different than zero—signal is_one. After the processor stalls the CPB can start operating over the register file, in which case the register file will write the results of the CPB. This state is represented by a signal called processing_carry. When processing_carry is one, the register file writes the results of the CPB to its registers, otherwise the results of the segmented accumulator are written depending on the write_enable flag.

The CPB requires a stop condition, so that the segmented accumulator can resume processing. Since the CPB uses carry save, the carry propagation requires a cycle per register, starting at the least significant one. Therefore, the number of cycles required by the CPB to guarantee that every carry is propagated, is equal to the number of registers. Hence, the stop condition for the CPB occurs when all registers have their carries propagated. Herein, the stop condition is implemented using a counter, however when the number of registers is not a power of two a comparator is required. Instead of using a comparator, the counter register is initialised with a specific value, which when added with the number of registers will be a power of two. Using this initial value for the CPB counter allows the stop condition to be an overflow check of the counter. Thus, the CPB counter is a register of size \( \lceil \log_2 (\text{Registers}) \rceil \) set initially to \( 2^{\text{size} - \text{Registers}} \), which increments when processing_carry is one. When processing_carry is zero the counter is reset to its initial value. If the last register generates a carry out and the CPB counter overflows, the system overflows.

Using the previous definition of the CPB counter, the processing_carry signal is formally defined as follows:

- Store the value of is_one delayed by one clock cycle when the CPB counter has its initial value;
- When the CPB counter overflows, processing_carry is reset;
- Otherwise is one.

When writing the final results of the CPB to the register file, all carry registers must be reset as to avoid the re-propagation of previously processed carries.

**Example 3.2.1.** Continuing Example 3.1.1. The CPB uses 38 adders. Each adder has a width equal to the segment size, 16 bits.

An architecture with a register file containing 38 registers has a CPB counter with the following...
characteristics:

\[
\begin{align*}
\text{size} &= \lceil \log_2 (38) \rceil = 6; \\
\text{initial value} &= 2^6 - 38 = 64 - 38 = 26.
\end{align*}
\]

The CPB counter needs 6 bits and will reach the last register when its value exceeds 63 (overflow).

### 3.3 Carry Processing Optimisation

The carry processing mechanism always starts at the least significant register to avoid extra carry generations. Since the least significant selected register may generate a carry in for the next most significant register, having a dedicated adder for each register is redundant. Thus, the CPB only requires one adder. To store the carry out from the previous propagation, a register is used. The block diagram of the new architecture is in Figure 3.4, where the select_register signal is the result of the stall counter subtracted by its initial value.

Furthermore, it is not necessary to perform a carry propagation for every register in the register file, because most of them will not have a carry to propagate. The CPB knows the starting register from the generated segmented accumulator carries, and therefore can start propagating carries in the first register which will consume the first carry. The start register is the least significant register which generated a carry plus one.
Example 3.3.1. Consider a segmented accumulator with three adders, which selects register seven, eight and nine. The accumulation of the summand with register seven generates a carry out. The CPB must start propagating carries at register eight, since this is the first register which will consume the carry-in. For a segmented accumulator that has three adders (three carry outs) Table 3.1 shows all the possible start positions for all carries generated.

Table 3.1: Start Position relative to register selection for each combination of carries

<table>
<thead>
<tr>
<th>(c_{02})</th>
<th>(c_{01})</th>
<th>(c_{00})</th>
<th>Relative Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(X)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When the adders do not generate carries, no propagation occurs. When the first adder generates a carry out, the CPB will start at register eight. When register eight generates a carry out, and register seven does not, the CPB will start at register nine. When register nine generates a carry, and registers eight and seven do not, the carry processing block will start at register ten.

Generally, the relative distance to the start segment can be obtained by Equation 3.1 where \(\text{carries} \in [0, 2^{\text{NAdders}}]\).

\[
\text{Relative Distance}(\text{carries}) = \begin{cases} 
X & \text{if carries} = 0 \\
i + 1 & i = \max(x \in [0, \text{NAdders}]: \text{carries} \mod 2^x = 0) 
\end{cases}
\] (3.1)

Similarly the minimum number of cycles required by CPB to propagate a given carry sequence is predictable. If more cycles then the predicted minimum are necessary, then the CPB propagates carries until one propagation does not generate carry out.

The prediction uses the carries out generated by the segmented accumulator. If only one carry is generated then the minimum number of cycles required is one. If two carries are one in sequential registers, the minimum number of cycles is two. Often there are carries generated between non-sequential registers. In such a case the minimum number of cycles required is the difference between the most significant carry generated and the least significant carry generated plus one. The extra cycles described ensure the registers between the carries generations absorb the possibility of another carry generation during the propagate operation.

Example 3.3.2. Suppose an architecture with the same characteristics as Example 3.3.1 (three adders in the segmented accumulator). Assume that the summand selects registers seven, eight and nine. The carry sequence 011\(_2\) requires at least two cycles to propagate, for registers eight and nine, respectively. The carry sequence 101\(_2\) requires at least three cycles to propagate, for registers eight, nine and ten, respectively. In the previous carry sequence, the extra cycle
used to propagate carries between registers eight and ten, is required because register nine may generate another carry to register ten. Table 3.2 shows the minimum number of cycles for each combination of generated carries.

Table 3.2: Minimum Number of Cycles for each carry combination with three carries

<table>
<thead>
<tr>
<th>$c_{0_2}$</th>
<th>$c_{0_1}$</th>
<th>$c_{0_0}$</th>
<th>Min. N. of Cycles</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>9/8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10/9/8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10/9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>10/9/8</td>
</tr>
</tbody>
</table>

Example 3.3.3. Suppose an architecture using four adders in the segmented accumulator (four carries out). Assume that the summand selects registers seven, eight, nine and ten. The carry sequence $0101_2$ requires at least three cycles to propagate, for registers eight, nine and ten, respectively. The carry sequence $1010_2$ requires three cycles to propagate, for registers nine, ten and eleven, respectively. The minimum number of cycles for an architecture with four adders (four carries) in the segmented accumulator is in Table 3.3.

Table 3.3: Minimum Number of Cycles for each carry combination with four carries

<table>
<thead>
<tr>
<th>$c_{0_2}$</th>
<th>$c_{0_1}$</th>
<th>$c_{0_0}$</th>
<th>$c_{0_2}$</th>
<th>Min. N. of Cycles</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>9/8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>10/9/8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>10/9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11/10/9/8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>11/10/9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>11/10/9/8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>11/10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>11/10/9/8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>11/10/9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>11/10/9/8</td>
</tr>
</tbody>
</table>
Generally, the minimum number of cycles can be obtained by Equation 3.2, where carries \( \in [0, 2^{N\text{Adders}}] \).

\[
\text{Min. Cycles(carry)} = \begin{cases} 
X & \text{if carries = 0} \\
\lceil \log_2 (\text{carries}) \rceil - \text{Rel. Dist.}(\text{carries}) + 1 & \text{if carries is even} \\
\lceil \log_2 (\text{carries}) \rceil & \text{if carries is odd}
\end{cases}
\tag{3.2}
\]

The stop propagating carries condition is defined as, in order of priority:
1. Segment selector overflow;
2. The CPB does not generate a carry and the minimum number of predicted cycles is over;

### 3.4 Architecture Analysis

The number of adders is inversely proportional to the segment size. Therefore, the proposed segmented architecture can be designed with different requirements in mind.

In the execution stage, the critical path goes through the pre-shifter and adders to the register file (Figure 3.4). The width of the adder is equal to the segment size \( y \), and the pre-shift is logarithmically proportional to the segment size \( y \). Hence, the smaller the segment size \( y \) is, the shorter the critical path is. From the segmented accumulator definition the segment size \( y \) must be a power of two and greater or equal than two. Thus, the smallest critical path possible is for two-bit adders and a shift of one bit. Having two-bit adders and a one bit shift requires:
- \( \lceil L/2 \rceil \) registers in the register file;
- \( \exp - 1 \) bits for the register selection;
- and \( \lceil (2l + 2 - 1)/2 \rceil \) adders,

where \( L \) is the full size of the accumulator, \( \exp \) is the amount of bits used by the exponent after the multiplication stage and \( l \) is the significand size (subsection 2.3.2).

**Example 3.4.1.** For a segmented accumulator which supports the IEEE 754 single-precision format to have the best critical path possible the following requirements must be met. The size of \( L \) is 554 + \( k \) bits, with \( k = 46 \) bits \( \Rightarrow L = 600 \) bits. The smallest segment size \( y \) is two bits, which has a shift size of one bit. Therefore, the architecture requires:

\[
\text{Registers} = \left\lfloor \frac{600}{2} \right\rfloor = 300,
\]

of two bits;

\[
\text{select} = 9 - 1 = 8 \text{ bits}
\]

for the register selection; and

\[
N\text{Adders} = \left\lfloor \frac{2 \times 24 + 2 - 1}{2} \right\rfloor = 25
\]

two bit adders.

To analyse the register and adder usage in the execution stage, the segment size can also be written
in function of $N\text{Adders}$,

$$y = \left\lceil \frac{2l - 1}{N\text{Adders} - 1} \right\rceil_2,$$

(3.3)

where the $\lceil x \rceil_2$ notation means the smallest power of two greater or equal than $x$, which in this case is $(2l - 1)/(N\text{Adders} - 1)$.[Figure 3.5] shows the variation of the segment size $y$ with the number of adders for four floating-point formats: 16, 32, 64 and 128.

![Figure 3.5: Relationship between the segment size (bits) and number of adders, for four floating-point with 16, 32, 64 and 128 bits](image)

The minimum number of adders in the segmented accumulator is two (one adder is a non-segmented accumulator). Therefore, for two adders in the segmented accumulator the architecture must use the following parameters:

- the segment size $y$ is $\lceil 2l - 1 \rceil_2$;
- the shift uses $\log_2 (\lceil 2l - 1 \rceil_2)$ from the exponent;
- there are $\lceil L/\lceil 2l - 1 \rceil_2 \rceil$ registers in the register file;
- and $\text{exp} - \lceil 2l - 1 \rceil_2 - 1$ bits to select the registers.

**Example 3.4.2.** For a segmented accumulator which supports the IEEE 754 single-precision format to use the least amount of resources possible in the execute stage, the following requirements must be met. The size of $L$ is $554 + k$ bits, with $k = 46$ bits $\Rightarrow L = 600$ bits. The smallest number of adders in the segmented accumulator is two. Therefore, the architecture requires:

$$y = \lceil 2 \times 24 - 1 \rceil_2 = 64 \text{ bits};$$

the shift uses

$$\text{shift} = \log_2 (64) = 6 \text{ bits};$$
Continued

the register selection uses

\[ select = 9 - 6 = 3 \text{ bits}; \]

and

\[ \text{Registers} = \lceil 600/64 \rceil = 10 \]

of 64 bits.

Concluding, larger segment sizes implies a reduced number of registers and therefore a smaller register file decoder, and vice versa. Therefore, when optimising the critical path more adders and registers should be used, and when optimising for resource usage, larger segment sizes should be used.

### 3.5 Conclusion

This chapter proposes a segmented accumulator architecture, which stalls the accumulation process when there are any carries to propagate. The carry propagation uses a dedicated block, the CPB.

The carry optimisation uses two techniques. The first allows the CPB to start propagating carries at the first register that will absorb the least significant carry, by leveraging the \text{select_registers} signal. The second predicts the minimum number of cycles required by the CPB by inspecting the generated carries. Both techniques in conjunction allow the stop propagating carries condition to be well defined while guaranteeing that every carry propagates correctly and definitively.

When designing a segmented accumulator the size of the segments may be adjusted in order to meet the design requirements. A lower segment size, with more (smaller) adders and registers, provides a reduced critical path. Conversely, increasing the segment size, with less (larger) adders reduces the resource usage.

However, in most situations the accumulation processor does not need to stall while the CPB is processing carries. The next chapter explores this idea.
Chapter 4

Segmented Accumulator Non-Stalling Architecture

Based on the conclusions made in the previous chapter, this chapter proposes an architecture which is able to process two vectors of any size without stalling.

The first section describes the improvements made to the previous architecture, specifically the execution stage and the CPB. The second section analyses the hazards that may prevent the architecture from operating without stalling. Herein a relation is found between the number of stalls, the number of adders and the length of the critical path, which allows the architecture to be optimised for critical path or resources used.

4.1 Basic Non-Stalling Architecture

The non-stalling architecture uses a dedicated adder, to propagate any carry generated by the segmented accumulator, such that the CPB autonomously finds and propagates the carries from the register file. The improved CPB is, for the remainder of this document, called Free Flow Adder (FFA). Furthermore, to improve the throughput and to shorten the critical path, the previous execution stage (Figure 3.3) is split into three stages:

- The Segment Fetch (SF) stage: where the segments are read from the register file and the summand is split into \( n \) segments, and where the FFA selects the carry to propagate;
- The Execute (EX) stage: where the addition operation is performed for both the segmented accumulator and the FFA;
- The Write Back (WB) stage: where the results of the EX stage are written to the register file.

The block diagram of the datapath is shown in Figure 4.1.

The FFA is managed by identifying non-zero status flags in each carry register. In every cycle, the FFA selects the first carry register which has a carry to propagate. If no carry register has a carry to propagate then the FFA idles. After the selection of the carry register, the FFA will store the register ID to avoid selecting a carry that is being processed. This requirement is due to the propagate operation requiring more than one cycle to write the result. Since three carries can be processed at the same time in different stages, the FFA needs to store the previous two selections. Furthermore, the FFA cannot select any of the segments selected by the summand, otherwise there would be two write operations in the same register.
4.2 Hazard Identification & Solutions

There are two important hazards that may force the processor to stall. The first hazard occurs when two consecutive summands select the same registers, \textit{i.e.}, summand $n$ will consume the carries out and accumulation results from summand $n - 1$. This hazard is referred herein as \textbf{Read After Write (RAW)}.

\textbf{Example 4.2.1.} Consider, in the architecture from Figure 4.1 that all summands select register seven and that a carry-in will be generated for registers eight, nine and ten. The current summand must read the most recent values of each register, which may have not been yet written to the register file when the summand reaches the \textbf{SF} stage. As a result, the summand will be operating on the wrong data. Figure 4.2 shows the sequence of operations to obtain a \textbf{RAW} hazard.
The second hazard occurs when an accumulation generates a carry to a carry register that already holds a carry, and the FFA has not yet propagated its contents. This hazard is referred herein as Carry Write Conflict (CWC).

Example 4.2.2. Consider, in the architecture from Figure 4.1 that all registers in the register file have all bits set to one. Any operation performed to any register will, necessarily, generate three carries out in the same cycle. If the first summand selects the least significant registers, the next summand selects the three next most significant registers, and so forth, every register will have a carry to propagate after six cycles (there are 18 registers in the register file \(\Rightarrow 18/3 = 6\)). In this situation, a specific addition to one of the most significant registers generates another carry out. However, the FFA is only able to propagate one carry per cycle. Even if the FFA starts propagating carries in the next cycle after their generation, the FFA will not be able to propagate the most significant carries before the new carry out is written to the carry register. Therefore, the stored carry is overwritten and lost. Figure 4.3 shows the sequence of operations to obtain a CWC hazard.
4.2.1 Read After Write Hazard Analysis

A common occurrence during processing is the existence of a data dependency between the current summand and the previous summand's computation. However, because it takes four cycles to write the new values in the register file, the current summand in the SF stage will read outdated data from the register file. Although this is not a traditional RAW hazard, due to the nature of the architecture, much of the same ideas are still prevalent. In this case the register selection may overlap between summand's, i.e., a summand \( n \) can select one, or more of summand's \( n-1 \) registers and carries out. When the summand's \( n \) register selection is equal to the register selection of summand's \( n-1 \) or \( n-2 \), summand \( n \) must select all registers from the EX or WB stage, respectively. If the summand's \( n \) register selection differs from summand's \( n-1 \) by one, summand \( n \) must select all registers except the most significant one; for a register selection difference of two, summand \( n \) must select all registers except the two most significant one's; etc. Thus, there will only be registers read from the EX or WB stages, if the difference between the current register selection and the stages register selection is

\[
[-NAdders + 1, NAdders - 1],
\]  

(4.1)

\( e.g., \) using \( NAdders = 4 \) a read from advanced stages occurs when the difference is in \([-3, 3]\).

The number of possible register selections for any given register selection is \( NAdders \times 2 - 1 \). The selection of the addition result, for a register selection in the SF stage \( sfr \), and a register selection in

<table>
<thead>
<tr>
<th>MUL/SHIFT</th>
<th>SF</th>
<th>EX</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>...</td>
<td>( S_4 )</td>
<td>( S_5 )</td>
</tr>
<tr>
<td>Sel. Seg. 0</td>
<td>...</td>
<td>Gen. Carry 7, 8 &amp; 9</td>
<td>Write Results</td>
</tr>
<tr>
<td>( S_5 )</td>
<td>...</td>
<td>( S_4 )</td>
<td>( S_6 )</td>
</tr>
<tr>
<td>Sel. Seg. 12</td>
<td>...</td>
<td>Gen. Carry 10, 11 &amp; 12</td>
<td>Write Results</td>
</tr>
<tr>
<td>( S_6 )</td>
<td>...</td>
<td>( S_7 )</td>
<td>( S_6 )</td>
</tr>
<tr>
<td>Sel. Seg. 15</td>
<td>...</td>
<td>Gen. Carry 13, 14 &amp; 15</td>
<td>Write Results</td>
</tr>
<tr>
<td>( S_7 )</td>
<td>( S_7 )</td>
<td>( S_6 )</td>
<td>( S_7 )</td>
</tr>
<tr>
<td>Sel. Seg. 15</td>
<td>...</td>
<td>Gen. Carry 16, 17 &amp; 18</td>
<td>Write Results</td>
</tr>
<tr>
<td>( S_8 )</td>
<td>( S_7 )</td>
<td>( S_7 )</td>
<td>( S_7 )</td>
</tr>
<tr>
<td>Gen. Carry 17</td>
<td>...</td>
<td>Overwrite Carry 17</td>
<td>FFA: Selects Seg. 6 to propagate carry, Seg. 4 is updated</td>
</tr>
</tbody>
</table>

Figure 4.3: CWC hazard
the EX stage $\text{exr}_s$, is generally described as:

\[
\text{Selected Adder Result}(i) = i - \text{exr}_s + \text{sfr}_s, \quad \text{if} \quad \text{exr}_s - \text{sfr}_s \leq i \land \text{exr}_s - \text{sfr}_s \geq i - N\text{Adders} + 1,
\]

(4.2)

where $i$ is the segment number, $i \in [0, N\text{Adders}]$. Equation 4.2 identifies the adder that contains the result which summand $n$ must read.

**Example 4.2.3.** Consider an architecture with three adders in the EX stage. There are $N\text{Adders} \times 2 - 1 = 3 \times 2 - 1 = 5$ possible selections. The selection of register seven by summand $n$ will consequentially read registers seven, eight and nine from the register file. If summand $n + 1$ and summand $n$ have a register selection difference in the range $[-2, 2]$, then it must read some or all of its operands from the adders results in the EX stage.

- If summand $n+1$ has $\text{sfr}_s = 8$, the selected registers are eight, nine and ten. The difference between $\text{exr}_s$ and $\text{sfr}_s$ is $7 - 8 = -1 \in [-2, 2]$. Thus, summand $n + 1$ must read registers eight and nine from adders one and two in the EX stage, respectively;
- If summand $n + 1$ has $\text{sfr}_s = 6$, the selected registers are six, seven and eight. The difference between $\text{exr}_s$ and $\text{sfr}_s$ is $7 - 6 = 1 \in [-2, 2]$. Thus, summand $n + 1$ must read registers seven and eight from adders zero and one in the EX stage, respectively.

In the EX or WB stage, register $y + 1$ absorbs the carry out from register $y$. When summand’s $n$ register selection is equal to the register selection of summand’s $n - 1$ or $n - 2$, summand $n$ must select all carries except the least significant from the EX or WB stage, respectively. If summand’s $n$ register selection differs from summand’s $n - 1$ by one, summand $n$ must select all carries except the two least significant ones; for a register selection difference of two, summand $n$ must select all carries except the three least significant ones; etc. Thus, there will only be carries read from the EX or WB stages, if the difference between the current register selection and the stages register selection is:

\[
[-N\text{Adders}, N\text{Adders} - 2],
\]

(4.3)

e.g., using $N\text{Adders} = 4$ a read from advanced stages occurs when the difference is in $[-4, 2]$.

The number of possible register selections for any given register selection is $N\text{Adders} \times 2$. The selection of the carry out, for a register selection in the SF stage $\text{sfr}_s$ and a register selection in the EX stage $\text{exr}_s$, is generally described as:

\[
\text{Selected Carry Out}(i) = i - \text{exr}_s + \text{sfr}_s - 1, \quad \text{if} \quad \text{exr}_s - \text{sfr}_s < i \land \text{exr}_s - \text{sfr}_s \geq i - N\text{Adders},
\]

(4.4)

where $i$ is the segment number, $i \in [0, N\text{Adders}]$. Equation 4.4 identifies the adder that contains the carry which summand $n$ must absorb.

**Example 4.2.4.** Consider an architecture with three adders in the EX stage. There are $N\text{Adders} \times 2 = 3 \times 2 = 6$ possible selections. If summand $n + 1$ and summand $n$ have a register selection difference in the range $[-3, 1]$, then it must read some or all of its carries in from the adder’s carry out in the EX stage.
• If summand \( n+1 \) has \( sfr_s = 8 \), the selected registers are eight, nine and ten. The difference between \( exr_s \) and \( sfr_s \) is \( 7 - 8 = -1 \in [-3, 1] \). Thus, summand \( n+1 \) must read carries out seven, eight and nine from adders zero, one and two in the \( \text{EX} \) stage, respectively;

• If summand \( n+1 \) has \( sfr_s = 6 \), the selected registers are six, seven and eight. The difference between \( exr_s \) and \( sfr_s \) is \( 7 - 6 = 1 \in [-3, 1] \). Thus, summand \( n+1 \) must read carry out seven from adder zero in the \( \text{EX} \) stage.

### 4.2.2 Read After Write Hazard Solution

Asserting that more than one summand is operating over the same data, in different pipeline stages, requires each stage to keep its summand’s register selection until the end of the \( \text{WB} \) stage. Once the summand reaches the \( \text{SF} \) stage it can read the most recent registers from the register file, or from the \( \text{EX} \) or \( \text{WB} \) stages. The selection uses MUXs at the end of the \( \text{SF} \) stage and forwarding paths taken from the end of the \( \text{EX} \) and \( \text{WB} \) stages.

The \( \text{FFA} \) has a simpler register selection scheme, because it cannot select from itself. There is another important detail regarding the reset operation at the end of the propagate operation, when the \( \text{FFA} \) is propagating a carry to the immediate register after the summand’s selection, e.g., the summand selects registers seven, eight and nine, and the \( \text{FFA} \) selects register ten. Then, the reset operation of the \( \text{FFA} \) and the write operation of register ten’s carry out will generate a structural hazard due to two distinct signals trying to write to the same carry register. The solution is to always write the contents of the previous register carry out. The updated block diagram for the \( \text{SF} \) stage is shown in Figure 4.4.

Another important detail occurs when a summand selects a carry through a forwarding path. A carry selection is equivalent to a carry read operation, which, much like the \( \text{FFA} \), must reset the carry in the advanced stage. The carry delete operation is performed when:

• the difference between the register selection of the summand in the \( \text{SF} \) stage and one of the stages, or the \( \text{FFA} \) is in the range resulting from Equation 4.3
• the write enable flags are set in both stages, or the \( \text{FFA} \) and
• the forwarding path is selected by either the segments or the \( \text{FFA} \)

The final datapath block diagram is shown in Figure 4.5.
Figure 4.4: Datapath for the Segment Fetch Stage with three adders in the Execute Stage and one FFA.
Figure 4.5: Final Datapath with three adders in the Execute Stage and one FFA (depicted with a dashed box)
4.2.3 Carry Write Conflict Hazard Analysis

The \text{CWC} hazard occurs when the \text{FFA} fails to process the pending carries, before another accumulation to a more significant register generates another carry in a position, which has yet to be propagated. The worst case occurs when the carry register file is full, i.e., all carry registers store a carry bit. The size of the carry register to withstand the minimum number of stored carries before a \text{CWC} hazard occurs depends on the number of maximum carries generated by the summand, which is

\[
\text{Number of Max. Carries} = \left\lceil \frac{L}{y} \right\rceil - \left\lfloor \frac{k}{y} \right\rfloor, \tag{4.5}
\]

where \(y\) is the size of the segment, \(L\) is the fixed-point accumulation size (Equation 2.4), and \(k\) is the number of bits to guard from overflow. Since the \text{FFA} can only select registers that are not selected by the summand, the number of selectable registers by the \text{FFA} is \(\text{Number of Max. Carries} - N\text{Adders}\). For any given number of generated carries, in a single cycle, the \text{FFA} can only propagate one carry. Therefore, the minimum number of bits in the carry register is

\[
\text{Carry Register Size} \geq \left\lceil \log_2 (\text{Number of Max. Carries} - N\text{Adders} + 1) \right\rceil, \tag{4.6}
\]

where the addition of one in the equation represents a carry number with all bits set to zero, which does not count towards the total. By increasing the redundancy of the carry save mechanism the carry’s write operation has to change. With a carry register size greater than one, a write operation is an accumulation to the register.

**Example 4.2.5.** An IEEE 754 single-precision architecture with \(y = 32\) bits and \(k = 100\) bits, has an \(L = 654\) bits. Applying Equation 4.5

\[
\text{Number of Max. Carries} = \left\lceil \frac{654}{32} \right\rceil - \left\lfloor \frac{100}{32} \right\rfloor = 21 - 3 = 18.
\]

The segmented accumulator uses 3 adders, and applying Equation 4.6

\[
\text{Carry Register Size} \geq \left\lceil \log_2 (18 - 3 + 1) \right\rceil = 4 \text{ bits}.
\]

Each carry register must have at least 4 bits to avoid CWC hazards.

Equation 4.6 shows there is a relation between the number of stalls, the segment size, and the number of adders. Rewriting Equation 4.6 in function of the segment size \((y)\) is

\[
\frac{L}{y} - \frac{k}{y} = \frac{2l + y - 1}{y} + 1 = \frac{2e_{\text{max}} + 2|e_{\text{min}}| + 1}{y}. \tag{4.7}
\]

Substituting in Equation 4.8

\[
\text{Carry Register Size} = \log_2 \left( \frac{2e_{\text{max}} + 2|e_{\text{min}}| + 1}{y} \right) \Leftrightarrow \frac{y}{2} = \frac{2e_{\text{max}} + 2|e_{\text{min}}| + 1}{2 \text{Carry Register Size}} \tag{4.8}
\]

Figure 4.6 shows the variation of the number of bits in the carry register with the segment size.
As expected the bigger the size of the segment the smaller the size of the carry register must be. Equation 4.6 defines the relation between the minimum number of bits in the carry register for the architecture to not stall, and the number of selectable registers (NAdders).

If the non-stalling conditions are not verified, then the number of stalls is the difference between the maximum number of selectable registers, and the maximum stored carry in the carry register

\[ \text{Number of Stalls}(\text{Carry Register Size}, y) = \frac{2e_{\text{max}} + 2|e_{\text{min}}| + 1}{y} - 2\text{Carry Register Size}. \]  

(4.9)

Figure 4.7 shows how the number of stalls varies with the segment size and the number of bits in the carry register.

Again, as expected, as the size of the segment increases the number of stalls decreases, and increasing the number of bits in the carry register decreases the number of stalls for any floating point size. It is also important to note that to have no stalls and smaller segment sizes the critical path may change from the adders in the EX stage to the carry accumulation in the carry register file, because the carry register size will be greater than the segment size. Thus, the best critical path is achieved when the width of the adder is equal, or close to, the size of the carry register. Observing Figures 4.7d and 4.7c, the carry register size needed to achieve no stalls is much smaller than the segment size, which means the segment size should be smaller to achieve a better critical path. Therefore, the best critical path for floating-points with 128 bits is achieved with a segment size of 16 bits, for floating-points with 64 bits with segment size of 16 bits, for floating-points with 32 bits with segment size of 8 bits and for floating-points with 16 bits with a segment size of 4 bits. Generally, to achieve the best critical path for any floating-point size:

\[ y = \lceil \lfloor \log_2 (2e_{\text{max}} + 1) \rceil \rceil, \]  

(4.10)

where the \( \lceil x \rceil \) notation means the smallest power of two greater or equal than \( x \), which in this case is \( \lfloor \log_2 (2e_{\text{max}} + 2|e_{\text{min}}| + 1) \rfloor \), \( y \) is the segment size and a power of two, and \( e_{\text{max}} \) is the maximum exponent of the floating-point.
Example 4.2.6. Applying Equation 4.10 to an architecture that uses the IEEE 754 single-precision format:

\[ y = \lceil \log_2 (2 \times 127 + 1) \rceil_2 = 8 \text{ bits}. \]

The architecture should use a segment size of eight bits to achieve the shortest critical path. Having a segment size of 8 bits \( \Rightarrow N\text{Adders} = 7 \).

To achieve no stalls, the carry register size must be (from Equation 4.6):

\[ \text{Carry Register Size} \geq 6 \text{ bits}. \]

Concluding, to achieve the best critical path a segmented accumulator architecture using IEEE 754 single-precision format should have: 8 bit segments, 7 adders in the \( \text{EX} \) stage, and a carry register size of 6 bits to avoid stalling.

Example 4.2.7. Applying Equation 4.10 to an architecture that uses the IEEE 754 double-
Continued

precision format:

\[
y = \left\lceil \log_2 \left( 2 \times 1023 + 1 \right) \right\rceil_2 = 16 \text{ bits}.
\]

To achieve the best critical path the segment size is 16 bits. A segment size of 16 bits \( \Rightarrow N_{\text{Adders}} = 8 \).

To achieve no stalls, the carry register size must be (from Equation 4.6):

\[
\text{Carry Register Size} \geq 8 \text{ bits}.
\]

Concluding, to achieve the best critical path a segmented accumulator architecture using IEEE 754 double-precision format should have: 16 bit segments, 8 adders in the \( \text{EX} \) stage, and a carry register size of 8 bits to avoid stalling.

Equation 4.6 shows the relation between the size of the carry register and the number of adders used in the \( \text{EX} \) stage. Thus, it can be rewritten as the segment size in function of the number of adders yields:

\[
y = \left\lceil \frac{2l - 1}{N_{\text{Adders}} - 1} \right\rceil_2. \tag{4.11}
\]

Replacing \( y \) in Equation 4.6:

\[
\text{Carry Register Size} \geq \left\lfloor \log_2 \left( \frac{2e_{\text{max}} + 2|e_{\text{min}}| + 1)(N_{\text{Adders}} - 1)}{2l - 1} \right) \right\rfloor. \tag{4.12}
\]

**Example 4.2.8.** An architecture that uses the IEEE 754 single-precision format which requires two adders in the \( \text{EX} \) stage must have a segment size of (from Equation 4.11):

\[
y = \left\lceil \frac{2l - 1}{N_{\text{Adders}} - 1} \right\rceil_2 = 64 \text{ bits}.
\]

The segment size is 64 bits.

A segment size of 64 bits requires a carry register size of: (from Equation 4.12):

\[
\text{Carry Register Size} \geq 3 \text{ bits}.
\]

Concluding, a segmented accumulator architecture using IEEE 754 single-precision format with two adders in the \( \text{EX} \) stage has: 64 bit segments, and a carry register size of 3 bits to avoid stalling.

**Example 4.2.9.** An architecture that uses the IEEE 754 double-precision format which requires two adders in the \( \text{EX} \) stage must have a segment size of (from Equation 4.11):

\[
y = \left\lceil \frac{2l - 1}{N_{\text{Adders}} - 1} \right\rceil_2 = 128 \text{ bits}.
\]

The segment size is 128 bits.
A segment size of 128 bits requires a carry register size of (from Equation 4.12):

\[ \text{Carry Register Size} \geq 5 \text{ bits}. \]

Concluding, a segmented accumulator architecture using IEEE 754 double-precision format with two adders in the EX stage has: 128 bit segments, and a carry register size of 5 bits to avoid stalling.

Figure 4.8 shows the variation of the number of bits in the carry register with the number of adders in the EX stage.

\[
\text{Number of Adders / Carry Register Size (bits)}
\]

![Graph showing the relation between the number of bits in the carry register and the number of adders in the EX stage, for Floating-Point sizes of 16, 32, 64 and 128 bits.]

The number of adders increases with the size of the carry register and of the floating-point operand. If the non-stalling conditions are not verified, then the number of stalls is:

\[
\text{Number of Stalls} = \frac{(N\text{Adders} - 1)(2^{e_{\max}} + 2|e_{\min}| + 1)}{2^l - 1} - 2^\text{Carry Register Size}. \tag{4.13}
\]

Figure 4.9 shows how the number of stalls varies with the number of adders in the EX stage and the number of bits in the carry register.
As the number of adders increases so does the number of stalls, for any floating-point size. Equation 4.13 also indicates that it is impossible to have no stalls with only one adder in the EX stage. This is the same conclusion reached in Section 3.4.

The lower resource configuration is the one that has two adders in the EX stage. Using two adders, besides decreasing the size of the register file decoder, also decreases the number of required MUXs in the SF stage (subsection 4.2.1). The use of two adders requires twelve MUXs in the SF: two 2:1 MUX’s to select the register result from the EX and WB stages, two 2:1 MUX’s to select the carry out result from the EX and WB, and two 2:1 MUX’s to select from the register MUX’s and carry out MUX’s, for each adder.

4.2.4 Carry Write Conflict Hazard Solution

Equation 4.6 and Equation 4.9 define the conditions necessary for the accumulation processor to not stall. To solve the CWC hazard the size of the carry register must be sufficient to accommodate for more carries and the write operation to the carry register becomes an accumulation.
4.3 Conclusion

This chapter proposes an architecture that is able to avoid stalling the accumulation process while processing two vectors. The FFA allows the processing of any input while propagating carries, being autonomous in its selection, and, above all else, avoiding hazards. RAW hazards are avoided by using forwarding paths from the EX and WB stages. CWC hazards are avoided by increasing the redundancy of the carry save mechanism, i.e., by storing more than one bit, and changing the write operation of a carry to an accumulation.

The architecture may be optimised for critical path, performance or resource usage. Optimising for no-stalls requires a multi-bit carry register, and six extra MUXs per adder for forwarding paths. Optimising for resources used requires lowering the size of the register file decoder and the number of adders, thus lowering the number of forwarding paths, i.e., using a large segment size. Similarly to the stalling architecture, shortening the critical path requires a smaller segment size.
Chapter 5

Segmented Accumulator
Addition/Subtraction Architectures

The two architectures proposed in the previous chapters do not support signed addition/subtraction. As such, this chapter proposes a signed redundant number system which allows the segmented accumulator to perform signed addition. Using this signed representation, modifications to both architectures are proposed and analysed.

5.1 Generalised Signed-Digit Number Representations

Traditional signed representations, require a sign propagation to every register starting at the summand’s selection. Since the accumulator uses segmented access, traditional signed representations are not sufficient. In [36], Parhami proposes a framework to design redundant number representations. There are eight variables which define the Generalized Signed-Digit (GSD) Table 5.1 shows their description.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Radix</td>
</tr>
<tr>
<td>α</td>
<td>Minimum Redundant Digit</td>
</tr>
<tr>
<td>β</td>
<td>Maximum Redundant Digit</td>
</tr>
<tr>
<td>λ</td>
<td>Minimum Carry</td>
</tr>
<tr>
<td>μ</td>
<td>Maximum Carry</td>
</tr>
<tr>
<td>$p_i$</td>
<td>Position Sum</td>
</tr>
<tr>
<td>$u_i$</td>
<td>Interim Sum</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Redundancy Index</td>
</tr>
</tbody>
</table>

A digit set of radix $r$ is redundant if it contains more than $r$ digits. A GSD set $[-\alpha, \beta] (d_i)$ is redundant when [37]

$$\alpha \geq 0, \beta \geq 0, \text{ and } \alpha + \beta + 1 > r. \quad (5.1)$$

An important parameter of the GSD is its redundancy index. The redundancy index represents the amount by which the size of its digit set exceeds the size $r$ of a non-redundant digit set for radix $r$ [38].
The redundancy index is
\[ \rho = \alpha + \beta + 1 - r. \]  
(5.2)

Therefore, adding two GSD numbers, \( x_i \) and \( y_i \), can break the carry propagation chain with the following algorithm [36]:

1. Compute position sum \( p_i = x_i + y_i \);
2. Separate \( p_i \) in an interim sum \( u_i \) and a carry digit \( c_i \), such that \( c_i \in [-\lambda, \mu] \) and \( u_i = p_i - rc_i \);
3. Calculate the final sum digit \( d_i = u_i + c_i - 1 \).

**Algorithm 5**: Carry free addition steps using GSD numbers

Algorithm 5 is applicable to a GSD redundant representation if one of the following conditions is satisfied [36]:

- \( r > 2, \rho \geq 3 \),
- \( r > 2, \rho = 2, \alpha \neq 1, \beta \neq 1 \).

Having a carry free addition implies that the carry is immediately absorbed by the next digit. Thus, for a digit \( d_i \) to absorb one carry, the interim sum is limited between

\[ u_i = d_i - c_{i-1} = [-\alpha, \beta] - [-\lambda, \mu] = [-\alpha + \lambda, \beta - \mu]. \]  
(5.3)

Therefore, the minimal values for \( \mu \) and \( \lambda \) are

\[ \lambda_{\text{min}} = \left\lfloor \frac{\alpha}{r - 1} \right\rfloor, \text{ and } \mu_{\text{min}} = \left\lfloor \frac{\beta}{r - 1} \right\rfloor, \]  
(5.5)

which has the minimal digit carry set \( \{-1, 0, 1\} \) [36]. Once \( \lambda \) and \( \mu \) are known the carry digit selection is performed by comparing the position sum \( p_i \) with the constants \( C_j \), where \( -\lambda < j \leq \beta \).

**Example 5.1.1.** For the minimal digit carry set \( \{-1, 0, 1\} \) there are two constants, \( C_0 \) and \( C_1 \), and the selection is

\[ c_i = \begin{cases} 
1 & \text{if } p_i \geq C_1 \\
0 & \text{if } C_0 \leq p_i < C_1 \\
-1 & \text{if } p_i < C_0 
\end{cases} \]

Generally, for any carry digit set, the following constraints must be met

\[ \lambda - \alpha + j r \leq C_j \leq \beta - \mu + 1 + (j - 1)r, \]  
(5.6)

where \( \lambda < j \leq \mu \). The choice of the \( C_j \) constants have a direct impact in the conversion between the position sum \( p_i \), and the interim sum \( u_i \) and carry digit \( c_i \). Herein the comparison constants \( C_j \) are selected as to obtain both parameters \( (u_i, c_i) \) directly from the position sum \( p_i \), i.e., the MSBs of the position sum \( p_i \) defines the carry and the remaining LSBs define the interim sum \( u_i \). Therefore, the constants selected must be powers of two for the previous condition to hold true. The \( C_j \) constants selected, for a carry digit set \( c_i \), are

\[ C_{1-\lambda} = -2^{\lambda+\mu}, \quad C_{2-\lambda} = -2^{\lambda+\mu-1}, \ldots, C_0 = -2^{\mu+1}, \quad C_1 = 2^{\lambda+1}, \ldots, C_{\mu-1} = 2^{\lambda+\mu-1}, \quad C_\mu = 2^{\lambda+\mu}, \]  
(5.7)
where the constant is negative when $j \leq 0$, and $\lambda < j \leq \mu$.

Selecting the $C_j$ constants sets restrictions on $\alpha$ and $\beta$. Rewriting Equation 5.6 and Equation 5.5

$$\lambda + (1 - \lambda)r - C_{1-\lambda} \leq \alpha \leq \lambda(r - 1)$$

and

$$C_{\mu} + (\mu - 1)(1 - r) \leq \beta \leq \mu(r - 1).$$

Example 5.1.2. For the carry digit set $\{-2, -1, 0, 1, 2\}$, the constants are (from Equation 5.7)

$C_{-1} = -16, C_0 = -8, C_1 = 8, C_2 = 16.$

$\alpha$ and $\beta$ are

$$\alpha \geq 2 + (1 - 2)r - C_{-1} = 2 - r + 16 = 18 - r \text{ and}$$

$$\alpha \leq \lambda(r - 1) = 2(r - 1),$$

and

$$\beta \geq C_{\mu} + (2 - 1)(1 - r) = 16 + 1 - r = 17 - r \text{ and}$$

$$\beta \leq \mu(r - 1) = 2(r - 1).$$

For $r = 2^4$, $2 \leq \alpha \leq 30$ and $1 \leq \beta \leq 30$. Thus, selecting $\alpha = 8$ and $\beta = 7 \Rightarrow d_i = [-8, 9].$

To facilitate the add/subtract operation of any redundant system, the position sum $p_i$, the digit set $d_i$, and the interim sum $u_i$ use the two's complement representation. This way the redundant system uses the minimum amount of resources to perform each operation. The carry digit set $c_i$ uses the one's complement representation to further ease decoding (this will be clear in Examples 5.1.3 and 5.1.4).

The number of bits required by the position sum $p_i$ is dependent on the interim sum $u_i$ and the carry digit set $c_i$. Therefore, the minimum number of bits to decode $u_i$ into two's complement is

$$\text{size}(u_i) = \lceil \log_2(\max(|u_i|)) \rceil + 1.$$  

(5.10)

The number of bits used for the carry digit set $c_i$ is

$$\text{size}(c_i) = \begin{cases} 
\lceil \log_2(\max(|c_i|)) \rceil + 2 & \text{if } \mu \text{ is a power of two} \\
\lceil \log_2(\max(|c_i|)) \rceil + 1 & \text{if } \mu \text{ is not a power of two}
\end{cases}$$

(5.11)

in one's complement. As a result, the positional sum range is

$$p_i = [-(\lambda r + u_{\min}), \mu r + u_{\max}]$$

$$= [-\lambda r - \alpha + \lambda, \mu r + \beta - \mu]$$

$$= [\lambda(1 - r) - \alpha, \mu(r - 1) + \beta],$$

(5.12)

and the minimum number of bits is

$$\text{size}(p_i) = \lceil \log_2(\max(|p_i|)) \rceil + 1.$$  

(5.13)
Example 5.1.3. Designing a redundant number system with a carry digit set \( c_i = [-2, 2] \) and radix \( r = 2^3 \), will have the following comparison constants (from Equation 5.7)

\[
C_{-1} = -16, \ C_0 = -8, \ C_1 = 8, \ C_2 = 16.
\]

The digit set \( d_i \) is in the range

\[
\begin{align*}
\alpha & \geq 2 + (1 - 2)r - C_{-1} = 2 - r + 16 = 18 - r = 10 \quad \text{and} \\
\alpha & \leq \lambda(r - 1) = 2(r - 1) = 14,
\end{align*}
\]

and

\[
\begin{align*}
\beta & \geq C_\mu + (2 - 1)(1 - r) = 16 + 1 - r = 17 - r = 9 \quad \text{and} \\
\beta & \leq \mu(r - 1) = 2(r - 1) = 14.
\end{align*}
\]

Selecting \( \beta = 9 \) and \( \alpha = 10 \) \( \Rightarrow d_i = [-10, 9] \). The position sum is in the range \( p_i = [\lambda(1 - r) - \alpha, \mu(r - 1) + \beta] = [-24, 23] \), and the interim sum is in the range \( u_i = [-8, 7] \). The minimum number of bits for \( p_i \), \( u_i \), \( c_i \) and \( d_i \) is

\[
\begin{align*}
\text{size}(u_i) &= \lceil \log_2 (8) \rceil + 1 = 4 \text{ bits}, \\
\text{size}(c_i) &= \lceil \log_2 (2) \rceil + 2 = 3 \text{ bits}, \\
\text{size}(p_i) &= \lceil \log_2 (24) \rceil + 1 = 6 \text{ bits}, \\
\text{size}(d_i) &= \lceil \log_2 (10) \rceil + 1 = 4 + 1 = 5 \text{ bits}.
\end{align*}
\]

Using Table 5.2 for the position sum \( p_i \) decoding and Table 5.3 for the carry decoding, the carry digit \( c_i \) and interim sum \( u_i \) are obtained as follows:

\[
\begin{align*}
c_i &= p_{i5} \ p_{i4} \ p_{i3} , \\
u_i &= p_{i5} \ p_{i2} \ p_{i1} \ p_{i0} .
\end{align*}
\]
Table 5.2: Addition steps for $u_i = [-8, 7]$ and $c_i = [-2, 2]$

<table>
<thead>
<tr>
<th>$p_{12}$</th>
<th>$p_{10}$</th>
<th>$c_{12}$</th>
<th>$c_{10}$</th>
<th>$u_{12}$</th>
<th>$u_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>010111</td>
<td>23</td>
<td>010</td>
<td>2</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>010000</td>
<td>16</td>
<td>010</td>
<td>2</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>001111</td>
<td>15</td>
<td>001</td>
<td>1</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>001000</td>
<td>8</td>
<td>001</td>
<td>1</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>000111</td>
<td>7</td>
<td>000</td>
<td>0</td>
<td>0111</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>000000</td>
<td>0</td>
<td>000</td>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>111111</td>
<td>-1</td>
<td>111</td>
<td>0</td>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>111000</td>
<td>-8</td>
<td>111</td>
<td>0</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>110111</td>
<td>-9</td>
<td>110</td>
<td>-1</td>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>110100</td>
<td>-16</td>
<td>110</td>
<td>-1</td>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>101111</td>
<td>-17</td>
<td>101</td>
<td>-2</td>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>101000</td>
<td>-24</td>
<td>101</td>
<td>-2</td>
<td>1000</td>
<td>-8</td>
</tr>
</tbody>
</table>

Table 5.3: Carry code decode for carry digit set $c_i = [-2, 2]$

<table>
<thead>
<tr>
<th>Carry Code</th>
<th>Carry Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>2</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>110</td>
<td>-1</td>
</tr>
<tr>
<td>101</td>
<td>-2</td>
</tr>
</tbody>
</table>

Although a redundant system designed around the carry digit set is possible, a system designed around the interim sum $u_i$ provides more flexibility when implementing the hardware. The main goal behind designing a redundant system around the interim sum $u_i$ is that its limits can be set to a multiple of the radix, thus simplifying the implementation of the system. For an interim sum $u_i = [-r, r - 1]$, the final sum is

$$d_i = u_i + c_i = [-r, r - 1] + [-\lambda, \mu] = [-r - \lambda, r - 1 + \mu] \Rightarrow \alpha \geq r + \lambda \text{ and } \beta \geq r + \mu - 1.$$

(5.14)
Therefore, the position sum is
\[
p_i = u_i + r c_i = [- (u_{i_{\text{min}}} + r c_{i_{\text{min}}}), u_{i_{\text{max}}} + r c_{i_{\text{max}}}] = [-r - r \lambda, r - 1 + r \mu] = [-r(1 + \lambda), r(1 + \mu) - 1]. \tag{5.15}
\]

The minimum number of bits required for each parameter is:
\[
\begin{align*}
\text{size}(u_i) &= \lceil \log_2 (r - 1) \rceil + 1; \\
\text{size}(c_i) &= \begin{cases} \\
\lceil \log_2 (\max(|c_i|)) \rceil + 2 & \text{if } \mu \text{ is a power of two} \\
\lceil \log_2 (\max(|c_i|)) \rceil + 1 & \text{if } \mu \text{ is not a power of two}
\end{cases} \\
\text{size}(p_i) &= \lceil \log_2 (\max(|p_i|)) \rceil + 1; \\
\text{size}(d_i) &= \lceil \log_2 (\max(|d_i|)) \rceil + 1.
\end{align*}
\tag{5.16a - 5.16d}
\]

Once again, the interim sum \( u_i \) and digit \( d_i \) use two’s complement and the carry digit \( c_i \) uses one’s complement. The comparison constants \( C_j \) are
\[
C_{1-\lambda} = -\lambda 2^\lambda, \quad C_{2-\lambda} = -(\lambda - 1) 2^\lambda, \ldots, \quad C_0 = -2^\lambda, \quad C_1 = 2^\mu, \ldots, \quad C_{\mu-1} = (\mu - 1)2^\mu, \quad C_{\mu} = \mu 2^\mu, \tag{5.17}
\]
where \(-\lambda < j \leq \mu\).

**Example 5.1.4.** Designing a redundant number system using radix \( r = 2^4 \Rightarrow u_i = [-16, 15] \), and a carry digit set \( c_i = [-4, 4] \). Therefore, the digit set is \( d_i = [-20, 19] \), and the position sum is \( p_i = [-80, 79] \). The minimum number of bits required for each parameter is (from Equation 5.16):
\[
\begin{align*}
\text{size}(u_i) &= \lceil \log_2 (16) \rceil + 1 = 5 \text{ bits}; \\
\text{size}(c_i) &= \lceil \log_2 (4) \rceil + 2 = 4 \text{ bits}; \\
\text{size}(p_i) &= \lceil \log_2 (80) \rceil + 1 = 8 \text{ bits}; \\
\text{size}(d_i) &= \lceil \log_2 (20) \rceil + 1 = 6 \text{ bits}.
\end{align*}
\]

Applying Equation 5.17, the comparison constants are
\[
C_{-3} = -64, \quad C_{-2} = -48, \quad C_{-1} = -32, \quad C_0 = -16, \quad C_1 = 16, \quad C_2 = 32, \quad C_3 = 48, \quad C_4 = 64.
\]

Using Table 5.4 for the position sum \( p_i \) decoding and Table 5.5 for the carry digit \( c_i \) decoding, the carry digit \( c_i \) and interim sum \( u_i \) are obtained as follows:
\[
\begin{align*}
c_i &= p_{i7} p_{i6} p_{i5} p_{i4}, \\
u_i &= p_{i7} p_{i3} p_{i2} p_{i1} p_{i0}.
\end{align*}
\]
Table 5.4: Addition steps for $u_i = [-16, 15]$ and $c_i = [-4, 4]$

<table>
<thead>
<tr>
<th>$p_{i2}$</th>
<th>$p_{i10}$</th>
<th>$c_{i2}$</th>
<th>$c_{i10}$</th>
<th>$u_{i2}$</th>
<th>$u_{i10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100 1111</td>
<td>79</td>
<td>0100</td>
<td>4</td>
<td>0 1111</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0010 0000</td>
<td>64</td>
<td>0100</td>
<td>4</td>
<td>0 0000</td>
<td>0</td>
</tr>
<tr>
<td>0011 1000</td>
<td>47</td>
<td>0011</td>
<td>3</td>
<td>0 1111</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0010 0000</td>
<td>32</td>
<td>0010</td>
<td>2</td>
<td>0 0000</td>
<td>0</td>
</tr>
<tr>
<td>0001 0000</td>
<td>16</td>
<td>0001</td>
<td>1</td>
<td>0 1111</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0000 1111</td>
<td>15</td>
<td>0000</td>
<td>0</td>
<td>0 1111</td>
<td>15</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0000 0000</td>
<td>0</td>
<td>0000</td>
<td>0</td>
<td>0 0000</td>
<td>0</td>
</tr>
<tr>
<td>1111 1111</td>
<td>-1</td>
<td>1111</td>
<td>0</td>
<td>1 1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1111 0000</td>
<td>-16</td>
<td>1111</td>
<td>0</td>
<td>1 0000</td>
<td>-16</td>
</tr>
<tr>
<td>1110 1111</td>
<td>-17</td>
<td>1110</td>
<td>-1</td>
<td>1 1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1110 0000</td>
<td>-32</td>
<td>1110</td>
<td>-1</td>
<td>1 0000</td>
<td>-16</td>
</tr>
<tr>
<td>1101 1111</td>
<td>-33</td>
<td>1101</td>
<td>-2</td>
<td>1 1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1101 0000</td>
<td>-48</td>
<td>1101</td>
<td>-2</td>
<td>1 0000</td>
<td>-16</td>
</tr>
<tr>
<td>1100 1111</td>
<td>-49</td>
<td>1100</td>
<td>-3</td>
<td>1 1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1100 0000</td>
<td>-64</td>
<td>1100</td>
<td>-3</td>
<td>1 0000</td>
<td>-16</td>
</tr>
<tr>
<td>1011 1111</td>
<td>-65</td>
<td>1011</td>
<td>-4</td>
<td>1 1111</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1011 0000</td>
<td>-80</td>
<td>1011</td>
<td>-4</td>
<td>1 0000</td>
<td>-16</td>
</tr>
</tbody>
</table>
### 5.2 Using GSD Numbers in the Segmented Accumulator

In this application the GSD base used is $2^y$, where $y$ is the segment size. The interim sum is $u_i = [-2^y, 2^y - 1]$, the digit set is

\[
d_i = u_i + c_i = [-2^y, 2^y - 1] + [-\lambda, \mu] = [-2^y - \lambda, 2^y - 1 + \mu],
\]

thus the digit set $d_i$ is in the interval $[-\alpha, \beta]$, where

\[
\alpha \geq 2^y + \lambda \text{ and } \beta \geq 2^y + \mu - 1.
\]

Thus, the position sum is

\[
p_i = u_i + rc_i
\]

\[
= [-u_{i\text{min}} + rc_{i\text{min}}, u_{i\text{max}} + rc_{i\text{max}}]
\]

\[
= [-2^y - 2^y \lambda, 2^y - 1 + 2^y \mu]
\]

\[
= [-2^y(1 + \lambda), 2^y(1 + \mu) - 1].
\]

The minimum number of bits required for each parameter is:

\[
\text{size}(u_i) = \lceil \log_2 (2^y - 1) \rceil + 1; \quad (5.21a)
\]

\[
\text{size}(c_i) = \begin{cases} 
\lceil \log_2 (\max(|c_i|)) \rceil + 2 & \text{if } \mu \text{ is a power of two} \\
\lceil \log_2 (\max(|c_i|)) \rceil + 1 & \text{if } \mu \text{ is not a power of two} 
\end{cases} \quad (5.21b)
\]

\[
\text{size}(p_i) = \lceil \log_2 (\max(|p_i|)) \rceil + 1; \quad (5.21c)
\]

\[
\text{size}(d_i) = \lceil \log_2 (\max(|d_i|)) \rceil + 1. \quad (5.21d)
\]

Using GSD numbers, the width of the adder is equal to the size of the position sum $p_i$, and its inputs are a segment from the summand and a selected digit $d_i$ from the register. The carry register size is
equal to the size of each carry digit \( c_i \). After the addition, the result of the position sum \( p_i \) splits into two components the carry digit \( c_i \) and the interim sum \( u_i \). The interim sum \( u_i \) is stored in the digit register \( d_i \) with the necessary sign extension, and the carry digit \( c_i \) accumulates to the carry register. Since the carry digit set \( c_i \) uses one’s complement representation and the system must store two’s complement, the \text{MSB} \ of the carry digit \( c_i \) is also the carry in. The unit is shown in the block diagram of Figure 5.1.

The carry propagation is only performed when the carry register overflows. Using GSD numbers, the carry propagation mechanism adds the contents of the digit \( d_i \) and the previous carry \( c_{i-1} \), the result of the addition is the new digit \( d_i \). If this operation overflows the carry out of the addition accumulates with the carry register \( c_i \). When writing the new digit \( d_i \), the previous carry \( c_{i-1} \) is reset.

Figure 5.1: Block Diagram of an addition and subtraction using GSD numbers in the segmented architecture

**Example 5.2.1.** A processor that supports the inner product for single-precision floats using the IEEE 754 format, a segment size of 32 bits and a carry digit set \( c_i = [-2, 2] \) requires three adders of (from Equation 5.21c)

\[
\text{adder width} = \text{size}(p_i) = \lceil \log_2 (2^{32} (1 + 2)) \rceil + 1 = 35 \text{ bits.}
\]

The carry and digit register sizes are (from Equation 5.21b and Equation 5.21d)

\[
\text{size of carry register} = \text{size}(c_i) = \lceil \log_2 (2) \rceil + 2 = 3 \text{ bits},
\]

\[
\text{size of digit register} = \text{size}(d_i) = \lceil \log_2 (2^{32} + 2) \rceil + 1 = 34 \text{ bits.}
\]

The size of the interim sum \( u_i \) is (from Equation 5.21a)

\[
\text{size}(u_i) = \lceil \log_2 (2^{32} - 1) \rceil + 1 = 33 \text{ bits.}
\]

After the addition, the position sum \( p_i \) splits into two words, the interim sum \( u_i \) and the carry digit.
Continued

c_i. Therefore, the decoding of the position sum $p_i$ for the interim sum $u_i$ and $c_i$ is

\[ c_i = p_{34}p_{33}p_{32}, \]
\[ u_i = p_{34}p_{33}p_{32} \cdots p_{21}p_{10}. \]

Since the digit register $d_i$ uses 34 bits, the interim sum needs to sign extend one bit before storing the result.

5.2.1 Conversion from GSD to IEEE 754

After the end of the accumulation process, all digit registers $d_i$ need to be converted to the IEEE 754 format, where the significand is in sign-magnitude and the exponent is in a bias representation. The conversion of the redundant representation to a traditional representation involves writing the current digit in the new digit set and a carry into the next most significant digit.

Example 5.2.2. Using a radix-32 number with digit set $d_i = [-16, 64]$, for the redundant number $2\ 33\ -3\ 16\ 1\ -10\ 50$, the conversion to the digit set $d_i = [0, 31]$ is:

Table 5.6: Conversion to radix-32

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>33</td>
<td>-3</td>
<td>16</td>
<td>1</td>
<td>-10</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>-3</td>
<td>16</td>
<td>1</td>
<td>-9</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>33</td>
<td>-3</td>
<td>16</td>
<td>0</td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>29</td>
<td>16</td>
<td>0</td>
<td>23</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>29</td>
<td>16</td>
<td>0</td>
<td>23</td>
<td>18</td>
</tr>
</tbody>
</table>

Rewrite 50 = 32 (carry 1) + 18
Rewrite -9 = -32 (carry -1) + 23
Rewrite -3 = -32 (carry -1) + 29
Rewrite 32 = 32 (carry 1) + 0
Answer: all digits in radix-32

Generally the conversion of a GSD redundant system to a traditional representation is

\[
\text{Final Number} = r^{-\text{OLSD}} \sum_{i=0}^{k-1} (d_i \times r^i), \quad (5.22)
\]

where $r$ is the radix of the redundant system, OLSD is the Order of the Least Significant Digit, and $k$ is the number of digits.

Example 5.2.3. Continuing Example 5.2.2 and applying Equation 5.22 to $2\ 33\ -3\ 16\ 1\ -10\ 50$:

\[
\text{Final Number} = r^{-\text{OLSD}} \sum_{i=0}^{k-1} (\text{digit}_i \times r^i) =
\]
\[
= 32^9(50 \times 32^0 + -10 \times 32^1 + 1 \times 32^2 + 16 \times 32^3 + -3 \times 32^4 + 33 \times 32^5 + 2 \times 32^6)
\]
\[
= 3\ 252\ 159\ 218_{10},
\]

which is the same result of Example 5.2.2 but in radix-10.
Equation 5.22 is applicable to the segmented accumulator, the OLSD using floating-point is \((2l + 2 + 2\epsilon_{\text{min}})/y\) (Equation 2.4), where \(l\) is the significand size with the hidden bit, \(\epsilon_{\text{min}}\) is the minimum exponent and \(y\) is the segment size. The numerator will be referred to as OLSB (Order of Least Significant Bit), for the remainder of this document. The conversion from GSD to IEEE 754 follows these steps:

1. each digit is multiplied by its radix order (shift);
2. accumulate all digits;
3. convert from two’s complement to sign-magnitude.

The block diagram of the conversion unit is shown in Figure 5.2.

Figure 5.2: Block Diagram of the conversion unit

The size of the conversion accumulator and incrementer is

\[
\text{Conversion Accumulator Size} = \text{size}(d_i) + y(k - 1) + 1,
\]

where \(k\) is the number of digits and \(y\) is the segment size. Prior to the accumulation each digit shifts by a given amount. The shifter has a maximum left shift of \(y(k - 1) - \text{OLSB}\) and a maximum right shift of \(\text{OLSB}\). Thereafter, the chosen rounding method is applied to the final register and the exponent is processed using Equation 5.24, where LZD is the result of the LZD on the accumulation result. If Equation 5.24 results in a number equal or less than zero, then the final floating-point will be subnormal.

\[
\text{Final Biased Exp.} = \text{size}(d_i) + y(k - 1) - (2l - 1) - \text{bias} - \text{LZD} - 1
\]

Example 5.2.4. Consider a segmented accumulator using an 8-bit floating-point (four bit significand and three bit exponent) and an 8-bit segment size, and has three 10-bit digit registers \(d_i\). The digit registers \(d_2, d_1\) and \(d_0\) have the values 4, -5 and 15, respectively. The OLSB for an 8-bit floating-point is 16. Substituting in Equation 5.22 yields

\[
\text{Final Number} = 2^{8 \times -16/8} \sum_{i=0}^{2} (d_i \times 2^{8 \times i}) = d_0 \times 2^{-16} + d_1 \times 2^{-8} + d_2 \times 2^0.
\]

The maximum left shift is 0 and the maximum right shift is 16. From Equation 5.23 the conversion accumulator and incrementer size is

\[
\text{Conversion Accumulator Size} = \text{size}(d_i) + y(k - 1) + 1 = 10 + 8 \times 2 + 1 = 27 \text{ bits}.
\]
The shift and accumulation result is in Figure 5.3.

<table>
<thead>
<tr>
<th>$d_2$</th>
<th>000 0000 0100.0000 0000 0000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>111 1111 1111.1111 1111</td>
</tr>
<tr>
<td>$d_0$</td>
<td>000 0000 0000 0000 0000 1111</td>
</tr>
</tbody>
</table>

Figure 5.3: Shift and Accumulation result of the three digits, $d_0$, $d_1$, and $d_2$

From Equation 5.24 the final exponent is

$$\text{Final Biased Exp.} = \text{size}(d_i) + y(k - 1) - (2l - 1) - \text{bias} - \text{LZD} - 1 =$$

$$= 10 + 8 \times 2 - (2 \times 5 - 1) - 3 - 9 - 1 =$$

$$= 4.$$ 

The final result in 8-bit floating-point is 0 100 1111.

Example 5.2.5. A segmented accumulator architecture which supports GSD numbers and the IEEE 754 single-precision format uses: $L = 600$ bits, segment size $y = 32$ bits and carry digit set $c_i = [-2, 2]$.

The OLSB is

$$2l + 2 + 2|e_{\min}| = 2 \times 24 + 2 + 2 \times 126 = 302.$$ 

Each digit requires $\text{size}(d_i) = 34$ bits and there are $\lceil 600 / 32 \rceil = 19$ digit registers. Thus, the size of the conversion accumulator and incremenetr is (from Equation 5.23)

$$\text{Conversion Accumulator Size} = 34 + 32 \times (19 - 1) + 1 = 611 \text{ bits.}$$

The maximum right and left shift performed are 302 bits and 274 bits, respectively.

Equation 5.22 dictates that all digits accumulate and shift. Therefore, for larger floating-point formats, the number of digits required increases, which increases the size of the conversion accumulator and the shifter, e.g., the IEEE 754 single-precision format requires a conversion accumulator of 611 bits, and a maximum right and left shift of 302 bits and 274 bits, respectively (Example 5.2.5). However, the final significand only requires the most significant bits of the conversion accumulation result, because the rounding method used by the exact inner product core is truncation. Thus, if the conversion accumulation only focuses on the digit registers, which are representable by the format, and the size of the significand, the size of the conversion accumulator and shifter can be reduced.

Figure 5.3 from Example 5.2.4 shows there is a two bit overlap per digit caused by the redundancy in each digit. The two MSBs of a given digit $d_i$ add with the LSBs of digit $d_{i+1}$, the remaining bits are a sign extension. In fact, obtaining the final significand from Example 5.2.4 only requires the two most significant digits ($d_2$ and $d_1$), as shown in Figure 5.4.

Since most of the digit overlap is a sign extension, it is possible to use a small subset of all digits, to process the final significand, without using the remaining least significant digits. This process is the same.
Figure 5.4: Shift and Accumulation result of the two digits, $d_1$ and $d_2$. The final significand is underlined on the accumulation result.

as the segmented accumulator algorithm but reversed. The segmented accumulator splits a summand into segments and adds its contents to specific digits in the register file, whereas the conversion unit reads a subset of digits and processes a small part of the final result.

For this type of optimisation to work, certain restrictions are applied to the digit set in the redundant system. Consider the conversion unit selects $w$ digits to convert, where $sel$ is the ID of the most significant non-zero register. The redundant system must guarantee that when performing the conversion the final significand and its sign do not depend on the least significant digits, i.e., converting only $w$ digits guarantees the same result as using all digits for that portion of the final number. To ensure that the conversion unit only requires the selected digits for the final number, the summation of all least significant digits cannot be greater than the final number (Equation 5.25).

$$\sum_{i=0}^{w} \left| d_{sel-i} \right| \times 2^y (sel-i) > \sum_{j=0}^{sel-w-1} \left| d_j \right| \times 2^y j$$

The worst case scenario occurs when digit $sel$ contains the smallest positive digit value and the remainder $w$ digits contain zero, and the remaining least significant digits contain the largest negative digits, i.e., register $sel$ stores 1 and the remaining $w$ digits store zero, and least significant digits store $\alpha$. Substituting in Equation 5.25 results in

$$2^y \times sel > \sum_{j=0}^{sel-w-1} \alpha \times 2^y j.$$ (5.26)

or when digit $sel$ contains the smallest negative number and the remaining digits contain the largest positive number, i.e., register $sel$ stores $-1$ and remaining $w$ digits store zero, and the least significant digits store $\beta$. Substituting in Equation 5.25 results in

$$2^y \times sel > \sum_{j=0}^{sel-w-1} \beta \times 2^y j.$$ (5.27)

Solving in function of $\alpha$ and $\beta$ Equations 5.26 and 5.27

$$\alpha, \beta < \frac{2^y \times sel}{\sum_{j=0}^{sel-w-1} \left(2^y j\right)}.$$ (5.28)

Therefore, the limits for each redundant digit are

$$2^y + \lambda \leq \alpha < \frac{2^y \times sel}{\sum_{j=0}^{sel-w-1} \left(2^y j\right)}.$$ (5.29)

and

$$2^y + \mu - 1 \leq \beta < \frac{2^y \times sel}{\sum_{j=0}^{sel-w-1} \left(2^y j\right)}.$$ (5.30)
The digit set $d_i$ limits derive from the carry digit set $c_i$ and segment size $y$. Thus, to ensure that Equations 5.30 and 5.29 are satisfied

\[
\lambda < \frac{2^{y \times \text{sel}}}{\sum_{j=0}^{w-1} (2^y)} - 2^y
\]

(5.31)

and

\[
\mu < \frac{2^{y \times \text{sel}}}{\sum_{j=0}^{w-1} (2^y)} - 2^y - 1.
\]

(5.32)

Therefore, for the conversion unit to operate over a subset of digits, the carry set $c_i = [-\lambda, \mu]$ of the redundant system must verify Equations 5.31 and 5.32.

It is known which digit register $d_i$ contains the point. The OLSB parameter is the order of the least significant bit in memory, i.e., bit number $\text{OLSB} + 1$ is bit order zero, which means the point is situated between $\text{OLSB} + 1$ and $\text{OLSB}$. Since bit order zero is in bit number $\text{OLSB} + 1$, then the range which can be encoded in a given floating-point format is between $\text{OLSB} + 1 + e_{\text{max}}$ and $\text{OLSB} + 1 - |e_{\text{min}}| - l$. Therefore, there are $e_{\text{max}} + |e_{\text{min}}| + l$ bits that can be encoded in the final format. If there is a bit in a position greater than $\text{OLSB} + 1 + e_{\text{max}}$, the final result is infinity. If there is a bit in a position that is less than $\text{OLSB} + 1 - |e_{\text{min}}| - l$ and all other more significant registers are zero, the final result is a subnormal.

The location of each bit in the register file is given by

\[
\text{Register}_{FP} = \lfloor \text{Bit Number}/y \rfloor.
\]

(5.33)

Thus, the size of the conversion accumulator and incrementer is

\[
\text{Conversion Accumulator Size} = \text{size}(d_i) + y(MSED - LSED - 1) + 1,
\]

(5.34)

where $MSED$ is the most significant encodable digit and $LSED$ the least. Hence, the number of digits $w$ required by the conversion unit is

\[
w = MSED - LSED + 1.
\]

(5.35)

The most significant encodable register is $\lfloor (\text{OLSB} + 1 + e_{\text{max}})/y \rfloor$ and the least significant encodable register is $\lfloor (\text{OLSB} + 1 - |e_{\text{min}}| - l)/y \rfloor$. As a result, the maximum left and right shifts performed are $y\lfloor (\text{OLSB} + 1 + e_{\text{max}})/y \rfloor - \text{OLSB}$ and $y\lfloor (\text{OLSB} + 1 - |e_{\text{min}}| - l)/y \rfloor - \text{OLSB}$, respectively. Thus, the final exponent only takes into account the encodable registers. Adapting Equation 5.24 to only take into account the encodable registers,

\[
\text{Final Biased Exp.} = \text{size}(d_i) + y(k - 1) - (2l - 1) - \text{bias} - yr - LZD - 1,
\]

(5.36)

where $\sigma$ is the number of registers, more significant than the currently selected registers and within the register encodable range, which are zero.

Example 5.2.6. In Example 5.2.5 architecture, the digit set $d_i = [-2^{32} + 2, 2^{32} + 1]$ is within the
Continued

boundaries set by Equations 5.30 and 5.29:

\[
2^{32} + 2 \leq \alpha < \frac{2^{32} \times 13}{\sum_{j=0}^{10} (2^{y_j})},
\]

\[
2^{32} + 1 \leq \beta < \frac{2^{32} \times 13}{\sum_{j=0}^{10} (2^{y_j})}.
\]

Therefore the optimisation described can be applied to the redundant system. The fixed point is between bits 303 (bit 0) and 302 (bit −1). The highest encodable bit number is 430 and the lowest encodable bit number is 153. Thus, the registers that contain the encodable final result is in the range \([153/32], [430/32]\) = [4, 13]. The number of digits required are (from Equation 5.35)

\[
w = MSED - LSED + 1 = 13 - 4 + 1 = 10.
\]

The accumulator’s width is (from Equation 5.34)

\[\text{Conversion Accumulator Size} = \text{size}(d_i) + y(13 - 4 - 1) + 1 = 355 \text{ bits}.\]

The maximum right and left shift are 174 and 114, respectively.

From Example 5.2.6, the conversion unit requires an accumulator of 355 bits and to read ten registers. This is an improvement from the original 611 bit accumulator and nineteen registers. Furthermore, the shift improves from a \([-302, 274]\) shift bit range, to the shift bit range \([-174, 114]\). The size of the conversion accumulator and shifter can be further improved.

The size of the significand is smaller than the size of the accumulator. Thus, the width of the accumulator can be reduced to target a specific significand size while maintaining the same precision. The minimum number of digits \(w\) required for a specific significand size are

\[
w = \left\lceil \frac{\text{size}(l)}{\text{size}(d_i)} \right\rceil + 2. \tag{5.37}
\]

Therefore, the minimum adder width is

\[\text{Conversion Accumulator Size} = (w - 1) \times \text{size}(d_i) - (w - 2) \times 2 + 1. \tag{5.38}\]

The digits used in the accumulation are the most significant \(w\) registers different than zero and inside the range \([\lfloor OLSB + 1 - |e_{\min}| - l\rfloor/y], \lfloor OLSB + 1 + e_{\max}/y\rfloor\]. Reducing the size of the accumulator also allows the size of the shift to decrease. Expanding Equation 5.22 shows the shift difference between digits \(d_i\) is equal to the segment size \(y\). Thus, the shifter performs right shifts of \(y\) bits for \(w\) digits, where the first digit does not shift, the second digit shifts \(y\), the third digit shifts \(2y\), etc.

Example 5.2.7. Continuing the optimisation from Example 5.2.6, the minimum number of digits
Continued

\( w \) required by the conversion accumulator is (from Equation 5.37)

\[
\frac{24}{34} + 2 = 3.
\]

The size of the accumulator is (from Equation 5.38)

\[3 \times 34 - 1 = 67 \text{ bits}.
\]

Due to the usage of three digits, the maximum right shift is 64 bits.

Finally, the improved conversion stage uses a 67 bit accumulator, a 64 bit right shift, and three digits from the register file. Concluding, it is possible to have a smaller conversion stage while maintaining the same level of precision for arbitrary sized floating-points and segments.

### 5.3 Adding GSD Support to the Exact Inner Product Core

The basic architecture for the inner product core described in section 2.4 requires changes to the segmented accumulator and the conversion stage to add GSD support. The conversion unit needs to be updated to support the conversion from GSD digits to sign-magnitude, as described in subsection 5.2.1. In 5.6 the GSD conversion unit is split into two pipeline stages. The first stage converts from GSD to two's complement ("GSD to 2's complement" block), and the second converts from two's complement to sign-magnitude ("2's complement" block). The third stage of the conversion unit processes the exponent, and, the final, fourth stage normalizes the significand and builds the final floating-point number. The proposed segmented accumulator changes are specific to each architecture, and will be described in the following two subsections. The final proposed exact inner product architecture is shown in Figures 5.5 and 5.6.

---

**Figure 5.5: Block Diagram of the Multiplication and Accumulation Stages in the Inner Product Core**
5.3.1 Stalling Segmented Accumulator

The changes required for the stalling segmented accumulator (chapter 3) to support GSD numbers are: setting the size of the adders in the segmented accumulators equal to the position sum $p_i$, setting the size of the registers equal to the digit size $d_i$, setting the size of the carry registers equal to the carry size $c_i$, and updating the carry detection mechanism to support one’s complement and multi-bit carries. Example 5.2.1 shows how to resize the segmented accumulator for the position sum $p_i$, the size of each digit register $d_i$ and the size of the carry registers $c_i$. Furthermore, the carry detection mechanism should have the same output as the previous stalling architecture as to reuse the same algorithms to select the start digit of the carry propagation, and the stall predictor. GSD numbers are designed to withstand one carry propagation from the previous digit without generating another carry. In a situation where the carry propagation overflows, the carry out must be added into the carry digit of the next digit. Since a carry is only propagated on overflow, the new stop stalling condition, by order of priority, is:

1. Segment selector overflow;
2. The CPB does not overflow and the minimum number of predicted stalls is over;

5.3.2 Non-Stalling Segmented Accumulator

The changes required for the non-stalling segmented accumulator (chapter 4) to support GSD numbers are the same as the stalling segmented accumulator, except in the carry register $c_i$. In this architecture, to avoid stalling, the carry digit $c_i$ must be equal to the size of the carry register size in Equation 4.5. The carry selection in the SF stage, adders in the EX stage and the delete carry blocks in the EX and WB stages need to support carries with more than one bit.

5.4 Conclusion

The GSD representation allows processing signed numbers without propagating a sign every cycle to all more significant digits. The usage of GSDs demands, in general, minimal modifications to both architectures. In the Stalling Segmented Accumulator, the carry detection needs to add support for multi-bit carries and one’s complement, and the stop stalling condition changes to only propagate a carry when an overflow occurs. The Non-Stalling Segmented Accumulator only change is in supporting multi-bit carries throughout the accumulator. More specifically, the carry selection of the SF stage, the carry in of the adders in the EX stage, and the delete carry blocks need to support a multi-bit carry.
The conversion from [GSD] to [IEEE] 754 requires an accumulation and shift between all digits of the system. Adding a set of restrictions to the carry digit set \( c_i \) allows the conversion unit to operate over a small set of digits, which improves its resources used and critical path.

Therefore, using [GSDs], the segmented accumulator architectures are able to perform signed-additions with minimal modifications.
Chapter 6

Exact Inner Product Core Results Analysis

This chapter shows and analyses the FPGA implementation results for each proposed inner product core architecture.

The first section shows and analyses the FPGA implementation results for three FPGAs, of increasingly complexity, using three different segment sizes in each architecture. The parameters analysed are the critical path and the resource usage.

The second section tests the inner product core with vectors which are hard to process using traditional floating-point arithmetic. The inner product core results are then compared with traditional floating-point arithmetic results and with results calculated with arbitrary precision arithmetic.

The third section uses the proposed inner product core in a credit risk analysis example using a generalised linear model to predict which customers will default. At the end, the results obtained with the inner product core are compared with traditional floating-point arithmetic results.

6.1 FPGA Implementation Results

Both architectures were synthesized using Vivado 2016.3 for three Zynq-7000 devices: Z7010-1 (Artix-7), Z7020-1 (Artix-7) and Z7045-2 (Kintex-7). The non-default synthesis options used were: flatten hierarchy is set to “rebuilt”, register retiming is enabled and resource sharing is off. Each architecture was implemented for the most used IEEE 754 floating-point formats: 16-bit, 32-bit and 64-bit. Furthermore, each floating-point format used was implemented with three segment sizes: 8 bits, 16 bits and 32 bits. The three segment sizes were selected because the analysis performed in the previous chapters predicts the best resource usage and critical path among them. Both architectures use a register file to store the digits, use the truncation rounding method, and use 100 bits of overflow protection. Furthermore, the multipliers are pipelined for each floating-point size using Vivado’s recommendations. One pipeline register in the 16-bit floating-point multiplier, two pipeline registers in the 32-bit floating-point multiplier, and ten pipeline registers in the 64-bit floating-point multiplier.

Since the Z7020-1 and Z7045-2 FPGAs have more resources and routing available, the retiming algorithm will take advantage of them, hence resulting in a higher clock frequency. Finally, the Z7045-2 FPGA has a -2 speed grade, which means the clock frequencies for this device will be higher.
6.1.1 Stalling Segmented Accumulator Architecture

The Stalling Segmented Accumulator architectures uses a max carry digit of two. The clock frequencies and resources used for each floating-point size and segment size are shown in Table 6.1 and Table 6.2 respectively.

Table 6.1: Clock Frequency for the Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the clock frequency for three different segment sizes \( y \): 8 bits, 16 bits and 32 bits.

<table>
<thead>
<tr>
<th>FP Size</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y = 8</td>
<td>y = 16</td>
<td>y = 32</td>
</tr>
<tr>
<td>Z7010-1</td>
<td>107</td>
<td>111</td>
<td>116</td>
</tr>
<tr>
<td>Freq. (MHz)</td>
<td>65</td>
<td>68</td>
<td>74</td>
</tr>
<tr>
<td>Z7020-1</td>
<td>107</td>
<td>117</td>
<td>123</td>
</tr>
<tr>
<td>Freq. (MHz)</td>
<td>65</td>
<td>77</td>
<td>90</td>
</tr>
<tr>
<td>Z7045-2</td>
<td>200</td>
<td>227</td>
<td>227</td>
</tr>
<tr>
<td>Freq. (MHz)</td>
<td>107</td>
<td>158</td>
<td>172</td>
</tr>
</tbody>
</table>

Table 6.2: Resource Usage for the Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the resource usage for three different segment sizes \( y \): 8 bits, 16 bits and 32 bits.

<table>
<thead>
<tr>
<th>FP Size</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>y = 8</td>
<td>y = 16</td>
<td>y = 32</td>
</tr>
<tr>
<td>Z7010-1</td>
<td>1259</td>
<td>1043</td>
<td>1179</td>
</tr>
<tr>
<td>LUTs</td>
<td>6979</td>
<td>4997</td>
<td>4044</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>1538</td>
<td>1370</td>
<td>1344</td>
</tr>
<tr>
<td>DSPs</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Z7020-1</td>
<td>1263</td>
<td>1051</td>
<td>1182</td>
</tr>
<tr>
<td>LUTs</td>
<td>6978</td>
<td>4992</td>
<td>4062</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>1538</td>
<td>1370</td>
<td>1344</td>
</tr>
<tr>
<td>DSPs</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Z7045-2</td>
<td>1281</td>
<td>1097</td>
<td>1269</td>
</tr>
<tr>
<td>LUTs</td>
<td>6980</td>
<td>5033</td>
<td>4132</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>1538</td>
<td>1370</td>
<td>1343</td>
</tr>
<tr>
<td>DSPs</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The critical path for a 16-bit floating-point Stalling Segmented Accumulator is either in the conversion unit or in the CPB. Since the 16-bit floating-point architecture has a small long register size \( L = 80 \) bits, having a smaller segment size \( y \) increases the number of registers the conversion unit can select from. Thus, for smaller segment sizes the critical path is in the conversion unit, and for bigger segment sizes the critical path is in the CPB.

The critical path for a 32-bit floating-point Stalling Segmented Accumulator architecture is identical to the 16-bit architecture, in that it is either in the conversion unit or in the CPB.

The 64-bit architecture requires a large number of resources, thus it cannot be implemented in the Z7010-1 FPGA. As such, it was only implemented in the Z7020-1 and Z7045-2 FPGAs, where the former can only implement the configurations with segment size \( y \) of 16 bits or 32 bits. The critical path obtained for a 64-bit architecture and 8-bit segment size is in the conversion unit, and for the 16-bit and 32-bit segment sizes the critical path is in the CPB. Both paths are dominated by the register selection because the long accumulator size \( L \), for a 64-bit format, is in the 4000 bit range. The 64-bit architecture requires a higher number of register than the 16-bit and 32-bit architectures, which will, consequently, increase the size of the output MUXs in the register file.

In the analysis performed the best segment size, for the lowest resource usage and highest clock frequency, for all floating-point formats is 32 bits.
6.1.2 Non-Stalling Segmented Accumulator Architecture

The max carry digit used by the Non-Stalling Segmented Accumulator architectures is the minimum carry value to avoid stalling. The minimum carry values and their sizes are in Table 6.3. The clock frequencies and resources used for each floating-point size and segment size are shown in Table 6.4 and Table 6.5, respectively.

Table 6.3: Maximum and minimum carries, and their register size to avoid stalling

<table>
<thead>
<tr>
<th>FP Size</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg. Size</td>
<td>y = 8</td>
<td>y = 16</td>
<td>y = 32</td>
</tr>
<tr>
<td>Max/Min Carry</td>
<td>± 8</td>
<td>± 4</td>
<td>± 2</td>
</tr>
<tr>
<td>Carry Size (bits)</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.4: Clock Frequency for the Non-Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the clock frequency for three different segment sizes $y$: 8 bits, 16 bits and 32 bits.

<table>
<thead>
<tr>
<th>FP Size</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg. Size</td>
<td>y = 8</td>
<td>y = 16</td>
<td>y = 32</td>
</tr>
<tr>
<td>Z7010-1 Freq. (MHz)</td>
<td>96</td>
<td>100</td>
<td>111</td>
</tr>
<tr>
<td>Z7020-1 Freq. (MHz)</td>
<td>96</td>
<td>103</td>
<td>111</td>
</tr>
<tr>
<td>Z7045-2 Freq. (MHz)</td>
<td>192</td>
<td>212</td>
<td>238</td>
</tr>
</tbody>
</table>

Table 6.5: Resource Usage for the Non-Stalling Segmented Accumulator architecture using three IEEE 754 Floating-Point formats: 16 bits, 32 bits, and 64 bits. Each format shows the resource usage for three different segment sizes $y$: 8 bits, 16 bits and 32 bits.

<table>
<thead>
<tr>
<th>FP Size</th>
<th>16 bits</th>
<th>32 bits</th>
<th>64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seg. Size</td>
<td>y = 8</td>
<td>y = 16</td>
<td>y = 32</td>
</tr>
<tr>
<td>Z7010-1</td>
<td>LUTs</td>
<td>2449</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>Flip-Flops</td>
<td>860</td>
<td>763</td>
</tr>
<tr>
<td></td>
<td>DSPs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z7020-1</td>
<td>LUTs</td>
<td>2444</td>
<td>1899</td>
</tr>
<tr>
<td></td>
<td>Flip-Flops</td>
<td>860</td>
<td>763</td>
</tr>
<tr>
<td></td>
<td>DSPs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z7045-2</td>
<td>LUTs</td>
<td>2531</td>
<td>1967</td>
</tr>
<tr>
<td></td>
<td>Flip-Flops</td>
<td>860</td>
<td>763</td>
</tr>
<tr>
<td></td>
<td>DSPs</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The 16-bit Non-Stalling Segmented Accumulator architecture has the critical path either in the conversion unit, or in the FFA digit selection. The number of digits required by the conversion unit and the number of digits that the FFA can select are directly proportional to the number of registers in the register file. Since the number of registers is inversely proportional to the segment size $y$, increasing the segment size $y$ will decrease the number of registers. Therefore, for larger segment sizes $y$, the conversion unit requires less digits, and the amount of digits the FFA can select from is smaller.

The critical path in the 32-bit Non-Stalling Segmented Accumulator architecture is, for all segment sizes $y$, in the delete carry mechanism of the FFA in the WB stage. The critical path starts at the register selection in the SF stage, goes through the forwarding path control unit, and through the FFA carry delete mechanism and ends at the carry accumulation in the register file. Since having a larger segment size reduces the size of the carry, the forwarding path control unit, the carry delete logic and the size of the accumulator are smaller.
The 64-bit architecture requires a large number of resources, thus it cannot be implemented in the Z7010-1 FPGA. Likewise, for the Z7020-1, only the configuration with a 32-bit segment size is implemented successfully. Similarly to the 16-bit and 32-bit floating-point architectures, the critical path for 8-bit segments is in the conversion unit, and for 16-bit and 32-bit segments is in the delete carry mechanism of the FFA.

In the analysis performed the best segment size, for the lowest resource usage and highest clock frequency, for all floating-point formats is 32 bits.

### 6.1.3 Stalling and Non-Stalling Architectures Comparison

The ratios of the clock frequency and Look Up Tables (LUTs) usage between the non-stalling and stalling architectures for the 32-bit segment size for three floating-point sizes is shown in Figure 6.1. Figure 6.1a shows that the frequency has a minimal variation between both architectures. Figure 6.1b shows that the LUTs used by the non-stalling architecture decreases with the size of the floating-point, i.e., as the size of the floating-point increases the resources used by the forwarding paths are not as significant as the resources used by the register file.

Therefore, the non-stalling architecture has better performance for a higher resource usage, and the stalling architecture has lower performance for a lower resource usage.

(a) Clock frequency ratio

(b) LUTs ratio

Figure 6.1: Clock frequency and LUTs ratio between the non-stalling and stalling architectures using a 32-bit segment size, for three floating-point sizes: 16 bits, 32 bits and 64 bits

### 6.1.4 Performance and Resources Analysis

The proposed single-precision architectures were also compared with a single-precision Multiply-Accumulator (MAC) unit (Figure 6.2), composed by one multiplier and one accumulator both implemented using the Xilinx Floating-Point operator (Version 7.1), and with the single-precision floating-point unit present in the ARM Cortex-A9. The test vector used was the One Large Many Small, from subsubsubsection 2.2.1.1.1, adapted for a size of 10M.

Figure 6.2: The single-precision Floating-Point Multiply-Accumulate unit is divided in two blocks, one multiplier and one accumulator.
The number of cycles obtained for the MAC unit and the proposed architectures is the same, approximately 20M cycles (the vectors are stored interleaved in memory and the DMA sends 32-bit words, i.e., it takes two cycles to read two inputs). However, as expected, the MAC result has an accuracy penalty, the result obtained is 1.00, whereas the proposed architectures do not, which compute the correct result 1.0000001. Comparing the proposed architectures with the single-precision floating-point unit of the ARM (with compiler level 2 optimisations), shows a 6.85 times speedup and better accuracy. Using Neon units alongside the ARM, the proposed architectures have a 3.5 times speedup. The MAC unit and the ARM FP unit have the same final result accuracy.

The resources used in the single-precision proposed architectures (using a 32-bit segment size) and the single-precision MAC unit are shown in Table 6.6.

### Table 6.6: Resources used by the proposed single-precision architectures (stalling and non-stalling) and the single-precision Multiply-Accumulator (MAC) unit

<table>
<thead>
<tr>
<th></th>
<th>Stall</th>
<th>Non-Stall</th>
<th>MAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUTs</td>
<td>4044</td>
<td>6105</td>
<td>2097</td>
</tr>
<tr>
<td>Flip-Flops</td>
<td>1344</td>
<td>2420</td>
<td>702</td>
</tr>
<tr>
<td>LUTRAMs</td>
<td>0</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>DSPs</td>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

### 6.2 Evaluation Results

This section compares and analyses the results for two examples using the exact inner product core, traditional floating-point arithmetic and arbitrary precision arithmetic. The first tests focus on the precision of each arithmetic unit using vectors for which traditional floating-point arithmetic has trouble with. The second test uses the exact inner product core in a credit risk analysis example.

The results for traditional floating-point single-precision and double-precision arithmetic were obtained executing a C program in a GPP and for arbitrary precision were obtained executing bc (39) script with a precision of 100 decimal places. All numbers used in the exact inner product core use the IEEE 754 single-precision format, and were processed using a segmented accumulator with 32-bit segments and 100 bits of overflow protection. The inner product core was implemented in the Zybo board using the Zynq DMA to directly access the external memory. The vectors were stored interleaved in the external memory, i.e., address zero has $x_0$, address one has $y_0$, address two has $x_1$, address three has $y_1$, etc. The system is shown in Figure 6.3.

### 6.2.1 Accuracy Tests Results

In this subsection, the inner product core is evaluated using the hard to compute test sets described in subsubsubsection 2.2.1.1 but adapted for single-precision. The test sets selected are:
- Taylor series expansion for $e^{-2\pi}$, for 35 terms;
- Heavy Cancellation;
- sum equally spaced on $[1, 2]$ for $\alpha = 4096$;
- $\sum 1/i^2$, for the range $[1, 4096]$;
- One Large, many small.

The results obtained for arbitrary precision (using bc), the single-precision exact inner product core and the traditional arithmetic (using C) are in Table 6.7. The programs used to get the arbitrary precision results in bc and traditional arithmetic results in C are in Appendix A.
Table 6.7: Result comparison between Arbitrary Precision, the Single-Precision Exact Inner Product Core (EIPC) and Traditional Single-Precision Arithmetic

<table>
<thead>
<tr>
<th>Type of Data</th>
<th>Arbitrary Precision (100 d.)</th>
<th>EIPC (SP)</th>
<th>Traditional Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^{-25}$</td>
<td>$1.873410 \times 10^{-3}$</td>
<td>$1.873410 \times 10^{-3}$</td>
<td>$1.877666 \times 10^{-3}$</td>
</tr>
<tr>
<td>Heavy Cancellation</td>
<td>$2.0 \times 10^{-18}$</td>
<td>$2.0 \times 10^{-18}$</td>
<td>$0.000000$</td>
</tr>
<tr>
<td>Equal(1,2)</td>
<td>6143.500000</td>
<td>6143.500000</td>
<td>6143.366210</td>
</tr>
<tr>
<td>$\sum \frac{1}{i^2}$</td>
<td>1.644689</td>
<td>1.644689</td>
<td>1.644725</td>
</tr>
<tr>
<td>One large, many small</td>
<td>1.999999</td>
<td>1.999999</td>
<td>1.953673</td>
</tr>
</tbody>
</table>
Figure 6.3: System used to test each vector
These results show that the exact inner product core successfully computes all tests exactly, using the same inputs as the traditional arithmetic. Conversely, the traditional arithmetic results fail all tests because of rounding errors between accumulations and incorrectly inferring zero when adding a small number to a large number.

6.2.2 Credit Risk Analysis Example

This subsection evaluates the exact inner product core using a real world credit risk analysis example. Credit risk analysis is performed by banks to assert if a customer should be allowed to take a loan. Most importantly, banks want to know if the customer will pay its loan (not-default) or does not pay it (default), a parameter referred to as probability of default. To obtain the probability of default, banks use machine learning to examine the previous and current financial information of the customer, and try to predict the customer’s behaviour.

These type of problems are commonly referred to, in machine learning, as classification problems. In credit risk analysis, two methodologies can be used: Generalized Linear Model (GLM) or decision trees. Since GLM is the simpler method of the two, this example will use it. More specifically, and because the output is a binary response, a logistic regression is used. The model fitting used for the logistic regression is the Iteratively Reweighted Least Squares (IRLS).

The dataset used (from [40]) contains personal and financial information, a total of 24 parameters, from 30 000 Taiwanese customers, and whether or not they defaulted. The 24 parameters in the dataset are:

- Limit Balance;
- Gender;
- Education;
- Marital Status;
- Age;
- Repayment Status in Sept/Aug/July/June/May/April, 2005;
- Amount of Bill Statement in Sept/Aug/July/June/May/April, 2005;
- Amount Paid in Sept/Aug/July/June/May/April, 2005;
- Default.

The training set used contains 300 customers selected at random. Furthermore, all variables in the dataset are represented exactly using the IEEE 754 single-precision format, so it is reasonable to assume that all processing should be done using single-precision floating-point arithmetic. The logistic regression was processed using traditional single-precision floating-point arithmetic, the single-precision exact inner product core and double-precision arithmetic. The coefficients obtained for each arithmetic method, and the number of correct digits between the single precision results and double precision results, are in Table 6.8.

Analysing the coefficients results for each arithmetic method shows that the single-precision exact inner product core provides a higher level of accuracy (more correct digits) for most coefficients, using the same inputs as the traditional single-precision arithmetic.

6.3 Conclusion

The analysis of the FPGA implementation results, shows that using a Non-Stalling architecture is justified, at a resource expense, if performance is a critical requirement. Conversely, if there is a resource requirement the Stalling architecture is the best option. From the analysis performed, the best segment
Table 6.8: Coefficients obtained for the Logistic Regression using traditional single-precision arithmetic, the single-precision exact inner product core (EIPC) and traditional double precision arithmetic, and number of correct digits (NCD) for the traditional single-precision arithmetic and the single-precision inner product core

<table>
<thead>
<tr>
<th></th>
<th>Single-Precision</th>
<th>EIPC (SP)</th>
<th>Double Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NCD</td>
<td>Results</td>
<td>NCD</td>
</tr>
<tr>
<td>Intersection</td>
<td>4</td>
<td>$1.686 \times 10^{-1}$</td>
<td>5</td>
</tr>
<tr>
<td>Limit Balance</td>
<td>3</td>
<td>$7.207 \times 10^{-7}$</td>
<td>5</td>
</tr>
<tr>
<td>Gender</td>
<td>5</td>
<td>$-4.611 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Education</td>
<td>4</td>
<td>$-1.446 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Marital Status</td>
<td>4</td>
<td>$-1.076 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Age</td>
<td>3</td>
<td>$4.241 \times 10^{-3}$</td>
<td>6</td>
</tr>
<tr>
<td>Rep. Status in Sept.</td>
<td>6</td>
<td>$5.301 \times 10^{-1}$</td>
<td>7</td>
</tr>
<tr>
<td>Rep. Status in Aug.</td>
<td>6</td>
<td>$3.999 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Rep. Status in July</td>
<td>5</td>
<td>$-1.397 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Rep. Status in June</td>
<td>5</td>
<td>$4.201 \times 10^{-1}$</td>
<td>5</td>
</tr>
<tr>
<td>Rep. Status in May</td>
<td>5</td>
<td>$-7.860 \times 10^{-1}$</td>
<td>6</td>
</tr>
<tr>
<td>Rep. Status in April</td>
<td>5</td>
<td>$1.946 \times 10^{-1}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Stat. in Sept.</td>
<td>5</td>
<td>$-1.820 \times 10^{-5}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Stat. in Aug.</td>
<td>3</td>
<td>$-5.368 \times 10^{-6}$</td>
<td>3</td>
</tr>
<tr>
<td>Amnt. Stat. in July</td>
<td>4</td>
<td>$3.375 \times 10^{-5}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Stat. in June</td>
<td>3</td>
<td>$-2.202 \times 10^{-3}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Stat. in May</td>
<td>4</td>
<td>$3.605 \times 10^{-6}$</td>
<td>6</td>
</tr>
<tr>
<td>Amnt. Stat. in April</td>
<td>5</td>
<td>$-1.501 \times 10^{-5}$</td>
<td>6</td>
</tr>
<tr>
<td>Amnt. Paid in Sept.</td>
<td>5</td>
<td>$-9.274 \times 10^{-5}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Paid in Aug.</td>
<td>4</td>
<td>$-3.167 \times 10^{-5}$</td>
<td>6</td>
</tr>
<tr>
<td>Amnt. Paid in July</td>
<td>4</td>
<td>$3.123 \times 10^{-5}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Paid in June</td>
<td>3</td>
<td>$-9.289 \times 10^{-5}$</td>
<td>5</td>
</tr>
<tr>
<td>Amnt. Paid in May</td>
<td>4</td>
<td>$1.259 \times 10^{-5}$</td>
<td>7</td>
</tr>
<tr>
<td>Amnt. Paid in April</td>
<td>5</td>
<td>$-2.003 \times 10^{-5}$</td>
<td>6</td>
</tr>
</tbody>
</table>

size for both architectures and all floating-point sizes, in terms of clock frequency and resources used, is 32 bits.

The inner product core was tested with vectors which traditional floating-point arithmetic has trouble with. In these tests, the inner product core achieved the same results as the arbitrary precision results which fully validated the proposed architectures. The inner product core was further evaluated using a credit risk analysis example. Therein, the results show that the single-precision exact inner product core provides a significantly better accuracy than traditional single-precision floating-point arithmetic using the same inputs.
Chapter 7

Conclusion

An exact floating-point inner product operation allows the processing of two floating-point vectors without errors. The availability of an exact floating-point inner product operation facilitates a programmers job when implementing a numerical system and improves the results of critical numerical applications in the industry and academia. Most research performed on this subject is in the optimisation of addition with remainder algorithms. Such algorithms do not lend well to hardware implementation, as they require at least three arithmetic operations, and do not guarantee an exact final result.

To keep precision throughout processing, the long accumulator architecture expands the floating-point format to fixed-point, and stores it in memory until the end of processing. This poses an issue to the critical path, since each input uses the entirety of the memory for every accumulation, even when it is not necessary. For the input to only operate over the necessary portions of the memory, the accumulator needs to implement segmented access. The implementation of segmented access in the accumulator splits the memory into smaller memories and the large adder into smaller adders. Most importantly, using segmented access, an input will only load into the adders the parts of the memory which will update, thus the critical path will be shorter.

While using a segmented accumulator shortens the critical path of the architecture, it will add complexity to the carry propagation mechanism as it has to store a carry for every segment. Therefore, after any accumulation, the segmented accumulator must propagate all carries generated. The way in which the carry propagation is performed has an impact on the performance of the final architecture. Thus, two segmented accumulator architectures are proposed with different mechanisms for the carry propagation.

The stalling architecture stops the accumulation when there is at least one carry to propagate. The carry propagation block (CPB) starts at the first segment which will absorb the least significant carry generated, and keeps propagating while there are carries left. The architecture resumes accumulation when there are no more carries to propagate. When optimising the architecture, for the shortest critical path, there is a tradeoff between the number of segments and the size of the CPB. When minimising resource consumption larger segment sizes should be used to have a small register file decoder.

The non-stalling architecture uses a Free Flow Adder (FFA) to autonomously select and propagate carries. Three pipeline stages were included in the accumulator to improve the throughput and shorten the critical path. Furthermore and to cope with the hazards present in the architecture, forwards paths were added to each stage, each segment saves multi-bit carries, and the write carry operation changes to an accumulation. Since the critical path is in the conversion stage and the carry delete mechanism, optimising for the critical path requires balancing the size of the carry register with the number of registers. Similarly to the stalling architecture, optimising for resources used requires a small register file decoder, and a small number of forwarding paths, i.e., a large segment size should be selected.
The inclusion of the subtraction operation is done by taking advantage of GSDs. A redundant GSD number uses signed digits and carries to keep the sign bound to each digit, thus the sign propagation is halted at the next more significant digit. The support for GSDs requires minimal modifications to the architectures. The only modification which requires special care is handling signed multi-bit carries, the remaining accumulation architecture remains the same. Since the final number is in a limited format, processing the final result only requires a small range of numbers in memory. Therefore, the final conversion stage, from redundant representation to the IEEE 754 format, can be simplified by exploiting the limited precision of the final format.

Both architectures were implemented in three Zynq FPGAs for three IEEE 754 formats (16 bits, 32 bits and 64 bits) and three segment sizes (8 bits, 16 bits and 32 bits). From the results it is concluded that the non-stalling architecture provides better performance for more resource usage, and the stalling architecture provides better resource usage for worse performance. Furthermore, the inner product core was also evaluated using test sets which traditional floating-point arithmetic has trouble with, and in a credit risk analysis example. In the first case, the results showed that the proposed architectures can process the inner product of any two vectors exactly. In the second case, the inner product core is able to process the coefficient of a logistic regression more accurately than traditional single-precision arithmetic using the same inputs.

7.1 Future Work

There are four main points that should be the focus of future work:

- make further optimisations to the segmented accumulator architecture;
- analyse different storage methods in the segmented accumulator;
- adhere to the IEEE 754 standard for infinity, and NaN signalling and silent exceptions;
- add support for the round to nearest even rounding method.

Based on the conclusion from the FPGA implementation results (section 6.1, section 6.3), the first optimisation should be in the digit selection of the conversion unit and the FFA/CPB. In the conversion unit, the proposed architectures looks for the most significant non-zero digit every cycle. This mechanism can be improved by keeping track of the most significant summand digit selection and its final computation result, every time there is a write to the register file. More specifically, when a summand writes to the register file, a control unit checks whether the summand's most significant non-zero result ID is greater than the previous state ID and, if the condition is true, the new state ID will be the summand's most significant non-zero result ID. Therefore, instead of searching for the most significant non-zero register every cycle, the conversion unit already knows the register IDs to convert. This may remove the digit selection logic and the requirement for the non-zero status flags in each digit.

Example 7.1.1. Consider a segmented accumulator with three adders and twenty registers. The first summand S1, selects registers seven, eight and nine, and the results obtained for each register are 0, 40 and 0, respectively. Therefore, the new state ID is eight because it was the first non-zero most significant result. Afterwards, a second summand S2 enters the segmented accumulator and selects registers eight, nine and ten. The results for each register are 10, −1, 30, respectively. Thus, the new state ID is ten because it was the first non-zero most significant result and it was greater than the previous state ID. Figure 7.1 shows the steps at every clock cycle.
When the result of all selected digits is zero, then the conversion unit should fall back to the original method of finding the most significant non-zero digit in the register file. This same mechanism can be applied to the digit selection in the FFA/CPB.

Another possible optimisation would be in the GSD to sign-magnitude conversion. Instead of having an LZD after the two’s complement stage, that pipeline stage can be removed and replaced with an Lead Zero Anticipator in the GSD to two’s complement stage. A Lead Zero Anticipator will count the number of zeroes or ones in parallel with the accumulation block.

Both proposed architectures use a register file to store the digits. However, and since the processor is being implemented in an FPGA, the use of Block Random Access Memories (BRAMs) should be evaluated. Doing so would free up a large amount of LUTs used by each configuration, which would be a significant for smaller segment sizes $y$.

Currently, the input of the Inner-Product Core does not check for NaN and infinity. Thus, if one element of the vector is either a NaN or infinity the final result will be wrong. Furthermore, even though the architecture can generate an output as a NaN, it cannot differentiate between its two types: signalling and quiet. Finally, the rounding method should be changed to the IEEE 754 default, rounding nearest to even.
Bibliography


Appendix A

**BC and C Programs used for Accuracy Tests**

C program used to process the tests in subsection 6.2.1.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

float factorial(float x) {
    if (x <= 1.0) return (1.0);
    return (factorial(x - 1.0) * x);
}

int main(void) {
    // Taylor
    float sum = 0.0;
    float x = -2.0 * 3.1415927410125732421875;
    for (int i = 0; i < 35; i++) {
        sum += pow(x, i) / factorial(i);
    }
    printf("Taylor Result: %.12f\n", sum);

    // Heavy Cancellation
    for (int i = 0; i < 4096; i++) {
        if (i < 2047) {
            sum += 1.0;
        } else if (i == 2047 || i == 2048) {
            sum += 1.0e-18;
        } else {
            sum += -1.0;
        }
    }
    printf("Heavy Cancellation Result: %.9f\n", sum);

    // Equal
    x = 1.0/4096.0;
    for (int i = 0; i < 4096; i++) {
        // Code...
    }
}
```
sum += 1.0 + x * (float) i;
}

printf("Equal Result: %.9f\n", sum);
sum = 0.0;

// Higham
for (int i = 1; i < 4097; i++) {
    sum += 1.0/((float) i * (float) i);
}

printf("Higham Result: %.9f\n", sum);
sum = 0.0;

// One Large Many Small
for (int i = 0; i < 1000000; i++) {
    if (i == 0) {
        sum += 1.0 * 1.0;
    } else {
        sum += 1.0e-3 * 1.0e-3;
    }
}

printf("One Large, Many Small Result: %.9f\n", sum);
sum = 0.0;
exit(0);

bc program used to process the tests in subsection 6.2.1.

scale=100
#define factorial(x) {
    if (x <= 1) return (1);
    return (factorial(x-1) * x);
}
#define taylor_e(x, terms) {
    auto sum, y, x_aux;
    sum = 0;
    x_aux[0] = 1.0 ; x_aux[1] = -6.283185482; x_aux[2] = 3.947842025756836 * 10^-1; 
    x_aux[21] = -5.77999916962509 * 10^-18; x_aux[22] = 3.630424186159104 * 10^-19; 
    x_aux[27] = -3.5551404199463627 * 10^-24; x_aux[28] = 2.2337606453778155 * 10^-25; 
    x_aux[29] = -1.4035133170887874 * 10^-26; x_aux[30] = 8.818534459151187 * 10^-27; 
    x_aux[31] = -5.540848517420138 * 10^-28; x_aux[32] = 3.48141803788026 * 10^-29;
x_aux[33] = -2.1874394464300967 * 10^{-26}; x_aux[34] = 1.3744087567947784 * 10^{-27};

y[0] = 1.0; y[1] = 1.0; y[2] = 0.5; y[3] = 0.1666666716337204;
y[4] = 0.041666679084301; y[5] = 0.008333333767950535; y[6] = 0.0013888889225199819;
y[7] = 0.00019841270113829523; y[8] = 2.48015876422866904 * 10^{-05};
y[33] = 1.1516336010911503 * 10^{-37}; y[34] = 3.3871569908335175 * 10^{-39};

for (i = 0; i < terms; i++) {
    # sum += x^i / factorial(i);
    sum += x_aux[i] * y[i];
}
return sum;

#define equal(x, space) {
    auto sum, aux;
    sum = 0;
    aux = 1/4096;
    for (i = 0; i < space; i++) {
        sum += x * aux * i;
    }
    return sum;
}

#define higham(max) {
    auto sum;
    sum = 0;
    for (i = 1; i <= max; i++) {
        sum += 1/i ^2;
    }
    return sum;
}

#define heavy_cancellation() {
    auto sum;
    sum = 0;
    for(i = 0; i < 4096; i++) {
        if(i < 2047){

37}
84  sum += 1;
85  } else if(i == 2047 || i == 2048) {
86    sum += 1.000000045 * 10^(-18);
87  } else {
88    sum += -1;
89  }
90
91  return sum;
92}
93
94 define one_large_many_small() {
95  auto sum;
96  sum = 0;
97
98  for(i = 0; i < 1000000; i++) {
99    if(i == 0) {
100      sum += 1.0 * 1.0;
101    } else {
102      sum += 1.0000000475 * 10^(-3) * 1.0000000475 * 10^(-3);
103    }
104  }
105
106  return sum;
107}
108
109 taylor_e(-6.283185482025146484375, 35);
110 heavy_cancellation();
111 equal(1, 4096);
112 higham(4096);
113 one_large_many_small();