Exact Inner Product Processor in FPGA

Luís Fiolhais
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
luis.azenhas.fiolhais@tecnico.ulisboa.pt

***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.

***Index Terms*—exact inner product, floating-point, long-accumulator, segmented accumulator, FPGA, Generalised Signed Digits**

I. Introduction

The objective of this work is to develop an efficient hardware process toor to perform the inner product operation with full accuracy. In scientific computations high performance floating-point arithmetic is frequently required. In fact, the most used operation sequence is a multiplication followed by an addition, which justified the Institute of Electrical and Electronics Engineers (IEEE) standardisation of a fused multiply-add operation. However, there is still no standard for the inner product operation.

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 partial rounding. From [1], consider two vectors x and y of size $10^{16}+1$ such that

$$x_{0..10^{16}} = 1.0, \text{ and } y_0 = 1.0 \text{ and } y_{1..10^{16}} = 1.0 \times 10^{-16}$$

and whose elements are represented using the IEEE 754 single-precision format, the correct result of the inner product is 2.0 but the result of the computation is 1.0.

Consider also, from [2] two vectors of size five using the IEEE 754 single-precision format

$$\begin{aligned} a = [&2.718281828 \times 10^{10}, -3.141592654 \times 10^{10}, \\ &1.414213562 \times 10^{10}, 5.772156649 \times 10^{9}, \\ &3.010299957 \times 10^{9}] \end{aligned}$$

and

$$\begin{aligned} b = [&1.4862497 \times 10^{12}, 8.783669879 \times 10^{14}, \\ &-2.237492 \times 10^{10}, 4.773714647 \times 10^{15}, 185049]. \end{aligned}$$

The correct result of the inner product is -100657107 but the result of the computation is -2.305286×10^{18}.

These two examples show that the inner product operation should be computed as a single fused operation, and not as two independent operations. Often the precision selected for a numerical system may not adequately fit the computation to be performed if one intermediate result exceeds the precision limits of the representation. 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. For example, using the double-precision format in the previous examples does not guarantee accurate results.

The requirement for accurate inner product floating-point arithmetic is present in research areas such as high frequency trading in financial engineering, the learning mechanism in machine learning and shading in 3D graphics, just to name three. 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.

The following section introduces the state of the art on inner product solutions using floating-point arithmetic. In sections III and IV a stalling and a non-stalling Segmented

Accumulator Architecture are proposed, respectively. Section V presents the carry propagation mechanism to add signed-addition support, efficiently. Section VI presents and analyses Field Programmable Gate Array (FPGA) implementation results for both proposed segmented accumulator architectures using three floating-point formats. Finally, section VII presents conclusions to the work.

II. Inner Product Computation

This section introduces the state of the art on exact inner product solutions using floating-point arithmetic. The inner product is defined by a multiplication followed by an accumulation of all resulting terms:

$$s = \sum_{i=0}^{n} x_i \times y_i. \tag{1}$$

A. Fused Multiply-Add

The closest operation to the inner product in modern processors is the fused multiply-add operation. The fused multiplier-add operation is defined as $(A \times B) + C$, which should be processed without a range and precision limit and with only one rounding [3]. The steps necessary to complete this operation are the following: multiply significands, add exponents, align the result of the multiplication *without rounding or truncating* with C, add the result of the multiplication to C using the adder with the *full length of the multiplication result*, normalize, round, and store the data. Presently, Graphical Processing Units (GPUs) and GPPs implement this operation in hardware, compliant with this specification [4], [5].

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

B. Accumulation Solutions

There are many published solutions about floating-point accumulation [2], [6]–[17]. Two main alternatives can be identified: Addition with Remainder and Long Accumulator.

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 [9] by Pichat (Algorithm 1a). Pichat's algorithm stores each remainder in an array (x') where the first element is an approximation of the final result ($\sum_{i=0}^{n} x_i$) and the remaining elements are the errors of the accumulation for that term [11]. The final corrected result is obtained by accumulating the vector x'. Furthermore, a new vector of remainders can be created (x'') to achieve a better final accuracy. This process can be repeated according to the desired accuracy requirements [11].

Kahan also presented a similar algorithm in [12] (Algorithm 1b). The difference lies in the way the remainder is stored and merged into the final sum [13]. 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.

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

(a) Pichat's algorithm

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

(b) Kahan's Algorithm [12]–[14]

Figure 1: Addition with Remainder Algorithms

Table I: Comparison between all algorithms described using resources, critical path and precision

	Pichat	**Kahan**	**Long Accumulator**
Resources	3 FP Units 1 Memory	4 FP Units	Large Shifter and Accumulator
Critical Path	FP Unit	FP Unit	Accumulator
Accuracy	Very Good	Good	Full

2) Long Accumulator: The Long Accumulator (LA) algorithm computes an accumulation with full accuracy, using full-width fixed-point accumulations [2].

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/smallest possible multiplication result will have a $2e_{max}/2e_{min}$ 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 [2]:

$$L = k + 2e_{max} + 2l + 2|e_{min}|. \tag{2}$$

A direct implementation of the long accumulator would use a long shifter and a long adder. Performing the accumulation with a full width ripple-carry adder is unpractical as a carry propagation across it would be too expensive.

Instead of doing the operation with the entire register, only an adder with the size of the summand will be used, turning the long adder into a small one. This type of addition is generally referred to as *segmented accumulator* and was proposed by Kulisch in [2]. 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). Since the register is split into small registers the carries still need to be propagated from one segment to another. However, the carry does not need to be absorbed immediately in the next cycle, it can be stored for a certain segment and added at a later time (carry save).

3) Hardware Implementation Analysis: Table I summarises the differences between all algorithms described. 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

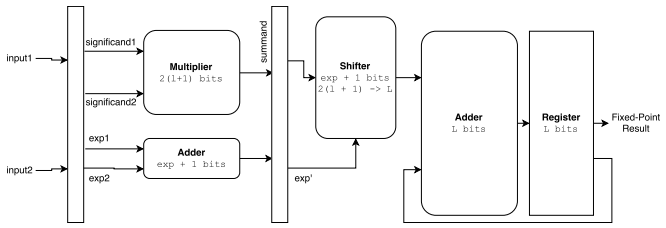


Figure 2: Mupltiplication and Accumulation (Non-Segmented Accumulator) Stages of the Inner Product Core

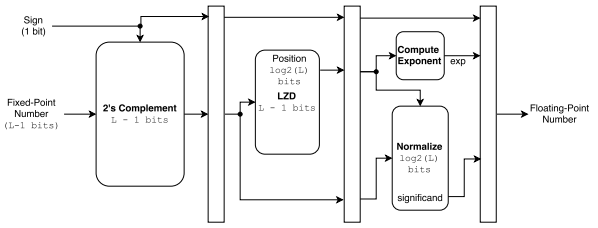


Figure 3: Fixed-Point to IEEE 754 Single Precision Conversion stages of the Inner Product Core

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 segmented long accumulator is selected.

C. 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.

The multiplication and accumulation stages are shown in Figure 2, and the final conversion stage is shown in Figure 3. 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) 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. 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.

III. Segmented Accumulator Stalling Architecture

This section proposes a segmented accumulator architecture using a register file to store the segments, which stalls the accumulation process when there are carries to propagate.

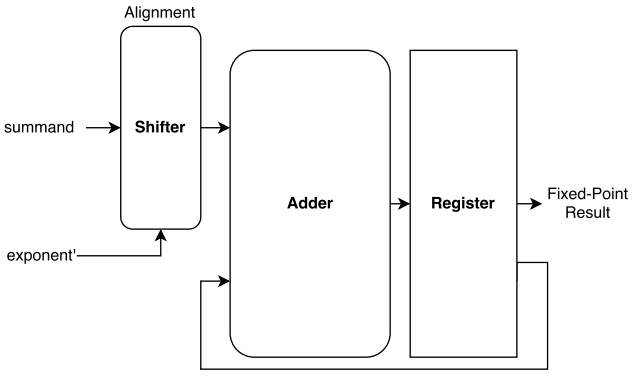


Figure 4: Execution Stage of the processor with full-width adder

A non-segmented accumulator is composed of a full-width adder and a large shifter, as shown by Figure 4.

The segmented accumulator replaces the full width adder by several segment sized adders, and the large alignment shifter is replaced by a pre-shifter and register selector. The pre-shifter changes the base of the summand to 2^y, and the register selection is a fixed shift of y.

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 Carry Processing Block (CPB), to propagate all carries generated by the segmented accumulator to all registers.

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. At the end of the carry processing the contents of the carry registers are reset. The basic architecture is shown in Figure 5.

The CPB starts propagating carries in the first register which will consume the first carry. The CPB identifies the start register from the generated carries in the segmented accumulator, where the start register is the least significant register which generated a carry plus one.

Furthermore, the CPB predicts the minimum number of cycles required by the carry propagation also from the carries generated by the segmented accumulator. The minimum number of cycles required is the difference between the most significant carry generated and the least significant carry generated plus one.

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

IV. Segmented Accumulator Non-Stalling Architecture

This section proposes an architecture which is able to process two vectors of any size without stalling the accumulation process to propagate the segment carries.

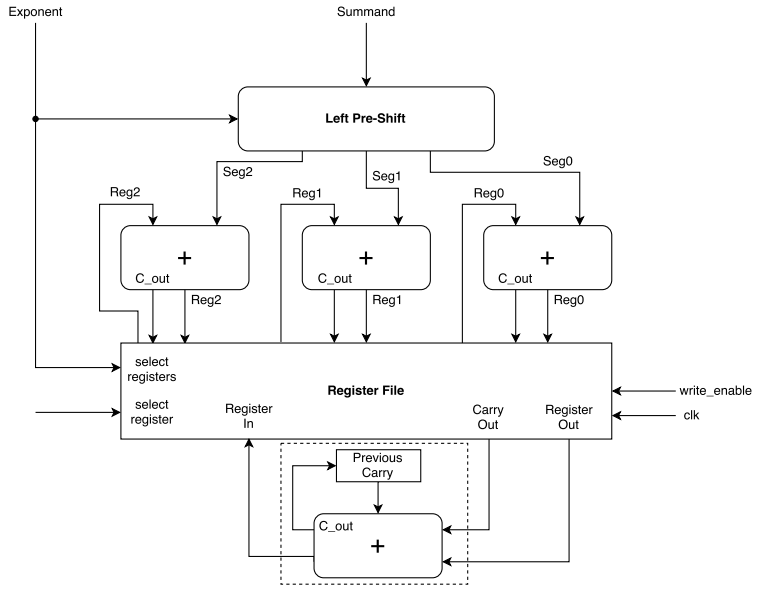


Figure 5: Stalling Segmented Accumulator and a CPB (depicted in a dashed box)

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 is split into three stages (Figure 6):

- 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 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.

A. Hazard Identification & Solutions

There are two hazards that may force the processor to stall. The first hazard occurs when two consecutive summands select the same registers, *i.e.*, summand n will consume the carries out and accumulation results from summand $n-1$. To solve the Read After Write (RAW) hazard, forwarding paths are included from the end of the EX and WB stages, and MUXs are used for selection at the end of the SF stage. Also, a carry selection is equivalent to a carry read operation, which, much like the FFA, must reset the carry in the advanced stage [18].

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). The size of the carry register (CRS) to withstand the minimum number of stored carries before a CWC hazard occurs is

$$CRS \geq \lceil \log_2 (\lceil L/y \rceil - \lfloor k/y \rfloor - NAdders + 1) \rceil, \quad (3)$$

where y is the size of the segment, L is the fixed-point accumulation size (Equation 2), k is the number of bits to guard from overflow, and NAdders is the number of adders. To solve the CWC hazard the size of the carry register must increase to accommodate for more carries and the write operation to the carry register becomes an accumulation [18].

The final architecture is shown in Figure 6.

V. Segmented Accumulator Addition/Subtraction Architectures

Since the accumulator uses segmented access, traditional signed representations are not sufficient because they require a sign propagation to all most significant registers starting at the summand's selection. In [19], Parhami proposes a framework to design redundant number representations (a digit set d_i of radix r is redundant if it contains more than r digits).

To avoid propagating a negative sign every cycle, the redundant digit set needs to support negative digits and carries, thus Generalized Signed-Digits (GSDs) are used. To minimise modifications in both architectures, the digit set d_i is represented with two's complement, and the carry digit c_i with one's complement. The base used is 2^y, where y is the segment size. To avoid propagating a sign every cycle the digit must be able

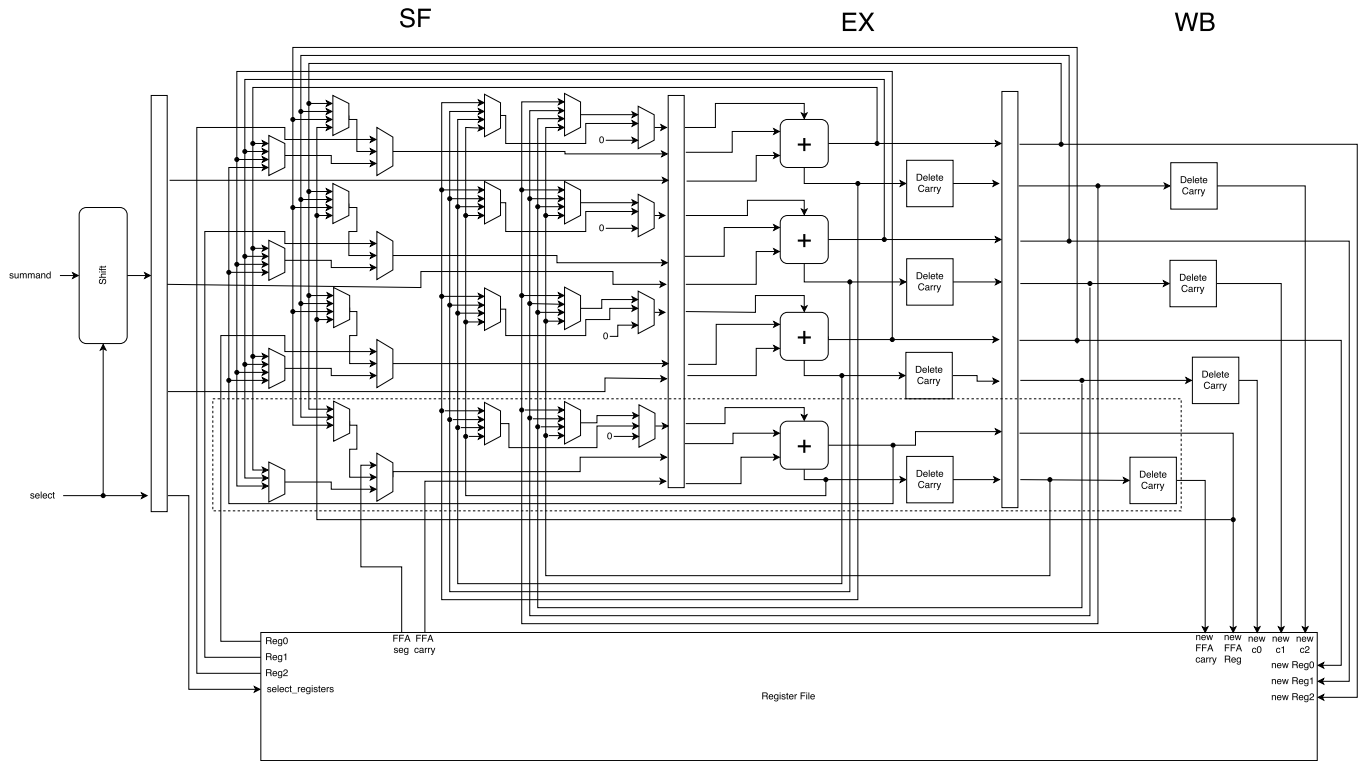


Figure 6: Non-Stalling segmented accumulator with three adders in the Execute Stage and one FFA (depicted in a dashed box)

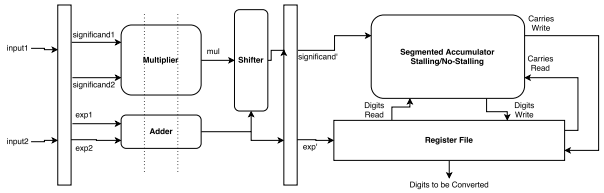


Figure 7: Block Diagram of the Segmented Accumulator Stages in the Inner Product Core

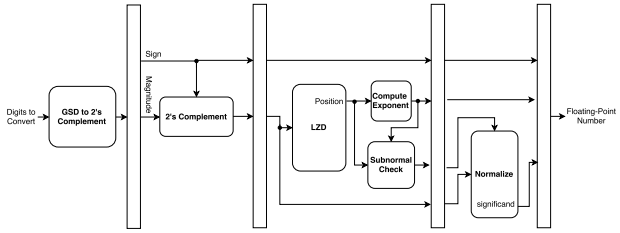


Figure 8: Block Diagram of the Conversion Stages in the Inner Product Core

to absorb a signed carry. Thus, for a carry digit set $c_i = [-\lambda, \mu]$ and $\lambda, \mu \geq 1$, the digit set is $d_i = [-2^y - \lambda, 2^y - 1 + \mu]$ [18].

Adding GSD support to the exact inner product core requires changes to the segmented accumulator and to the conversion stage to add GSD support described in subsection II-C.

To support GSD numbers the stalling segmented accumulator architecture requires the carry detection mechanism to support one's complement and multi-bit carries. The carry is only propagated on overflow, and a new stop stalling condition is defined as (by order of priority): segment selector overflow, and the CPB does not overflow and the minimum number of predicted stalls is over.

The non-stalling segmented accumulator architecture directly supports GSD numbers as long as it verifies Equation 3 (in order to avoid the CWC hazard). The final multiplication and accumulation stages in the exact inner product architecture are shown in Figure 7.

The final conversion from GSD to IEEE 754 follows these steps: each digit is multiplied by its radix order (shift), all digits are accumulated, and the final result is converted from two's complement to sign-magnitude. Since the final result is truncated, the conversion only needs the most significant non-zero w digits. The number of digits w required for a specific significand size is

$$w = \lceil size(l)/size(d_i) \rceil + 2. \tag{4}$$

The conversion accumulator (CAS) and incrementer size for w digits is

$$CAS = (w-1) \times size(d_i) - (w-2) \times 2 + 1, \tag{5}$$

and the shifter performs right shifts of y bits for w digits, where the first digit does not shift, the second digit shifts y, etc [18]. The final conversion stages for the exact inner product core are shown in Figure 8.

VI. Exact Inner Product Core Results Analysis

This section shows and analyses the FPGA implementation results for each proposed inner product core architecture, and performs two evaluations using the inner product core.

Table II: Clock Frequency (MHz) 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.

FP Size	16 bits			32 bits			64 bits		
Seg. Size	8	16	32	8	16	32	8	16	32
7010-1	107	111	116	65	68	74	-	-	-
7020-1	107	117	123	65	77	90	-	41	50
7045-2	200	227	227	107	158	172	65	78	84

Table III: 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.

	FP Size	16 bits			32 bits			64 bits		
	Seg. Size	8	16	32	8	16	32	8	16	32
7010-1	LUTs	1259	1043	1179	6979	4997	4044	-	-	-
	Flip-Flops	540	540	586	1538	1370	1344	-	-	-
	DSPs	1	1	1	2	2	2	-	-	-
7020-1	LUTs	1263	1051	1182	6978	4992	4062	-	44322	31039
	Flip-Flops	540	540	586	1538	1370	1344	-	6789	6088
	DSPs	1	1	1	2	2	2	-	9	9
7045-2	LUTs	1281	1097	1269	6980	5033	4132	82213	44877	29877
	Flip-Flops	540	540	586	1538	1370	1343	8494	6729	6085
	DSPs	1	1	1	2	2	2	9	9	9

Table IV: Clock Frequency (MHz) 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.

FP Size	16 bits			32 bits			64 bits		
Seg. Size	8	16	32	8	16	32	8	16	32
7010-1	96	100	111	65	68	80	-	-	-
7020-1	96	103	111	66	71	90	-	-	52
7045-2	192	212	238	105	151	163	62	85	103

Table V: 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.

	FP Size	16 bits			32 bits			64 bits		
	Seg. Size	8	16	32	8	16	32	8	16	32
7010-1	LUTs	2449	1898	1918	11616	7920	6105	-	-	-
	Flip-Flops	860	763	952	3637	2074	2420	-	-	-
	DSPs	1	1	1	2	2	2	-	-	-
7020-1	LUTs	2444	1899	1911	11629	7927	6120	-	-	39704
	Flip-Flops	860	763	952	3637	2074	2420	-	-	7969
	DSPs	1	1	1	2	2	2	-	-	9
7045-2	LUTs	2531	1967	2009	12054	8048	6286	142948	64363	39841
	Flip-Flops	860	763	952	3097	2570	2420	16009	10434	7960
	DSPs	1	1	1	2	2	2	9	9	9

A. *FPGA Implementation Results*

Both architectures were synthetized using Vivado 2016.3 for three Zynq-7000 devices: Z7010-1, Z7020-1 and Z7045-2. The non-default synthesis options are: 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 y: 8 bits, 16 bits and 32 bits [18]. Both architectures use a register file to store the digits, use the truncation rounding method, and use 100 bits of overflow protection.

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.

1) Stalling Segmented Accumulator Architecture: The Stalling Segmented Accumulator architectures uses a max carry digit of two. The clock frequencies and resources used are shown in Table II and Table III, respectively.

The critical path for a 16 and 32 bit floating-point stalling architecture is either in the conversion unit or in the CPB. 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 64 bit architecture requires a large number of resources, thus it cannot be implemented in the Z-7010 FPGA. 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 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, consequentially, 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.

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 clock frequencies and resources used are shown in Table IV and Table V, respectively.

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 Z-7010 FPGA. Likewise, for the Z-7020, 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.

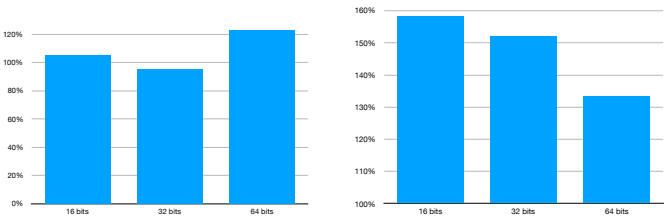
In the analysis performed the best segment size, for the lowest resource usage and highest clock frequency, for all floating-point formats is 32 bits.

B. 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 9.

Figure 9a shows that the frequency has a minimal variation between both architectures. Figure 9b 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 9: 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

C. Performance and Resources Analysis

The proposed single-precision architectures were also compared with a single-precision Multiply-Accumulator (MAC) unit (Figure 10), 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 [1], adapted for a size of 10M.

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

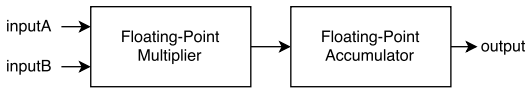


Figure 10: The single-precision Floating-Point Multiply-Accumulate unit is divided in two blocks, one multiplier and one accumulator

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 VI.

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

	Stall	**Non-Stall**	**MAC**
LUTs	4044	6105	2097
Flip-Flops	1344	2420	702
LUTRAMs	0	0	101
DSPs	2	2	9

D. Evaluation Results

This subsection 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` 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.

1) Accuracy Tests Results: In this subsection, the inner product core is evaluated using the hard to compute test sets described in [1], but adapted for single-precision. The test sets selected are:

- x_i is the ith term in the Taylor series expansion for $e^{-2\pi}$, for 35 terms. This is a classical example of rounding error propagation;
- $x_1 = x_2 = \cdots = x_{2047} = 1.0, x_{2048} = x_{2049} = 1.0 \times 10^{-18}, x_{2050} = x_{2051} = \cdots = x_{4096} = -1.0$, $\sum_{i=1}^{4096} x_i$ (Heavy Cancellation or HC). This test checks whether the arithmetic unit incorrectly infers an addition of a small number to a larger number as zero;

Table VII: Result comparison between Arbitrary Precision, the Single-Precision Exact Inner Product Core (EIPC) and Traditional Single-Precision Arithmetic

Data	AP (100 d.)	EIPC (SP)	TA
$e^{-2\pi}$	1.873410×10^{-3}	1.873410×10^{-3}	1.877666×10^{-3}
HC	2.0×10^{-18}	2.0×10^{-18}	0.000000
Equal(1,2)	6143.500000	6143.500000	6143.366210
$\sum 1/i^2$	1.644689	1.644689	1.644725
OLMS	1.999999	1.999999	1.953673

- $\sum x_i$ equally spaced on $[1,\ 2]$ for $n = 4096$. This is the easiest set to process as all numbers are exactly represented by the IEEE 754 single-precision format;
- $\sum 1/i^2$, for the range $[1,\ 4096]$. This set was used to be consistent with McNamee's work [1];
- $x_1 = 1.0, x_2 = x_3 = \cdots = x_{10^6} = 1.0 \times 10^{-3}$, $y = x$, $\sum_{i=1}^{10^6} x_i \times y_i$ (One Large, many small or OLMS). This test shares the same purpose as the Heavy Cancellation test. However, this example is more frequent than the Heavy Cancellation test, *e.g.*, when processing an integral.

The results obtained for arbitrary precision (using `bc`), the single-precision exact inner product core and the traditional arithmetic (using `C`) are in Table VII.

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.

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 the methodology used is logistic regression and the model fitting used is the Iteratively Reweighted Least Squares (IRLS).

The dataset used (from [20]) 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: There are 24 variables in the dataset, and they 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; and 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 coefficients obtained are in Table VIII.

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.

VII. 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

Table VIII: 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

	Single-Precision		**Inner Product Core (SP)**		**Double Precision**
	NCD	**Results**	**NCD**	**Results**	**Results**
Intersection	4	$1.686\,159 \times 10^{-1}$	5	$1.686\,639 \times 10^{-1}$	$1.686\,618 \times 10^{-1}$
Limit Balance	3	$7.207\,153 \times 10^{-7}$	5	$7.206\,954 \times 10^{-7}$	$7.206\,996 \times 10^{-7}$
Gender	5	$-4.611\,013 \times 10^{-1}$	6	$-4.611\,005 \times 10^{-1}$	$-4.611\,002 \times 10^{-1}$
Education	4	$-1.446\,193 \times 10^{-1}$	6	$-1.446\,234 \times 10^{-1}$	$-1.446\,230 \times 10^{-1}$
Marital Status	4	$-1.076\,489 \times 10^{-1}$	6	$-1.076\,568 \times 10^{-1}$	$-1.076\,567 \times 10^{-1}$
Age	3	$4.241\,511 \times 10^{-3}$	6	$4.240\,909 \times 10^{-3}$	$4.240\,908 \times 10^{-3}$
Rep. Status in Sept.	6	$5.301\,173 \times 10^{-1}$	7	$5.301\,170 \times 10^{-1}$	$5.301\,170 \times 10^{-1}$
Rep. Status in Aug.	6	$3.999\,695 \times 10^{-1}$	6	$3.999\,693 \times 10^{-1}$	$3.999\,696 \times 10^{-1}$
Rep. Status in July	5	$-1.397\,963 \times 10^{-1}$	6	$-1.397\,958 \times 10^{-1}$	$-1.397\,959 \times 10^{-1}$
Rep. Status in June	5	$4.201\,142 \times 10^{-1}$	5	$4.201\,121 \times 10^{-1}$	$4.201\,119 \times 10^{-1}$
Rep. Status in May	5	$-7.860\,660 \times 10^{-1}$	6	$-7.860\,625 \times 10^{-1}$	$-7.860\,623 \times 10^{-1}$
Rep. Status in April	5	$1.946\,046 \times 10^{-1}$	5	$1.946\,029 \times 10^{-1}$	$1.946\,030 \times 10^{-1}$
Amnt. Stat. in Sept.	5	$-1.820\,626 \times 10^{-5}$	5	$-1.820\,671 \times 10^{-5}$	$-1.820\,669 \times 10^{-5}$
Amnt. Stat. in Aug.	3	$-5.368\,139 \times 10^{-6}$	3	$-5.368\,968 \times 10^{-6}$	$-5.369\,043 \times 10^{-6}$
Amnt. Stat. in July	4	$3.375\,470 \times 10^{-5}$	5	$3.375\,687 \times 10^{-5}$	$3.375\,698 \times 10^{-5}$
Amnt. Stat. in June	3	$-2.202\,903 \times 10^{-5}$	5	$-2.203\,118 \times 10^{-5}$	$-2.203\,126 \times 10^{-5}$
Amnt. Stat. in May	4	$3.605\,376 \times 10^{-5}$	6	$3.605\,554 \times 10^{-5}$	$3.605\,553 \times 10^{-5}$
Amnt. Stat. in April	5	$-1.501\,226 \times 10^{-5}$	6	$-1.501\,268 \times 10^{-5}$	$-1.501\,264 \times 10^{-5}$
Amnt. Paid in Sept.	5	$-9.274\,984 \times 10^{-5}$	5	$-9.274\,957 \times 10^{-5}$	$-9.274\,945 \times 10^{-5}$
Amnt. Paid in Aug.	4	$-3.167\,262 \times 10^{-5}$	6	$-3.167\,362 \times 10^{-5}$	$-3.167\,368 \times 10^{-5}$
Amnt. Paid in July	4	$3.123\,009 \times 10^{-5}$	6	$3.123\,111 \times 10^{-5}$	$3.123\,117 \times 10^{-5}$
Amnt. Paid in June	3	$-9.288\,820 \times 10^{-5}$	5	$-9.289\,019 \times 10^{-5}$	$-9.289\,023 \times 10^{-5}$
Amnt. Paid in May	4	$1.259\,776 \times 10^{-5}$	7	$1.259\,804 \times 10^{-5}$	$1.259\,804 \times 10^{-5}$
Amnt. Paid in April	5	$-2.003\,924 \times 10^{-5}$	6	$-2.003\,911 \times 10^{-5}$	$-2.003\,913 \times 10^{-5}$

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.

REFERENCES

[1] J. M. McNamee, “A comparison of methods for accurate summation”, *ACM SIGSAM Bulletin*, vol. 38, pp. 1–7, Mar. 2004.

[2] U. W. Kulisch, “Advanced arithmetic for the digital computer - design of arithmetic units”, *Electronic Notes in Theoretical Computer Science*, vol. 24, pp. 68–139, Apr. 2000.

[3] “Ieee standard for floating-point arithmetic”, IEEE Computer Society, Tech. Rep., 2008.

[4] "Precision and performance: floating point and ieee 754 compliance for nvidia gpus", NVIDIA Corporation, Tech. Rep., 2015.

[5] Intel, *Intel intrinsics guide*, https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=fma&techs=FMA&expand=2407, [Online, accessed 28-November-2016], 2008.

[6] C. Baumhof, "A new vlsi vector arithmetic coprocessor for the pc", *Proceedings of the 12th Symposium on Computer Arithmetic*, pp. 210–215, Jul. 1995.

[7] S. Siegel and J. W. von Gudenberg, "A long accumulator like a carry-save adder", *Computing – Springer Journals*, pp. 203–213, Nov. 2011.

[8] J. W. V. Gudenberg, "Comparison of accurate dot product algorithms", INRIA, Research Report RR-2413, 1994. [Online]. Available: https://hal.inria.fr/inria-00074262.

[9] M. Pichat, "Correction d'une somme en arithmétique à virgule flottante", *Numerische Mathematik*, vol. 19, no. 5, pp. 400–406, Oct. 1972.

[10] U. Kulisch and G. Bohlender, "Formalization and implementation of floating-point matrix operations", *Computing*, vol. 16, no. 3, pp. 239–261, Sep. 1976.

[11] G. Bohlender, "What do we need beyond ieee arithmetic?", *Computer Arithmetic and Self-Validating Numerical Methods*, pp. 1–32, 1990.

[12] W. Kahan, "Further remarks on reducing truncation errors", *Communications of the ACM*, vol. 8, no. 1, pp. 40, 48, Jan. 1965.

[13] D. M. Priest, "Algorithms for arbitrary precision floating point arithmetic", in *Computer Arithmetic, 1991. Proceedings., 10th IEEE Symposium on*, Jun. 1991, pp. 132–143.

[14] A. Klein, "A generalized kahan-babuska-summation-algorithm", *Computing*, vol. 76, no. 3, pp. 279–293, Nov. 2006.

[15] J. Jankovic, M. Subotic, and V. Marinkovic, "One solution of the accurate summation using fixed-point accumulator", in *Telecommunications Forum Telfor (TELFOR), 2015 23rd*, Nov. 2015, pp. 508–511.

[16] M. A. Malcolm, "On accurate floating-point summation", *Commun. ACM*, vol. 14, no. 11, pp. 731–736, Nov. 1971.

[17] D. R. Ross, "Reducing truncation errors using cascading accumulators", *Commun. ACM*, vol. 8, no. 1, pp. 32–33, Jan. 1965.

[18] L. Fiolhais, "Exact inner product processor in fpga", Master's thesis, Instituto Superior Técnico, 2017.

[19] B. Parhami, "Generalized signed-digit number systems: a unifying framework for redundant number representations", *IEEE Transactions on Computers*, vol. 39, no. 1, pp. 89–98, 1990.

[20] I.-C. Yeh, *Default of credit card clients data set*, https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients, [Online, accessed 28-August-2017], 2016.