POLITECNICO
DI TORINO

MASTER OF SCIENCE IN ELECTRONIC ENGINEERING

Master's Thesis

Design and characterization of Variable Latency adders for floating-point arithmetic units

Supervisor

Prof. Maurizio MARTINA

Candidate

Leonardo PEDONE

April, 2018

Acknowledgments

I would like to thank Professor Maurizio Martina for the opportunity to work on this project for my thesis of the Master of Science in Electronic Engineering and for the support provided in this last months. Considering that in these years of study the main field of interest were digital systems I consider the work done to be very satisfying because it gave me the opportunity to put most of the accumulated knowledge into practice. It was interesting and challenging to learn more about adders architectures and study new techniques like the speculative approach to increase their performances in a practical application like the floating-point addition.
At the end of this work I feel that I have increased my knowledge regarding digital systems and improved the skills acquired during my studies.

Dedication

Because of the personal nature of this section and to make sure that the people to whom it is addressed can better understand it, the language used is Italian.

Alla fine di questo percorso universitario presso il Politecnico che segna anche la fine del mio cammino di formazione, il primo pensiero va ai miei genitori grazie ai quali non sarei potuto arrivare a questo traguardo e sopratutto non sarei la persona che sono ora. Il loro appoggio sia dal punto di vista morale che economico mi ha permesso di seguire la mie ambizioni e le mie passioni e spero di essere riuscito almeno in parte a ripagarli e renderli fieri. Naturalmente un grande sostegno è arrivato anche dal resto della mia famiglia e parenti, perciò meritano tutti un ringraziamento a partire da mio fratello Emanuele fino a tutti i nonni, zii e cugini.

La seconda persona che vorrei ringraziare è Alessia, per tutti questi anni insieme, per essere cresciuti tenendoci per mano, per avermi supportato anche nei momenti più difficili, per non aver mai fatto ostacolo della distanza che ci separava e per moltre altre cose ancora che lei sa.

Per ultimi, ma non per questo meno importanti vorrei ringraziare i miei amici; i nuovi amici conosciuti durante il cammino universitario tra Ancona e Torino e gli amici di una vita con cui sono cresciuto a Civitanova tra cui Massimiliano e Francesco che oltre ad essere amici sono stati anche coinquilini e compagni di vita per questi anni al Politecnico.

Contents

Chapter 1

Introduction

1.1 Main Pourpose and Goals

The aim of this Master thesis is the study, design and validation of architectures for floating point addition that are based on a speculative approach.
Previous works showed that nowadays Graphic Processing Unit (GPUs) are used in a vast range of application and are not limited to image/video processing. This is because the high parallelism computation that a GPU can provide, with respect to the classic CPU, can achieve better performances when a big amount of data need to be analyses and processed.
What has been done previously was using a preexisting GPU model and some benchmarks to found what type of operation frequently used were not yet implemented. The results showed that floating point addition are very frequent in benchmarks, so the focus was to design an arithmetic unit and implement it to improve the model with a hardware accelerator.
Starting from this results what has been done in this work was perform a more exhaustive study of the arithmetic block in order to improve its performances.
From the various classes of arithmetic blocks present in the literature the one selected for the study is the *Variable-Latency* arithmetic unit based on the concept of Speculation. Speculation means that the unit assumes that

the results of the operations is correct most of the times. Based on this, if the application in which the architecture is implemented can afford some errors, the results is produced by a *fast_ track*, i.e. a computational path that provide a faster result but with some approximations that could generate a wrong solution.
In this specific case a *parallel prefix adder(PPA)* is used and the assumption is that the carry generated during the additions does not propagate more that k bits. This is derived from the literature, where is pointed out that the carry rarely propagates more than $\log_2(n)$ bits in a n-bit adder.
If the results must be exact all the times a error detection and correction, i.e. a *error_ network*, must be implemented. In case of an error the wrong result from the *fast_ track* is discharged and one more clock cycle is needed to provide the correct result, from this the name *Variable-Latency*.
In the thesis a study was performed on arithmetic units with various levels of speculation, with and without the *error_ network*. Area, power and timing of the various design were evaluated with the different types of *IEEE754* format for floating-point numbers and confronted with the standard implementation in order to find the optimal configuration for different applications. At the end the error rate was evaluated to have one more parameter for the study. The final goal of the work is to provide a vast range of solutions for several applications that could be implemented in GPU, but also in other IC architecture as ASIC and FPGA.

1.2 Brief history of numerical computing

The concept of *numerical computing* has its root in the early history of civilization [1], although with different representations and methods all the population exploited the computation by mean of numbers (or other symbols) to solve various problems of everyday life. With the advancement of technology the rudimental tools like the abacus were replaced by more efficients machines like mechanicals calculators used untill the mid 70s where the advent of affordable computers provides a big jump forward in terms of performances.
Starting from the first general purpose computer the ENIAC created during WWII by what is considered the father of computers, Alan Turing, the development of new architecture was steady and fast. The ideas of using the binary format for numbers and storing the instruction to be executed in the memory of the computer proposed by Jhon von Neuman lay the foundations to modern computers.
Due to the high cost and the great size of the first machines the first uses were limited to the scientific field and information processing for large company that did not use them for a purely numerical computation purpose. Nowadays with the advance in technology everyone can afford a computer in their home, but the main focus remains the processing of information (i.e. text, video ,audio) but all the data are always converted into binary numbers and so the numerical computing is still the core of the machines.
Today the main field which employs pure numerical computing is used are still the scientific disciplines. In physics for the solution of models composed by complex equations that describes from the micro structure of small particles to the expansion of the universe and to process big volumes of data obtained from experiments. In the medical area for the analysis of clinical data and imaging techniques. Atmospheric scientists use the computing performances for weather prediction by elaborating an high number of variables like moisture and atmospheric pressure. In the design industry (i.e. IC design, mechanical design, buildings design) the computing performances of the

CAD tools allow for more complex, accurate and reliable projects. In short the *computational science* can be considered a newly fundamental branch of science and the improvement of computational techniques gives access to the creation of faster and more performing computers.

1.3 IEEE754 format for floating-point numbers representation

Floating-point number representation is the most used standard representation for data since the early days of computers [1]. Initially computer manufacturers developed their own floating point system, usually to optimize some aspects of their machines. An example was IBM that uses an hexadecimal system in order to improve the range of the number that can be represented and decrease the amount of shifts needed for the normalization, but the drawback was on the accuracy with respect of a binary system. The absence of a common standard produced many difficulties like create a software able tu run smoothly on every computer or the inconsistency of some floating-point properties. The solution was provided in the early 80s when a collaboration between computer scientists and manufacturers gave birth to the first **IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)** used until today with some minor modifications.
In this section a description of the IEEE754 standard for the floating point data is given. This is the data format implemented in the arithmetic unit, its structure heavily affected the design of the architecture.

The last revision of the standard was published by IEEE in 2008[2] and its main purpose is to provide a mean to perform floating-point based computations that will yield the same results independently from the hardware and/or software used. The standard specifies formats for the binary and decimal data, operations, conversions and exceptions handling.

A finite number is described by three fields:

- Sign
- Exponent

- Significand

The number is written as displayed below, the base can be arbitrary and in this case in 2:

$$number = (-1)^{sign} \times base^{(exponent-bias)} \times ((1.)significand) \quad (1.1)$$

The sign is represented with a single bit and can be either 0 for positive numbers or 1 for negative as in the standard binary numbers. The exponent and the significand are the fields that displays the main properties of floating-point numbers.
The exponent is represented in a biased form, the bias is used to achieve only positive values for the exponent although the original value may be negative. As can be seen in 1.1 in order to obtain the biased exponent the bias must be added to the original value:

$$Exp_{bias} = Exp + bias \quad (1.2)$$

.
The bias does not influence the final result when performing operations because the same quantity is added at all the numbers with the same format.
The significand is always represented in a normalized form, this is used to achieve compatibility and interchangeability between operands and results of an operation.
The normalization for the IEEE754 format consist in shifting the number until an *unsigned fractional point* form is achieved, by doing this the significand always have a leading 1 in the MSB position. After that the MSB is discharged and only implicitly represented in the significand field.
Often although the operands of an operation are normalized the final result may not be in the correct form, this implies that a normalization must be performed. When the significand part of the result is shifted the exponent part must be modified accordingly for a correct result.
This makes normalization a critical part when designing an arithmetic unit

based on the IEEE754 format.

The IEEE754 format for floating-point numbers provides various length for data parallelism, they are represented in table 1.1. In the table are described the most common parallelism used but the standard can be extended to all length of data. Alternatives names can be used for the main formats, starting from *binary_16* they are: *half-precision, single-precision, double-precision, quad-precision.*

	IEEE754-2008				
Field (in bits)	***binary_16***	***binary_32***	***binary_64***	***binary_128***	***binary_(n)***
sign	1	1	1	1	1
exponent field width (w)	5	8	11	15	$round(4\log_2(n)) - 13$
trailing significand field width (t)	10	23	52	112	$n - w - 1$
precision (p)	11	24	53	113	$n - round(4\log_2(n)) + 13$
storage width (n)	16	32	64	128	$1 + t + w$ (multiple of 32)
maximum exponent(e_{max})	15	127	1023	16383	$2^{w-1} - 1$
bias	15	127	1023	1683	e_{max}

Table 1.1: IEEE754-2008 standard

An example of number conversion to IEEE754 is provided in order to clarify what as been said so far, the initial number is -43.34375 and the floating-point format chosen is *binary_32*.
Starting from the sign, the number is negative so the sign bit is 1. Now we convert the absolute value of the number to a *fractional binary number*, 43.34375 becomes 101011.01011; now a normalization must be performed so the number is shifted to the right by 5 position in order to obtain a 1 as MSB of the *fractional part*, the result is 1.0101101011. Then the exponent is evaluated, the exponent must compensate for the shift in the opposite direction, so its value is 5; this value need to be biased, the bias for *binary_32* is 127 so the new value is $E = 5 + 127 = 132$. The biased exponent is converted to *unsigned binary* with length 8 bit imposed by the standard: 10000100. At the end the missing LSBs of the significand are filled with zeros to obtain the correct length of 23 bit and the leading 1 is truncated.

The result of the conversion is:

sign	*Exp(biased)*	*significand*
1	10000100	01011010110000000000000

In order to design a floating-point arithmetic unit that generates the right results is essential to correctly understand the various formats and how they works in the different operations. In chapter 3 a detailed explanation on floating-point addition and subtraction is provided. Multiplication and division are less complex because no pre-alignment or comparison of the operands is needed, and the sign of the result is simply a *xor* between the sign bits.

1.4 Previous work

In [3] a model for the AMD GPUs of the Evergreen family (Radeon HD 5000 series) was created thanks to the details provided in [4]. The model was created using OpenCL, a standard based on C language that allow to develop application for different targets (i.e. CPUs, GPUs, DSPs), to implement the architecture but with only a single Compute unit for simplicity. A OpenCL model can be tested by using various testbenches that evaluate the performances in some specific scenarios and one common application is the floating-point addition. The instruction set up to that point was fully operative but not complete and as a matter of facts it was lacking the support to floating-point addition.
In [5] the model was expanded by creating a VHDL model of a floating point adder and adding the relative instructions. In order to improve the performances various solution for the adder were tested and a first attempt to use speculation was made but considering only a small subset of the possible cases.
In this work the arithmetic unit is revisited and improved, and a deeper analysis of the various speculative solutions was performed to provide a complete study of the architecture that can be used to evaluate the better implementation according to the final application.

Chapter 2

Parallel prefix adder

Adders are one of the most common component in a digital architecture and various types area available, from the most simple Ripple Carry Adder to the more optimized Carry Look Ahead, Parallel Prefix and more. Due to the wide range of possibilities, the type of adder must be selected with regards to the architecture in which is implemented so that the final performances are maximized.In this thesis the architecture is designed to perform addition of floating point numbers, various formats are used but in all the cases the parallelism of data is quite high. As proved in [6] when working with long data the architecture with the best overall performances is usually the Parallel Prefix Adder. This because the other adders (i.e. Ripple Carry, Carry Skip, Carry Select, Carry Lookahead) offers a linear dependency $O(n)$from the data length for both area and delay, where PPA provides better results with a logarithmic behavior $O(log_2(n))$.

2.1 Parallel prefix adders (PPA)

Starting from two n-bit operands $A = a_{n-1}a_{n-2}...a_0$ and $B = b_{n-1}b_{n-2}...b_0$ the resulting sum of the i-th couple of bit s_i and carry out c_i can be evaluated as:

$$s_i = a_i \oplus b_i \oplus c_{i-1} \tag{2.1}$$

$$c_i = a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_{i-1} \tag{2.2}$$

In classic adders the carry out evaluation of the last bit must wait for the correct carry to propagate from the stage of the adder where it has been generated. In Carry Lookahead adders all the internal carries are evaluated in parallel, this is possible because they depend only on the bits of the two operands [7]. The concepts of carry propagate $p_i = a_i \oplus b_i$ and carry generate $g_i = a_i b_i$ signals are introduced, as the names implies propagate means that a couple of bit pass on the carry in received and generate that those bit produce a carry out. Now this two signals are used in 2.2:

$$c_i = g_i + p_i \cdot c_{i-1} \tag{2.3}$$

This equation can be expanded by substituting $c_{i-1} = g_{i-1} + p_{i-1} \cdot c_{i-2}$:

$$c_i = g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot c_{i-2} \tag{2.4}$$

The same can now be done with c_{i-2}, if this procedure is repeated till c_0 the final result is:

$$c_i = g_i + p_i \cdot g_{i-1} + p_i \cdot p_{i-1} \cdot c_{i-2} + ... + c_0 \cdot p_0 \cdot ... \cdot p_{i-1} \cdot p_i \tag{2.5}$$

Equation 2.5 shows that there are only two ways to have a carry in in the i-position, either the i_{-1}-th bit generates one or it is generated from previous bit and propagated till the i-th bit. The components required are only basics $AND-OR$ and the total delay of the architecture is quite good, if T_{gate} is the delay of a single component, the total delay is $5T_{gate}$ (1 for p and g, 2 for c_i, 2 for s_i). There are two problems with this solution for high value of n, first there is a large area overhead because more components are required, second

components need an high fan-in $(n + 1)$. The Parallel Prefix adder solve this problems with the implementation of the **Parallel Prefix Network** to evaluate the terms needed in 2.5.
A schematic view of the PPA adder structure is illustrated in Fig. 2.1, there are three main stages that produce the final sum:

1. **Pre-processing**
2. **Parallel Prefix Network**
3. **Post-processing**

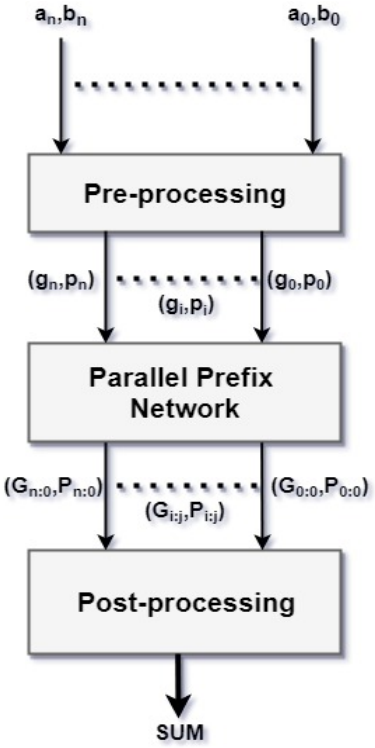


Figure 2.1: Block diagram of Parallel Prefix Adder

The fist is the **pre-processing** stage where the p_i and g_i are generated in the same way as said before. The second stage is the **Parallel Prefix Network** and is the main part of the adder. Two new terms are introduced, the group-propagated carry $P_{i:j}$ and the group-generated carry $G_{i:j}$ defined as follows:

$$P_{i:j} = \begin{cases} p_i & \text{if } i = j \\ p_i \cdot p_{i-1:j} & \text{if } i > j \end{cases} \tag{2.6}$$

$$G_{i:j} = \begin{cases} g_i & \text{if } i = j \\ g_i + p_i \cdot g_{i-1:j} & \text{if } i > j \end{cases} \tag{2.7}$$

This two are the prefixes used for the evaluation of the carry for every weighted position and are considered as a pair $(P_{i:j}, G_{i:j})$. A new operator is introduced to deal with this pair of signals and is called **carry operator**[8], it will be indicated with "$\circ$". The behavior of the new operator is the following:

$$(g_i, p_i) \circ (P_{i-1:j}, G_{i-1:j}) = (g_i + p_i \cdot G_{i-1:j}, p_i \cdot P_{i-1:j}) = (G_{i:j}, P_{i:j}) \tag{2.8}$$

The carry operator has the *associative* property:

$$[(g_i, p_i) \circ (P_{i-1:1}, G_{i-1:1})] \circ (g_0, p_0) = (g_i, p_i) \circ [(P_{i-1:1}, G_{i-1:1}) \circ (g_0, p_0)] \tag{2.9}$$

But it is not *commutative* because $g_i + p_i \cdot G_{i-1:j} \neq g_{i-1} + p_{i-1} \cdot G_{i:j}$.

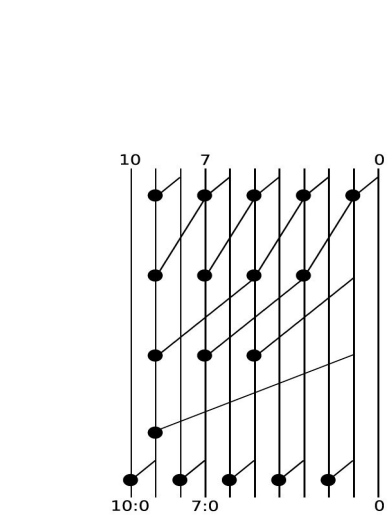
Now all the elements to build the network are present, the associative property of the carry operator offers the possibility to aggregate the intervals in different orders and also overlap them but obtaining the same result in the end. This leads to various solution for the prefix network, each provides different performances in terms of area and delay. In Tab.2.1 the existing solution for the prefix network are represented using two parameter:

- **Complexity**: number of carry operator used in the network.
- **Delay**: number of levels in the that create the critical path of the network.

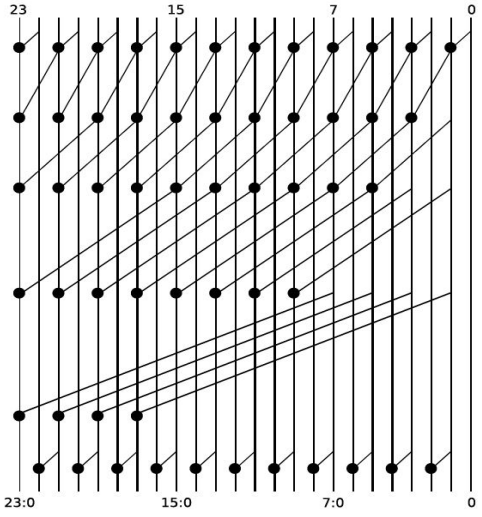
Parallel Prefix Network		
Topology	*Complexity*	*Delay*
Ladner-Fischer[9]	$\frac{n}{2}\log_2(n)$	$\log_2(n)$
Brent-Kung[10]	$2n-2-\log_2(n)$	$2\log_2(n)-2$
Kogge-Stone[11]	$2\log_2(n)-n+1$	$\log_2(n)$
Han-Carlson[12]	$\frac{n}{2}\log_2(n)$	$ceil(\log_2(n)+1)$

Table 2.1: Performances of the various topologies for the parallel prefix network.

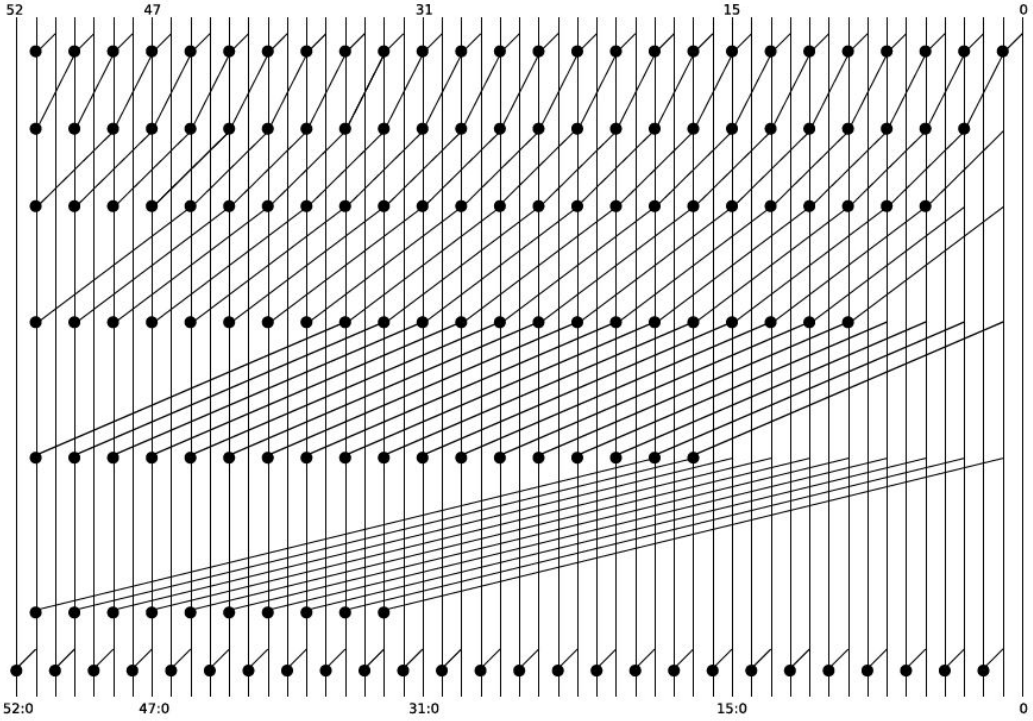
Some observations can be done on the presented topologies, Brent-Kung(BK) uses a smaller number of computational reducing the area but the depth of the network increases and so the delay, Kogge-Stone(KG) achieves high speed but the complexity of the circuit increases, Ladner-Fisher(LF) offers a good trade-off between the two[13]. Han-Carlson(HC) is an hybrid that uses the outer rows of the BK and the internal rows of the KG [14] providing a delay similar to the letter but with less complexity.
This last network is the one selected in this thesis and the motives are basically two. First is the delay, looking at Tab. 2.1 the best architectures should be LF and KS, but another parameter that must be considered is the *fan-out* (i.e. how many operators are connected at the output of a single operator), the best solution in this respect is KG but its complexity is really high, so a good trade-off is HC that has slightly worse in delay and fan-out but it is less complex. The second reason is that as proved in [14] the *error_network* for a speculative HC adder has better performances than a KG one, more details are in 2.3.



(a) p=11



(b) p=24



(c) p=53

Figure 2.2: Han-Carlson networks for binary_16 32 and 64 precisions of IEEE754 standard.

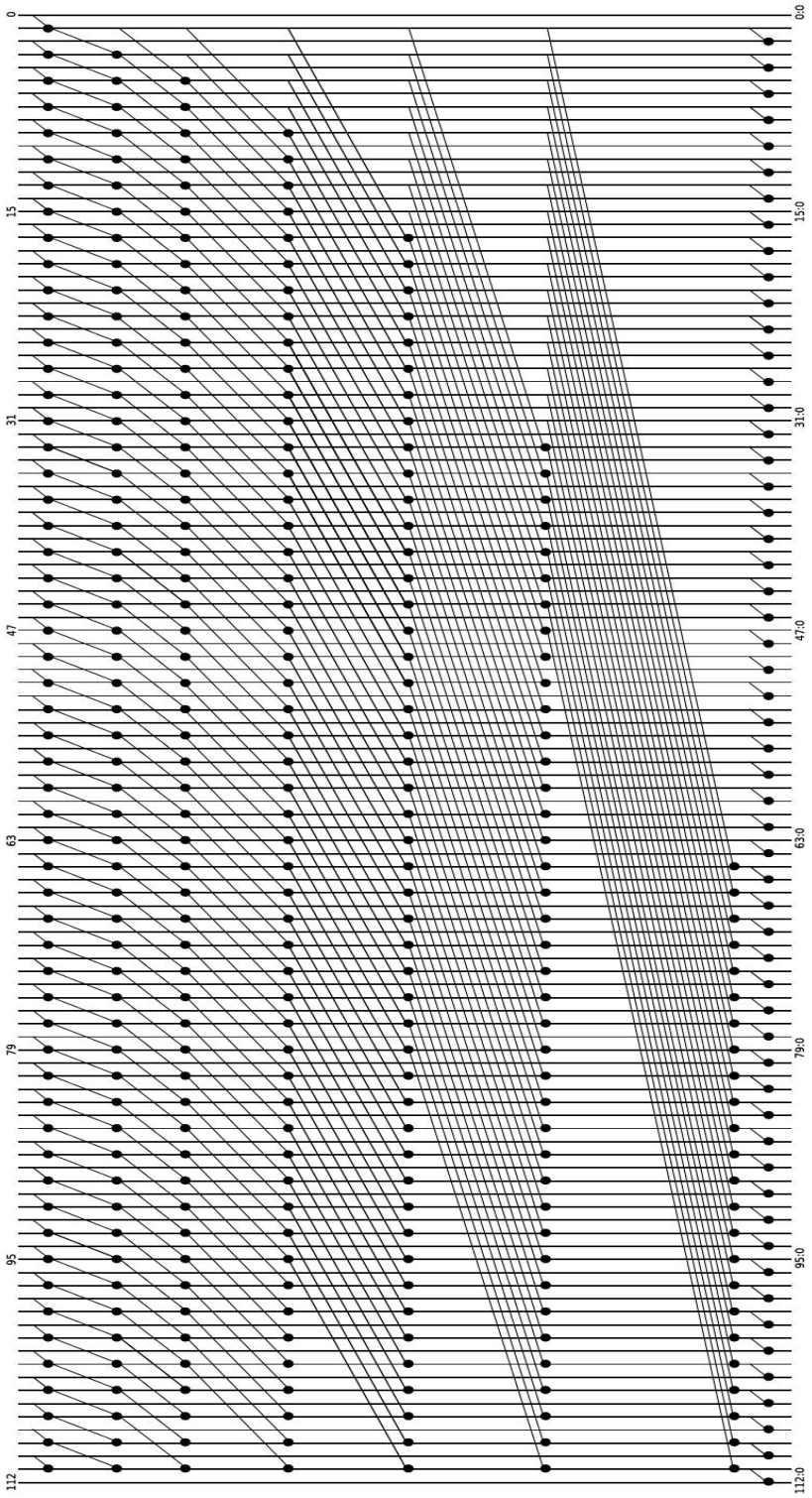


Figure 2.3: Han-Carlson networks for all four precisions of IEEE754 standard.

In 2.3 are the HC networks for the four IEEE754 standard considered in this work, the number of bit is the *precision)*(p) of the various format because the adder is used for the sum of the significands. Each line of the network receives as inputs the couple of (g_i, p_i) from the pre-processing stage, the carry operators "$\circ$" has in input a couple of (g_i, p_i) or a couple of $(G_{i:j}, P_{i:j})$ from a previous operand. The final outputs of the network are the couples $(G_{i:0}, P_{i:0})$ with $p - 1 \geq i \geq 0$. Lastly it is worth noticing how in the HC the operators are placed in the even lines only at the last stage

The last stage of the adder is the **post-processing**, where the carries and the final sum are evaluated as follow:

$$c_i = G_{i-1:0} + P_{i-1:0} \cdot c_0 \tag{2.10}$$

$$s_i = p_i \oplus c_i \tag{2.11}$$

Looking at Eq.2.10 it can be seen how each carry is independent from the others, the only exception is the initial carry in of the adder. If there is no input carry Eq.2.10 can be further simplified, but the architecture designed always allow a carry in so that both sum and subtraction can be performed. This is not a problem because the carry is available from the start with the two operators, so no delay is added.

2.2 Speculative approach

Nowadays with the increment of ICs complexity more performing architecture for components such as adders is required. The three key points in the optimization of new digital ICs are speed, area and power. As said in [15] for floating point arithmetic units a critical point of improvement is the power consumption because is much higher than fixed point signed/unsigned architecture, and to improve the performances a *inexact apprach* is proposed. This solution offers an overall improvement but provides an erroneous result, this can be neglected in some applications like image processing but is not a general solution.

Another possible approach to solve the problem is using **speculation**, first introduced for asynchronous components in [16]. In this solution the datapath is composed by two or more delay paths each with a different length where the shortest provide a result using the highest level of speculation and the longest use standard computation. Speculation means that the unit makes a guess during the computation of the result, if the assumption was right the output is correct otherwise is wrong. In synchronous contest a **variable latency speculative** adder was proposed in [17] using Kogge-Stone network, but for the reasons explained in the previous section the solution choosed in this work is the one in [14] which uses a Han-Carlson network.

In a parallel prefix network as seen in Eq.2.10 the group-propagated and the group-generated from the first position to the i-th are needed for a correct carry evaluation, this because is assured that if a carry is generated from a previous addition than is correctly passed on. But has been proven in [8] that carry propagation longer that $\log_2$ bit rarely occurs, so the assumption that has been made in the architecture is that the **maximum propagation of the carry is** $k = O(\log_2(n))$ with $k < n$. This assumption allow to prune some of the levels in the adder network. in particular the reduction is:

$$k = \frac{n}{2^C} \tag{2.12}$$

Where k is the maximum carry propagation, n the number of bit and C the collapsed levels. So the final number of stages in the network are:

$$N = ceil(1 + \log_2(k)) \tag{2.13}$$

The pruned levels cause some variations in the group-generate and group-propagate evaluation:

$$\begin{aligned} (G_{i:0}, P_{i:0}) \quad & \text{for } i \leq k \\ (G_{i:i-k+1}, P_{i:i-k+1}) \quad & \text{for } i > k,\ i \text{ odd} \\ (G_{i:i-k}, P_{i:i-k}) \quad & \text{for } i > k,\ i \text{ even} \end{aligned}$$

For the study various values of k were used for the main four IEEE754 standards, naturally longer formats can have more solution for the values of the carry propagation. In Tab.2.3 an overview on the complexity and delay of all the cases analyzed is provided(the precision p consider the non-normalized significand).

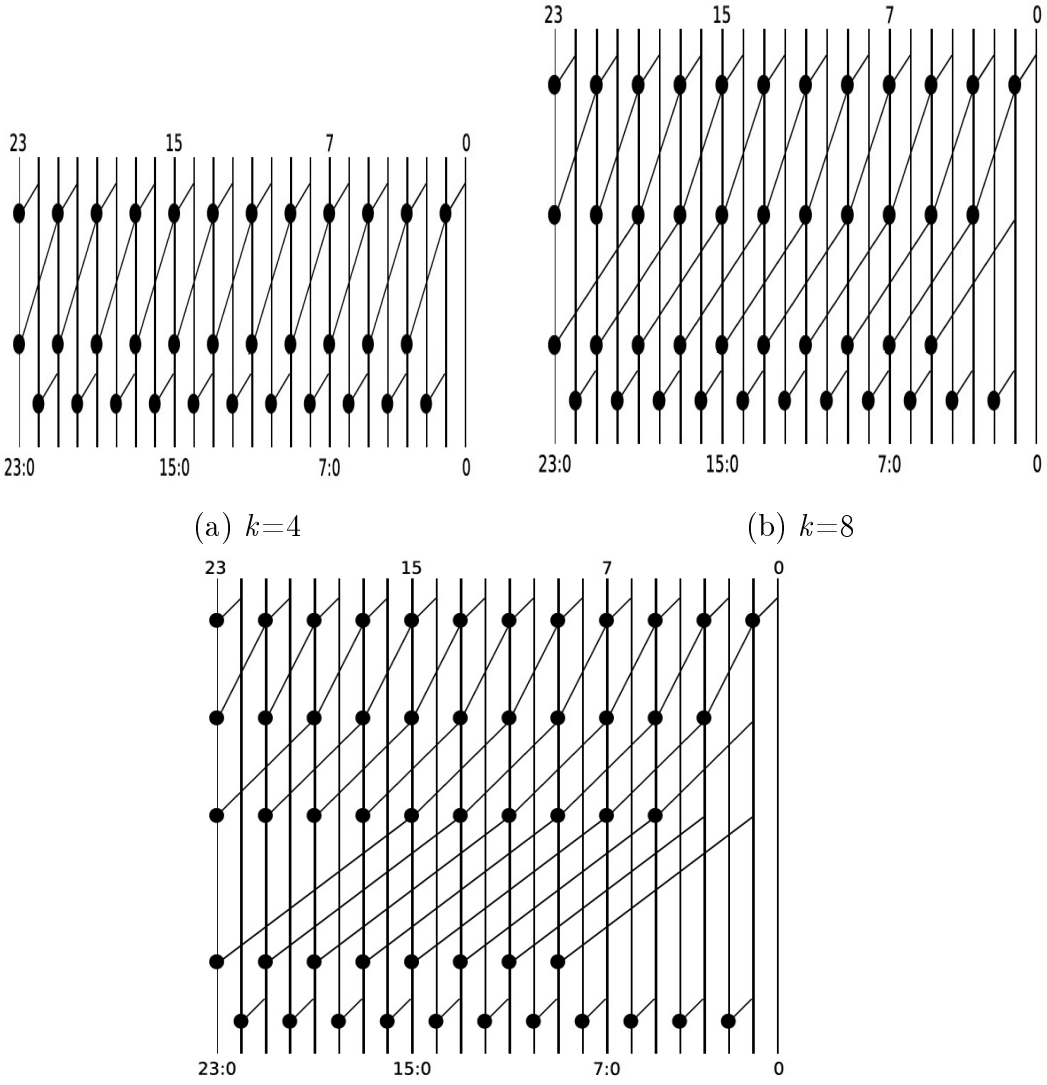
Han-Carlson Networks Delay and Complexity

Width		*Quantity*	*Exact*	*Speculative*				
n	p			k=4	k=8	k=16	k=32	k=64
		Complexity	18	14	17			
16	11	C-Reduction		22.22%	5.56%			
		Delay	5	3	4			
		D-Reduction		40%	20%			
		Complexity	56	34	44	52		
32	24	C-Reduction		39.29%	21.43%	7.14%		
		Delay	6	3	4	5		
		D-Reduction		50%	33.3%	16.67%		
		Complexity	151	77	101	123	141	

64	53	C-Reduction		49%	33.11%	18.54%	6.62%	
		Delay	7	3	4	5	6	
		D-Reduction		57.14%	42.86%	28.57%	14.29%	
		Complexity	385	165	221	273	321	361
128	113	C-Reduction		57.14%	42.6%	29%	16.6%	6.23%
		Delay	8	3	4	5	6	7
		D-Reduction		62.5%	50%	37.5%	25%	12.5%

Table 2.3: Delay and complexity for all the IEEE754 format speculative adders and improvements with respect to the exact adder.

It can be noticed that for a specific k the carry propagation length is fixed so larger parallelism have a bigger reduction in the critical path. As an example for $k = 4$ the pruned levels are 2 for binary_16 and 5 for binary_128, consequently there is a big improvement on delay and area. An aggressive approach seems to provide only positive aspects but there is the problem related to the error. More accurate analysis is provided in 4.3, but a thing can be easily said which is that in longer data the probability of an error is grow with the speculation. This because the carry is more likely to propagate more then 4 positions in a sum of 113 bit than 11 bit. A good solution is find a trade-off between performances and error that is compatible with the application in which the adder is implemented.
For the sake of clarification in Fig.2.4 are represented the three speculative Han-Carlson for the binary_32 as example.



(a) k=4 (b) k=8

(c) k=16

Figure 2.4: Han-Carlson prefix network for binary_32 format implemented with three levels of speculation $k = 4, 8, 16$.

The final stage of post-processing has some variation too because now the sum is evaluated using approximated carries:

$$\tilde{c}_i = \begin{cases} G_{i:0} + P_{i-1:0} \cdot c_0 & \text{for } i \leq k \\ G_{i:i-k+1} & \text{for } i > k,\ i \text{ odd} \\ G_{i:i-k} & \text{for } i > k,\ i \text{ even} \end{cases} \tag{2.14}$$

$$\tilde{s}_i = p_i \oplus \tilde{c}_i \tag{2.15}$$

2.3 Error detection and correction

In the variable latency speculative adder when a wrong result is generated this need to be detected and reported to the outside so that it can be discharged and the correct result will be provided in the next clock period, the details of this process are in 3.3.

An ***error_detection*** network must be created to find errors if some carries have propagated more than the speculation, this network generates a signal called **detection** that it is asserted when an wrong result is present. From Eq. 2.14 the condition for the error on the i-th bit can be obtained and is the following:

$$e_i = \begin{cases} 0 & \text{for } i \leq k \\ P_{i:i-k+1}G_{i-k:0} & \text{for } i > k,\ i \text{ odd} \\ P_{i:i-k}G_{i-k-1:0} & \text{for } i > k,\ i \text{ even} \end{cases} \tag{2.16}$$

The detection signal is asserted if one or more errors are present, so it is obtained from the sum of each i-th error.

$$D = \sum_{i=k+1,\ i \text{ odd}}^{n-1} P_{i:i-k+1}G_{i-k:0} + \sum_{i=k+1,\ i \text{ even}}^{n-1} P_{i:i-k}G_{i-k-1:0} \tag{2.17}$$

From Eq. 2.17 it can be noticed that the second summation is included in the first, as an example the terms for $i = 1, 2$ are evaluated.

$$P_{k+1:2}G_{1:0} + P_{k+2:2}G_{1:0} = P_{k+1:2}G_{1:0} \tag{2.18}$$

Eq. 2.17 is reduced to:

$$D = \sum_{i=k+1,\ i\ \text{odd}}^{n-1} P_{i:i-k+1} G_{i-k:0} \tag{2.19}$$

Further simplifications can be applied to Eq. 2.19. Let's evaluate the errors for $n-1$ and $n-3$:

$$P_{n-1:n-k} G_{n-1-k:0} + P_{n-3:n-2-k} G_{n-3-k:0} \tag{2.20}$$

If now the term $G_{n-1-k:0}$ is rewritten as:

$$G_{n-1-k:0} = G_{n-1-k:n-2-k} + P_{n-1-k:n-2-k} G_{n-3-k:0} \tag{2.21}$$

Eq. 2.21 can be substituted in Eq.2.20 obtaining as a result that the terms $n-1$ and $n-3$ are simplified :

$$P_{n-1:n-k} G_{n-k-1:n-k-2} + P_{n-3:n-k-2} G_{n-k-3:0} \tag{2.22}$$

This process can be iterated for all odd pairs (i.e $n-3$ and $n-5$, $n-5$ and $n-7$, $n-7$ and $n-9$,...) obtaining as final result a new equation for the detection signal:

$$D = \sum_{i=k+1,\ i\ \text{odd}}^{n-1} P_{i:i-k+1} G_{i-k:i-k-1} \tag{2.23}$$

However this final equation for the detection is still lacking. As has been pointed out in [5] if the adder has an initial carry in input this can be propagated along the network and so its contribution must be taken into account. Considering that the floating point arithmetic unit used in this work performs both addition and subtraction by setting the input carry of the adder to 1 or 0, a term must be added to Eq. 2.23.

$$D = \sum_{i=k+1,\ i \text{ odd}}^{n-1} P_{i:i-k+1} G_{i-k:i-k-1} + P_{k+1:2} P_{k:0} c_{in} \tag{2.24}$$

There is one thing to notice in Eq. 2.24, the term $P_{k:0}$ is evaluated in the last layer of the network, this implies that the *error_ detection* must wait the full delay of the speculation net before evaluating the error. This did not happen earlier because the detection were fully parallel to the last stage of speculation. The aftermaths of this change are detailed in 4.1.1.

Last the ***error_ correction*** network is treated. Its construction is quite simple, it is composed by all the pruned levels of the prefix network that are connected to the last non-pruned level. This networks provides the correct result in parallel with the *fast_ track* of the speculation, usually it has a longer delay but this is not a problem because the exact sum can be evaluated in two clock cycle.

Chapter 3

Floating point adder

3.1 Floating Point addition

Addition is usually the most used FP operation in digital ICs and can be integrated with the subtraction by simply using number in 2's complement format. Addition is theoretically more difficult to perform with respect to multiplication, this because it requires the two exponent to be equal and so an accurate confront is needed.
In general when addition is required between two floating point numbers A and B in the following form[7]:

$$A = (-1)^{s_1} \cdot N_1 \cdot 2^{E_1 - bias}$$

$$B = (-1)^{s_2} \cdot N_2 \cdot 2^{E_2 - bias}$$

With s the sign bit and N the significand and making the assumption that both exponents base is 2, first the condition to be verified is that $2^{E_1} = 2^{E_2}$, if this is not the case than a shift of the significand with the smaller exponent must be performed. Assuming $2^{E_1} > 2^{E_2}$ the significand N_2 is shifted by

$|E_1 - E_2|$ to the right, the choice of the shift rests on the smaller exponent to achieve a significand shorter than 1 and so utilize a smaller adder.
The final result can be computed as:

$$C = ((-1)^{s_1} \cdot N_1 + (-1)^{s_2} \cdot N_2 \cdot 2^{E_1 - E_2}) \cdot 2^{E_1 - bias} \tag{3.1}$$

At the end some adjustments may be needed in order to have a final result compliant to the format. If there is an excess in the significand a shift to the right is needed to achieve the form $1.XXX..$, on the contrary if there are one or more leading zeros (usually in subtraction) a shift to the left is used. The exponent must be shifted accordingly to maintain the correctness of the result. More details on the addition are provided in the following section when the actual arithmetic unit is designed.

Summarizing the main steps for an addition(or subtraction) of two floating point numbers are:

1. Evaluate the difference between the two exponents.
2. Shift the significand of the number with the smaller exponent to the right by $|E_1 - E_2|$.
3. Add the two significand (with the 1. in the front) and set the final exponent equal to the bigger initial exponent.
4. If needed shift the significand to the right or left to match the format used and change the exponent accordingly (normalization).

For the sake of clarification an example of the sum between two number is provided:

$$A = 43.34375$$

$$B = 134.0625$$

The two number are converted to the IEEE754 binary_32 format:

$$A = 01000010001011010110000000000000$$

$$B = 01000011000001100001000000000000$$

The sign is positive in both cases so now the exponent are analyzed:

$$E_1 = 10000100 \qquad\qquad E_2 = 10000110$$

$$E_2 > E_1 \Longrightarrow E_2 - E_1 = 10_2 \rightarrow 2_{10}$$

The exponent of A is smaller that the one of B and so its significand (de-normalized)need to be shifted to the right by 2.

$$N_1 = 1.01011010110000000000000 \Longrightarrow N_1 = 0.01010110101100000000000$$

Now the other significand is de-normalized and the sum is performed:

$$N_1 + N_2 = 1.00001100001000000000000 + 0.01010110101100000000000$$

$$\Downarrow$$

$$N_3 = 1.01100010110100000000000$$

In this case there is no need for furthers shifts of the significand, the final result is:

$$C = 01000011001100010110100000000000 \rightarrow 177.40625_{10}$$

3.2 Arithmetic unit design

The floating point adder described below is taken from [5] because it has already been implemented and validated in the GPU model of [3]. This choice has been made in order to expand the previous work with deeper and larger analysis and provide a compatible architecture for future study on GPU performances.

Following steps described in the previous section in order to design a arithmetic unit that works correctly the main components below are needed:

- **Exponent comparator**. The first step is to find which of the two exponent is bigger, this is performed with a component that compare the two exponents and send the result to the *exponent updater* that handles this result and decides the direction and the shift amount. Three signals are used to pass this informations:*shift_amount* is used to evaluate the difference between the two exponent and so how many shifts are needed, *flag* and *idem* are used to indicate which exponent is bigger or if the two are equal. A table with the possible combination is reported below.

Exponent comparison result	*flag*	*idem*
$\exp_a > \exp_b$	0	0
$\exp_a < \exp_b$	1	0
$\exp_a = \exp_b$	0	1
NaN	1	1

Table 3.1: Table of truth of exponent comparator generated signals

The VHDL description of the component is the following.

```
architecture behavior of exp_comparator is

signal exp_a, exp_b, exp_diff : signed(w downto 0);


begin

exp_a <= signed('0'&exponent_a);
exp_b <= signed('0'&exponent_b);
exp_diff <= exp_a - exp_b;
shift_amount<= unsigned(abs(exp_diff));
flag <= exp_diff(w);
idem <= '1' when exp_diff = 0 else '0';

end behavior;
```

It can be noticed how an extension is needed for both exponents in order to correctly detect the case $\exp_a < \exp_b$.

- **Right shifter**. After the analysis of the exponent the significand needs to be shifted accordingly. The *exponent updater* uses the *flag* signal to drive a multiplexer that select the correct input significand to the shifter and then sends the previous evaluated shift amount that is used by the component to shift the input to the right.

- **Significand swapper**. If when evaluating the sign of the two numbers they are different some more considerations are required. The adder is able to perform only addition, so in order to cope with this problem a 2's complement must be performed on the correct operand. The component drives some multiplexers that change the inputs of the adders from the original to the inverted significand. Then the signal *add_sub* is

set to declare if the operation is a sum (add_sub=0) or a subtraction (add_sub=1), in the last case the $carry\_input$ of the adder is set to 1 to perform the final step of the 2's complement conversion.

- **Adder**. The sum of the two significands is performed using the adders described in the previous chapter. More details on the implementation of the speculative adder with the $error\_network$ that can requires two clock cycles to perform the correct sum is provided in the next section. Now the focus is on when the $carry\_output$ of the adder need to be considered because it can affect the addition result and the final sign evaluation. In general the easiest case is when the two numbers have opposite sign and different exponent, in this case the carry is neglected because the final sign is the one of the number with the biggest exponent. The critical case is when the exponent is the same and the sign is the opposite, if this happens the carry is essential to the sign evaluation because otherwise is not possible to find which number has the highest absolute value. A table with that summarize the possible cases is provided below.

Adder carry out evaluation

Sign comparison	*Exponent comparison*	*Operation on carry out*
equal	equal	disregard if 0, otherwise concatenate to the MSB of the sum
equal	different	disregard if 0, otherwise concatenate to the MSB of the sum
opposite	different	always disregard
opposite	equal	if 1 the final sum sign is 0, otherwise the sign is 1

Table 3.2

- **Sign logic**. The carry out of the adder is not the only variable when evaluating the final sign, but others parameters are needed. The signals used in this component to compute the right sign with c_{out} are: $sign_A$, $sign_B$, $flag$ and $item$. A *boolean function* is created starting from Tab. 3.3 where all the possible combinations of this signals are reported and considering the sum of products (covering the 1s of the table). For the sake of synthesis only the combinations that provides a 1 are illustrated in the table.

c_{out}	$sign_A$	$sign_B$	$flag$	$idem$	$finalsign$
0	1	0	0	0	1
1	1	0	0	0	1
0	1	1	0	0	1
1	1	1	0	0	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	1	1	0	1	1
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	1	0	1
1	1	1	1	0	1

Table 3.3: Truth table for final sign evaluation

Once again is visible how the carry out is critical for the final sign evaluation, as written in Tab. **??** if the two exponents are equal and the two signs are opposite all the decision are taken based on the carry value. If

the carry is 1 no further actions are needed, otherwise this mean that the sign is negative and another operation must be performed which is 2's complement the result before passing it to the next component.

- **Normalizer**. This element has a fundamental importance because is responsible for shifting the sum result until the standard form $1.XXX$ is obtained. A VHDL description is provided for a better understanding.

```
architecture behavior of normalizer is

begin

normalization: process(raw_result, add_sub, c_out)

variable counter : integer := 0;

begin
if add_sub = '1' then
        for i in p-1 downto 0 loop
                if raw_result(i) = '0' then
                counter := counter + 1;
                end if;
                if raw_result(i) = '1' then
                exit;
                end if;
        end loop;
        norm_shifts <= to_unsigned(counter,w);
        right_left <= '1';
        norm_significand <= std_logic_vector
                (shift_left(unsigned(raw_result),
                counter));
elsif add_sub = '0' then
        if c_out(0) = '1' then
```

```
            norm_shifts <= to_unsigned(1,w);
            right_left <= '0';
            norm_significand <= c_out(0)&raw_result
                    (p-1 downto 1);
            elsif c_out(0) = '0' then
            norm_shifts <= to_unsigned(0,w);
            right_left <= '0';
            norm_significand <= raw_result;
            end if;
end if;
counter := 0;
end process;
```

The first operation is the *leading zero count*. If a subtraction is performed there is a possibility that there is no 1 in the MSB position, so a shift to the left is performed based on how many 0s are in the MSBs position before a 1 appears. In case of an addition the carry out must be take into account because an overflow could occur and so a shift to the right is needed.

- **Exponent Update**. This component mentioned before is now responsible for the shifting of the final exponent accordingly to the shift performed by the normalizer on the final sum.
- **Exceptions detector**. To provide a complete architecture able to handle all the particular cases a final component is required, its VHDL description is provided below for the binary_16 format.

```
architecture behavior of exceptions is

constant zero_cmp : std_logic_vector(p-1 downto 0)
                                    := (others => '0');
```

```
signal zero_check : std_logic;

begin

zero_check <= '1' when ((sum = "00000000000" and
    c_out = '1') or (exp_a = "00000" and exp_b
    = "00000")) else '0';

zero_process: process(zero_check, add_sub, idem,
                                    exp_a, exp_b)
begin
 zero <= '0';
 if (add_sub = '1' and idem = '1' and
        zero_check = '1') then
  zero <= '1';
  actual_exp <= "00000";
  actual_res <= "00000000000";
  end if;
 if (exp_a = "00000" and exp_b = "00000") then
    zero <= '1';
  actual_exp <= "00000";
  actual_res <= "00000000000";
 end if;
 if (exp_a = "00000" and not(exp_b = "00000")) then
   zero <= '0';
  actual_exp <= operandB(k-2 downto k-w-1);
  actual_res <= operandB(k-w-1 downto 0);
 end if;
 if (not(exp_a = "00000") and exp_b = "00000") then
  zero <= '0';
  actual_exp <= operandA(k-2 downto k-w-1);
```

```
  actual_res <= operandA(k-w-1 downto 0);
  end if;
 end process;

nan_process: process(exp_a, exp_b)
 begin
 nan <= '0';
 if(exp_a = "11111" or exp_b = "11111")then
 nan <= '1';
 actual_exp <= "11111";
 actual_res <= sum;
 end if;
 end process;

ovf_process: process(add_sub, c_out, exp_a, exp_b)
 begin
 ovf <= '0';
 if (add_sub = '0' and c_out = '1' and exp_a =
    "11110" and exp_b = "11110") then
  ovf <= '1';
  actual_exp <= "11110";
  actual_res <= "11111111111";
 end if;
end process;

unf_process:  process(add_sub, c_out, exp_a, exp_b)
 begin
 unf <= '0';
 if (add_sub = '1' and c_out = '0' and exp_a =
    "00001" and exp_b = "00001") then
 unf <= '1';
```

```
 actual_exp <= "00001";
 actual_res <= "10000000000";
 end if;
end process;

default_process: process(add_sub, idem, c_out,
                         exp_a, exp_b)
begin
 if not(add_sub = '1' and idem = '1' and zero_check
     = '1') and not(exp_a = "00000" and exp_b =
     "00000") and not(exp_a = "11111" or exp_b =
     "11111") and not(add_sub = '0' and c_out = '1'
     and exp_a = "11110" and exp_b = "11110") and
     not(add_sub = '1' and c_out = '0' and exp_a =
     "00001" and exp_b = "00001") and not((exp_a =
     "00000" and not(exp_b = "00000")))and not((not
     (exp_a = "00000") and exp_b = "00000")) then
  actual_exp <= exponent;
  actual_res <= sum;
 end if;
end process;

end behavior;
```

There are four main *"exceptions"* that must be handled accordingly: **nan**, **overflow**, **underflow**, **zero** and they are quite straightforward. The first is the case when one or both inputs are not numbers, overflow and underflow means that the final results exceed the upper and lower bound of the numbers that can be represented using the format, lastly is the zero case that requires a more detailed description. Three possibilities are possible: one of the operands is zero and so the result

is equal to the other operand, both operand are zero and so the result is zero, the result of two non-zero operand is zero. If any of this cases occurs a warning signal is generated.

Others additional standard components (i.e. registers, multiplexers, inverters) are used to complete the architecture. A scheme including the main blocks is in Fig. 3.1.

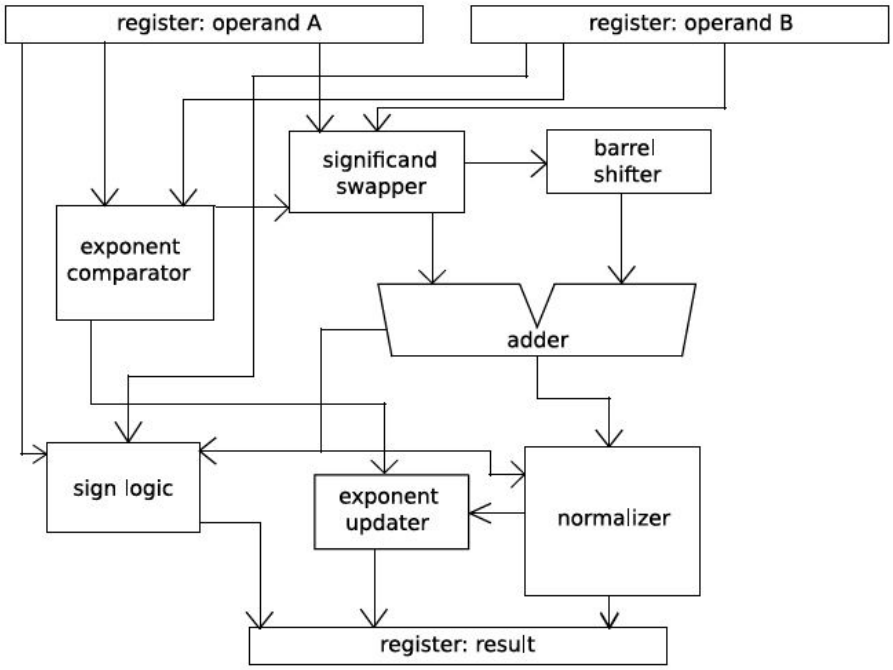


Figure 3.1: Floating point arithmetic unit basic blocks

3.3 Variable Latency Adder Implementation

The implementation of the speculative adder without *error_ network* does not require particular attentions, it can be simply replace the exact adder if the input and output signals described in the same way, it is a different story for the speculative adder with the *error_ network*. This last solution always provides a correct result exploiting the variable latency approach. As said before when the error in the result is detected a signal is generated and the architecture requires another clock cycle to provide the correct result. In this second clock cycle the output must switch to the exact one and the inputs must remain constant because the adder evaluate the correct result in parallel with the speculative one, it only need more time and so the critical path considered is always the one of the speculative path. The architecture need some modifications[17] to handle this situation, the adder along with the additional logic circuit is provided in Fig.3.2.

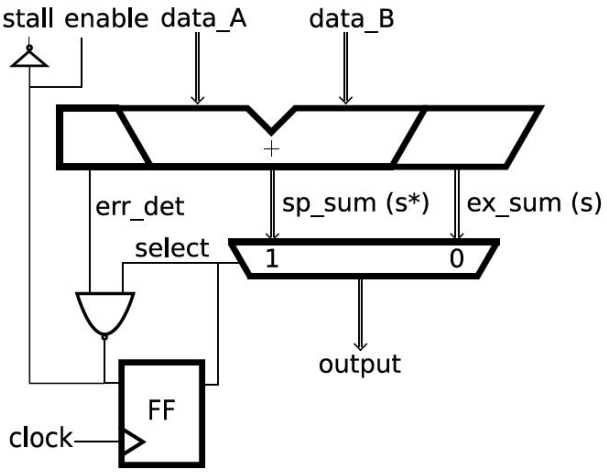


Figure 3.2: Variable latency adder along with the components needed for the implementation in the arithmetic unit.

The components added are a multiplexer to select between the speculative and exact sum, a NAND gate, an INVERTER and a Flip-Flop. The behavior

can be understood better with the timing diagram in Fig.3.3. In the default case the multiplexer *select* signal is set to 1 and the speculative sum is the output, then when an error is detected the steps followed are:

1. The detection network is triggered by an error and the *err_det* signal is asserted.
2. The NAND gate inputs *err_det* and *select* are both 0 so the *enable* is set to 1.
3. The *enable* signal is memorized in the Flip-Flop and is sent to the input registers to block the inputs.
4. The *stall* signal is generated inverting the *select* and is sent to the extern to inform that the input data must be stopped.
5. In the second clock cycle the Flip-Flop sets the *select* to 0 and so the multiplexer switch the output to the exact sum.
6. The *select* signal in input to the NAND assert the *enable* signal and so default situation is restored.

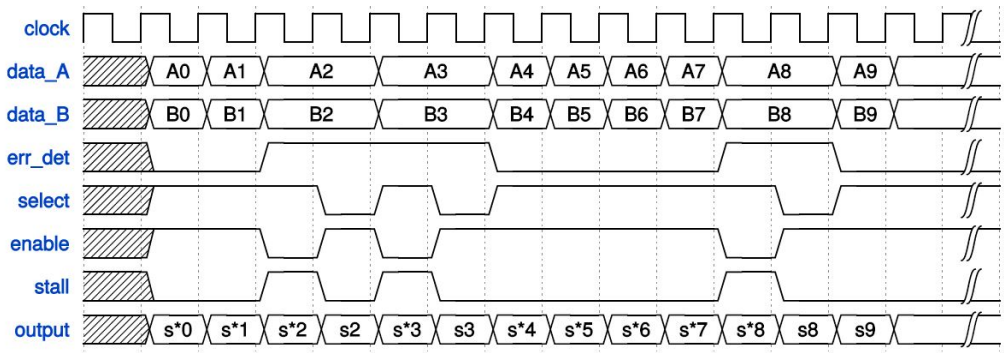


Figure 3.3: Timing diagram of variable latency adder

The Flip-Flop is used to maintain for a clock cycle the exact sum as output and then switch back to the speculative sum so that the exact sum is never set as the output for more than 2 clock cycle.
The architecture need also another "*exceptions*" signal to report that a data

is wrong and so must be discharged. For this purpose *inexact* signal is introduced along with the previous mentioned signals. All the "*exception*" were used in previous work as flag that are used in GPU models to highlight special cases.

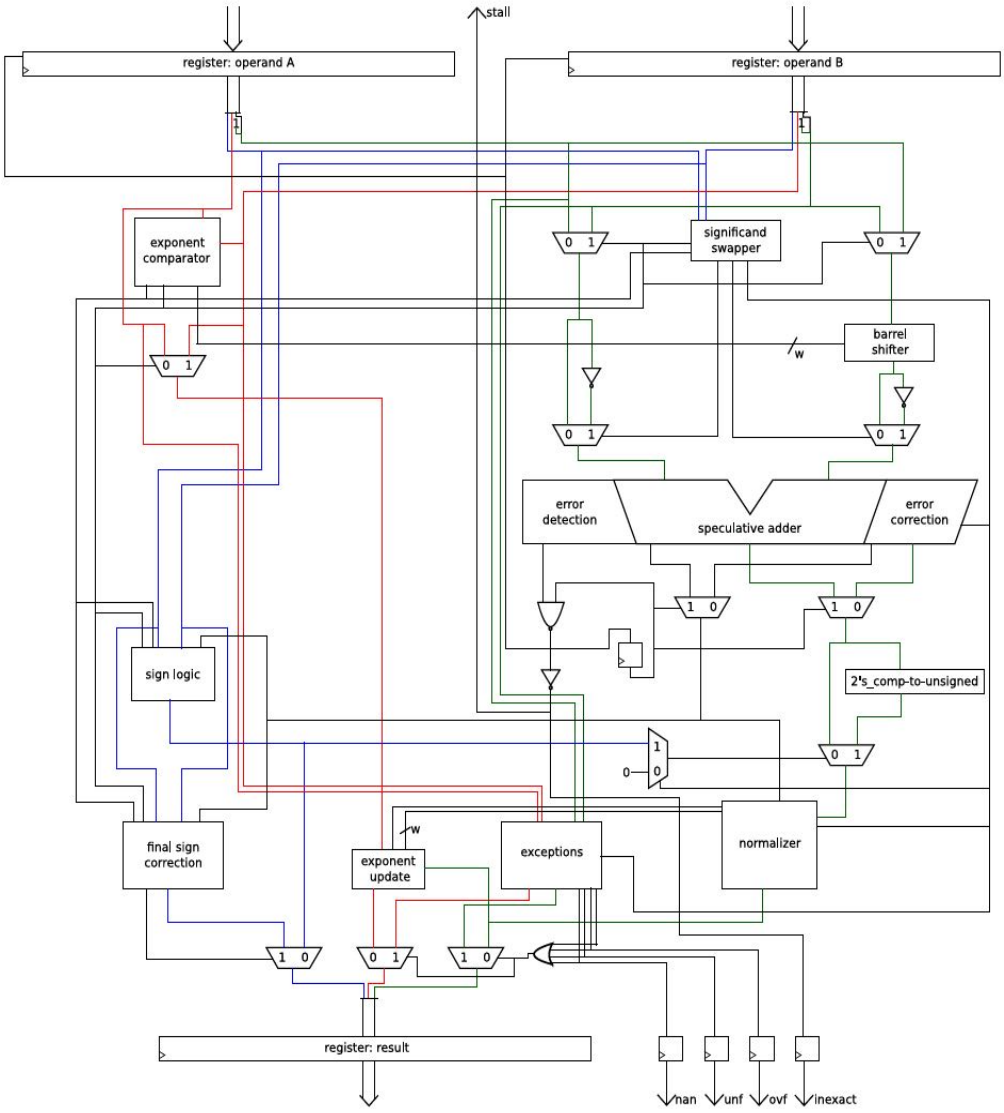


Figure 3.4: RTL representation of the floating point arithmetic unit.

Finally in Fig.3.4 a complete Register Lever representation of the full arith-

metic unit is provided. The signal are classified as follow:

- *Green wires*: *std_logic_vectors* with a width of p-bit with p the precision parameter of the corresponding IEEE754 format.
- *Red wires*: *std_logic_vectors* with a width of w-bit with w the exponent parameter.
- *Blue wires*: *std_logic* that represent the sign bit.
- *Black wires*:internal wires with 1-bit width or more, as specified on the wire.

Chapter 4

Results

In this Chapter all the results for the designed architecture are presented divided in three parts. First the characterization of the sole adder component is provided along with the results of delay, area and power generated from the synthesis. Then in the second part the same is done with the frequency, area and power for the complete arithmetic unit. Last an evaluation and analysis of errors in the speculative architecture for all the parallelism of the IEEE standard is provided.

In the characterization process two main programs were used. For the functional simulation of the components *Modelsim*® developed by Mentor Graphics and for the synthesis *Design compiler*® by Synopsys. *Modelsim* does not require particular settings for the simulation but for *Design compiler* a more detailed description of the setup must be provided.
Two libraries area specified: the **uk65lscllmvbbr_120c25_tc** library, that contains the descriptions of the UMC 65nm Low-K Multi-voltage Low Leakage RTV Tapless Standard cells used in the synthesis, and the **DesignWare Foundation** library from Synopsys which allow *Design Compiler* to identify basic components in the design (e.g. multiplexers, flip-flops, counters, ecc.) in order to provide the best configuration for a more efficient implementation.
Others setting like RTL names preservation, clock period and uncertainty

and inputs/outputs delays are used to provide a more better synthesis result, but in order to have a more realistic characterization a load must be set for the output of the cells. The load must be reasonable and usually a buffer is used, setting a specific load for each cell is a difficult task, so an alternative and realistic solution is to use the same load for all outputs. Analyzing the documentation of the cells library a good solution is using the *BUFFM4R* buffer as load.

For the sake of clarification a list of all the steps used in the characterization of the adder and the complete arithmetic unit is provided:

- functional simulation with *Modelsim* using VHDL test benches created ad hoc.
- synthesis with *Design Compiler* that creates a Verilog netlist of the design with the files to perform the power analysis and provides information on delay and area.
- the generated netlist and files are used along with a Verilog test bench in *Modelsim* to record the switching-activity details on a file.
- the previous file with the switching activity is used a last time in *Design Compiler* to generate the final report on the power consumption.

For the error analysis test vectors were generated using a C program. The program use the rand() function to create a random integer number, then the operator % is used to take the *reminder* of the division by 2 that can have has results only 1 and 0. The vectors are provided as input to the architecture and then the results are confronted with the ones generated by another C program that simulates the same behavior of the exact architecture.

4.1 Adder characterization

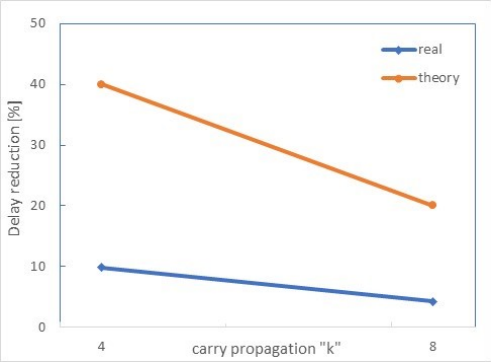
4.1.1 Timing

First the delay of the various adder architecture are analyzed. In table 4.1 are the results for the speculative architecture without the *error_network* and the ones for the exact architecture to provide a confront between the two.

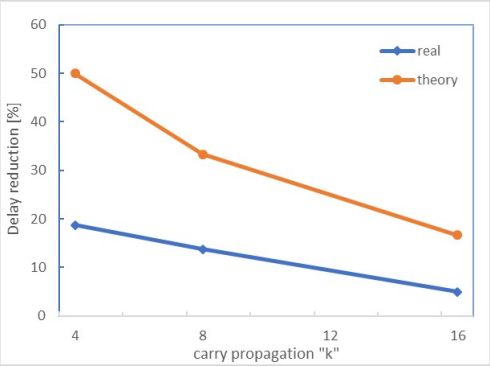
Adder Delay [*ps*]							
Width		*Exact*	*Speculative without error_network*				
n	p		K=4	K=8	K=16	K=32	K=64
16	11	700	640	680			
32	24	800	650	690	760		
64	53	900	640	690	760	830	
128	113	970	660	690	760	830	920

Table 4.1: Maximum delay confront between exact and speculative adder without *error_network*.

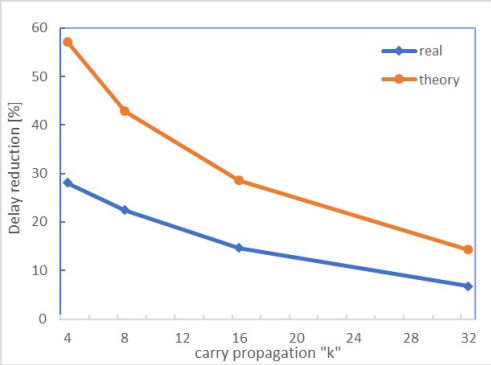
This results were confronted with the theoretical ones in (table cap 2 ref) and the delay improvements for the two cases area represented in Fig.4.1.



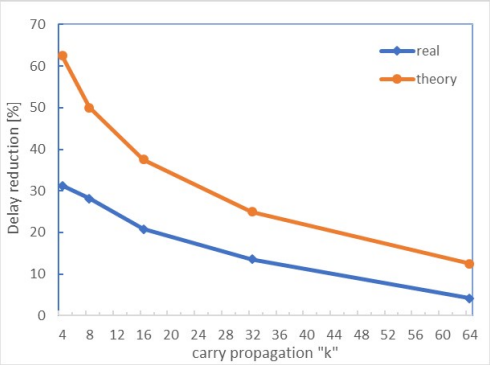
(a) Delay reduction for p=11



(b) Delay reduction for p=24



(c) Delay reduction for p=53



(d) Delay reduction for p=113

Figure 4.1: Confront between theoretical and real delay reduction percentage of the speculative adder.

As it can be seen the trend of the real curves in all the parallelism is similar to the theoretic curves, this is more evident for high precisions because they offer a better resolution thanks to the more levels of speculation possible. The noticeable thing is that the gain for the same level of speculation increase for higher levels of parallelism, this because there are more pruned levels. The major drawback visible is the lower reduction of the delay in the real case. This is justified by the real components used in the synthesis and by the fact that the theoretical evaluation is based only on the reduction of levels in the *prefix computation* part of the PPA.

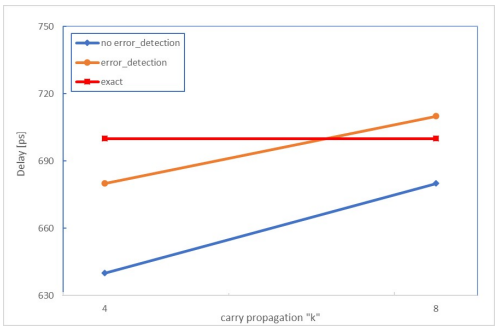
Adder Delay [*ps*]

Width		*Exact*	*Speculative with error_network*				
n	p		K=4	K=8	K=16	K=32	K=64
16	11	700	680	710			
32	24	800	750	760	800		
64	53	900	800	830	890	940	

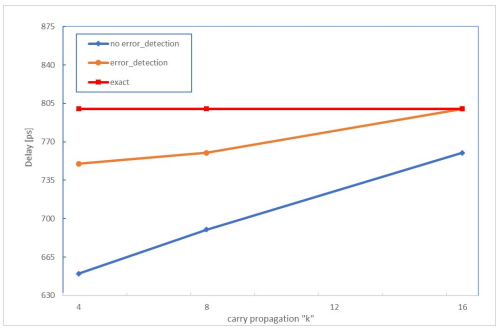
128	113	970	850	930	960	990	980

Table 4.2: Maximum delay confront between exact and speculative adder with *error_ network*

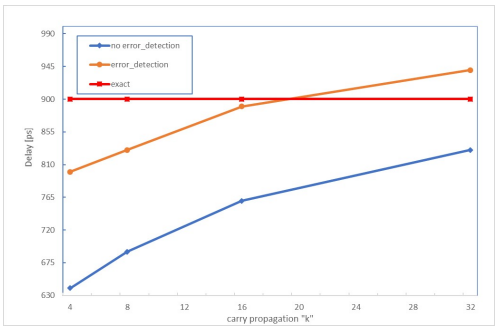
In table 4.2 are represented the delay results for the speculative adder that implements the *error_ network*. The delays for all the architecture of the adder are represented in Fig.4.2 in order to compare the different performances and offer a better analysis on the differences with respect to the exact solution.
It can be noticed that all the speculative solutions without *error_ network* provide clearly better performances in therm of delay in comparison to the exact and speculative adder with the *error_ network*, the shorter the propagation of the carry in the speculation, the better is the delay. It may seem that this is the better solution for an application, but another parameter must be take into account and that is the possibility of a wrong results. The analysis of the error in the speculative architecture is provided in 4.3.
For what concerns the adder that includes the *error_ network* the results are more interesting. Low values of k (4 for p=11 and $\leq$ 16 for the others) provides a smaller delay than the exact case, but when the speculation decrease the critical path became equal or even longer than the standard architecture, the cause can be found analyzing the synthesis reports. This behavior is caused by the carry in of the adder, theoretically the part of the architecture that provide the *error_ detection* should decrease its critical path when k increases as seen in section 2.3. But when an adder with a carry in is considered the situation changes a bit, because the *error_ network* need a value from the last stage of the speculative *fast_ track* to correctly evaluate the presence of an error. The final delay in this case is $T_{sum\_spec} + T_{detection}$, if there was not a carry in it would have been $max(T_{sum\_spec}, T_{detection})$. For high k the speculative sum path is close to the exact one, if the overhead of the detection is included the critical path ends to be longer.
Last it can be seen how the speculative curves with and without the *er-*



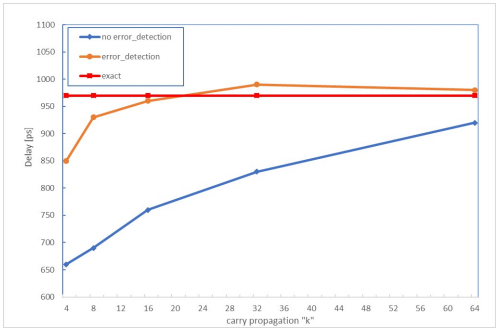
(a) Delay for p=11



(b) Delay for p=24



(c) Delay for p=53



(d) Delay for p=113

Figure 4.2: Confront between the delays of exact adder and speculative adder with and without *error_ network*.

ror_ network tend to be closer when k increases, this confirms how the delay of the detection net decreases for low speculation levels.

4.1.2 Area

In table 4.3 are represented the results for area occupancy of the exact adder and the speculative adder without *error_ network*.

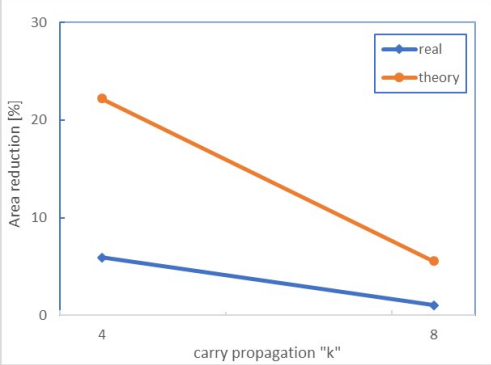
Adder Area								
Width		*Quantity*	*Exact*	*Speculative without error_ network*				
n	p			K=4	K=8	K=16	K=32	K=64
		# ports	265	241	259			
		# nets	372	365	367			

		# cells	143	149	152			
16	11	references	6	6	6			
		C-area [µm^2]	215.64	180.48	209.16			
		NC-area [µm^2]	378	378	378			
		T-area [µm^2]	593.64	558.48	587.16			
		# ports	649	517	577	625		
		# nets	936	711	798	873		
		# cells	374	269	304	335		
32	24	references	6	6	6	6		
		C-area [µm^2]	569.88	407.52	462.24	517.68		
		NC-area [µm^2]	799.2	799.2	799.2	799.2		
		T-area [µm^2]	1369.08	1206.72	1261.44	1316.88		
		# ports	1567	1123	1267	1399	1507	
		# nets	2227	1714	1751	1938	2114	
		# cells	871	658	667	740	818	
64	53	references	6	6	6	6	6	
		C-area [µm^2]	1370.88	925.08	1028.52	1147.32	1277.64	
		NC-area [µm^2]	1738.8	1738.8	1738.8	1738.8	1738.8	
		T-area [µm^2]	3109.68	2663.88	2767.32	2886.12	3016.44	
		# ports	3691	2383	2707	3019	3307	3547
		# nets	5234	3289	3727	4155	4568	4941
		# cells	2048	1247	1413	1577	1742	1899
128	113	references	6	6	6	6	6	6
		C-area [µm^2]	3223.08	1927.8	2189.16	2449.8	2716.2	2979.72
		NC-area [µm^2]	3682.8	3682.8	3682.8	3682.8	3682.8	3682.8
		T-area[µm^2]	6905.88	5610.6	5871.96	6132.6	6399	6662.52

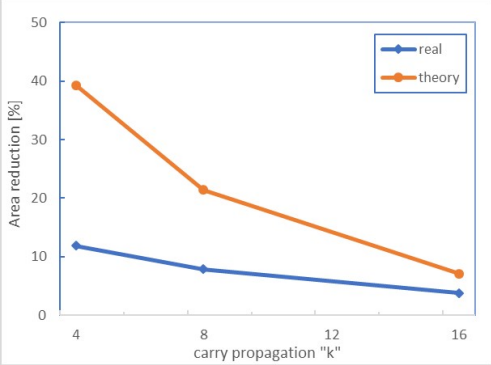
Table 4.3: Area occupancy of exact and speculative adder without *error_network*. C stands for "Combinational", NC for "Non-Combinational" and T for "Total".

As for the delay confront between theoretical and real area reduction for the speculative adder is represented in 4.3. It can be observed that like in the previous case the two behavior have the same trend, for low k the bigger

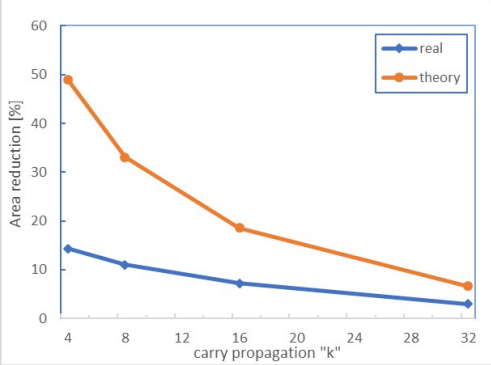
number of pruned levels cause a smaller area. Like for the delay, real area gain is smaller than the theoretical one for the same reasons.



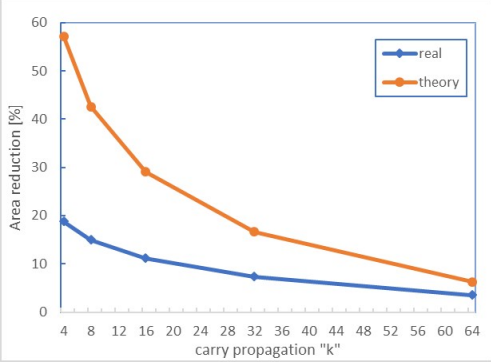
(a) Area reduction for p=11



(b) Area reduction for p=24



(c) Area reduction for p=53



(d) Area reduction for p=113

Figure 4.3: Confront between theoretical and real area reduction percentage of the speculative adder.

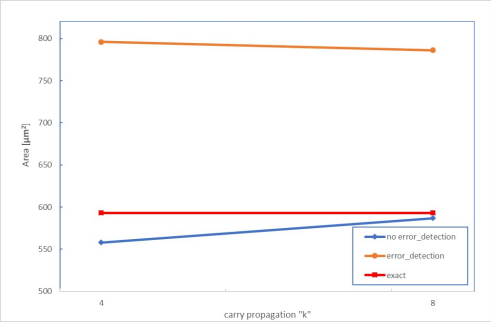
Area occupancy of speculative adder with *error_network* in comparison to exact is in Table 4.4.

Adder Area								
Width		*Quantity*	*Exact*	*Speculative with error_network*				
n	p			K=4	K=8	K=16	K=32	K=64
		# ports	265	356	356			
		# nets	372	483	481			

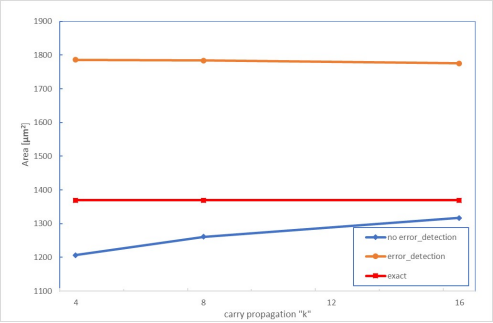
		# cells	143	187	184			
16	11	references	6	9	9			
		C-area [µm^2]	215.64	277.92	267.84			
		NC-area [µm^2]	378	518.4	518.4			
		T-area [µm^2]	593.64	796.32	786.24			
		# ports	649	828	828	828		
		# nets	936	1149	1149	1152		
		# cells	374	457	455	454		
32	24	references	6	9	9	9		
		C-area [µm^2]	569.88	706.68	704.52	695.52		
		NC-area [µm^2]	799.2	1080	1080	1080		
		T-area [µm^2]	1369.08	1786.68	1784.52	1775.52		
		# ports	1567	1592	1592	1592	1592	
		# nets	2227	2717	2730	2740	2788	
		# cells	871	1084	1095	1101	1141	
64	53	references	6	9	9	9	9	
		C-area [µm^2]	1370.88	1733.04	1734.84	1716.48	1759.32	
		NC-area [µm^2]	1738.8	2332.8	2332.8	2332.8	2332.8	
		T-area [µm^2]	3109.68	4065.84	4067.64	4049.28	4092.12	
		# ports	3109.68	4496	4496	4496	4496	4496
		# nets	5234	6239	6235	6238	6250	6261
		# cells	2048	2476	2470	2469	2473	2468
128	113	references	6	9	9	9	9	9
		C-area [µm^2]	3223.08	3985.92	3973.68	3962.52	3946.32	3910.68
		NC-area [µm^2]	3682.8	4924.8	4924.8	4924.8	4924.8	4924.8
		T-area[µm^2]	6905.88	8910.72	8898.48	8887.32	8871.12	8835.48

Table 4.4: Area occupancy of exact and speculative adder with *error_network*. C stands for "Combinational", NC for "Non-Combinational" and T for "Total".

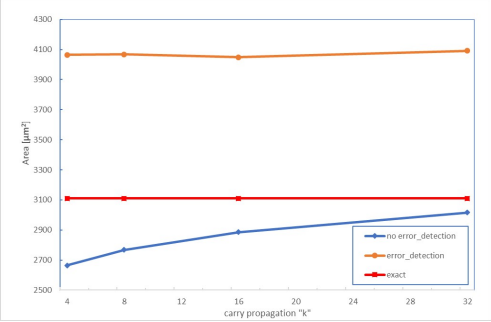
In 4.4 the area for all the configuration of the adder is represented. As seen before without the *error_network* the architecture is always smaller than the other cases, reintroducing levels makes the area rise till nearly the ex-



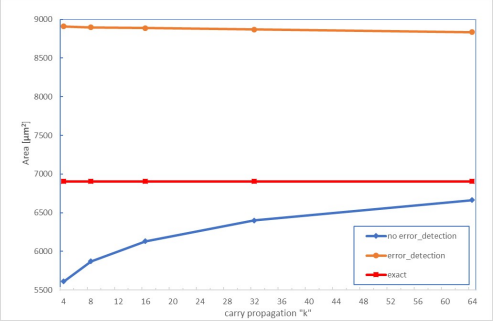
(a) Area for p=11



(b) Area for p=24



(c) Area for p=53

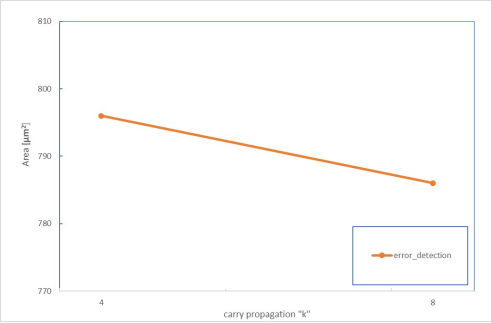


(d) Area for p=113

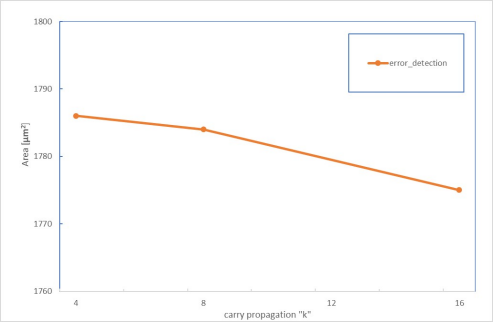
Figure 4.4: Confront between the area of exact adder and speculative adder with and without *error_network*.

act adder. If the *error_network* is introduced the area increases as expected and is always the biggest, this is caused by the *error_detection* and *error_correction* that provide a big overhead of about 23% with respect to the exact architecture.

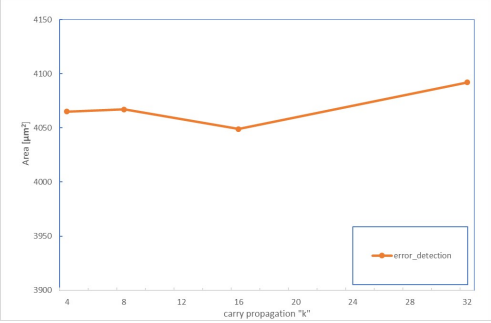
A zoom on this last curve is provided in Fig. 4.5 for a better understanding. The expected behavior is confirmed because the trend of the area is to decrease with the increase og k, this because the detection net needs less elements in order to find if there is an error. This results is the opposite of the one without *error_network*. The only exception is for the double precision case with k=32 where the total area increases and reach the biggest value, this should be because *Design Compiler* adopts a different strategy for the synthesis to improve the performances.



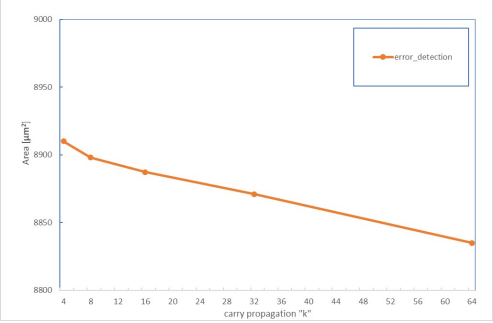
(a) Area for p=11



(b) Area for p=24



(c) Area for p=53



(d) Area for p=113

Figure 4.5: Zoom on the area behavior of the speculative adder with *error_ network*.

4.1.3 Power

Below are reported respectively the power consumption for the exact and speculative adder with (Table 4.5) and without (Table 4.6) *error_ network* obtained from the synthesis and switching activity.

Adder Power

Width		*Quantity*	*Exact*	*Speculative without error_ network*				
n	p			K=4	K=8	K=16	K=32	K=64
		CI-power [µW]	97.0488	90.997	95.6948			
16	11	NS-power [µW]	14.8925	14.8021	14.265			
		TD-power [µW]	11.937	105.7991	44.7868			
		CL-power [nW]	50.8677	44.7868	49.7388			

		CI-power	213.252	200.938	204.98	209.7357		
32	24	NS-power [µW]	35.419	27.4835	30.3047	33.14		
		TD-power [µW]	248.67	228.4215	235.286	242.8757		
		CL-power [nW]	115.29	105.1089	108.993	112.4961		
		CI-power [µW]	475.887	386.7843	392.912	401.7049	445.473	
64	53	NS-power [µW]	83.0988	66.7853	68.0098	71.2412	76.1693	
		TD-power [µW]	558.986	453.5696	460.921	472.9461	521.642	
		CL-power [nW]	263.544	234.124	240.119	248.3568	257.1	
		CI-power	1030.3	897.68	902.778	898.2141	991	1012.7
128	113	NS-power [µW]	187.689	108.4927	111.8	120.4226	162.126	175.18
		TD-power [µW]	1218	1006.173	1014.58	1018.6	1153.1	1187.8
		CL-power [nW]	580.538	498.6483	509.052	521.689	548.647	566.673

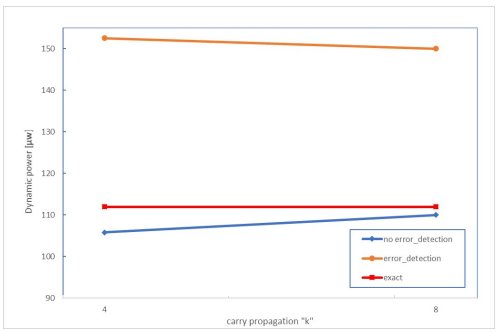
Table 4.5: Power consumption of exact and speculative adder without *error_ network*. CI stands for "Cell Internal", NS for "Net Switching", TD for "Total Dynamic" and CL for "Cell Leakage".

Adder Power								
Width		*Quantity*	*Exact*	*Speculative with error_ network*				
n	p			K=4	K=8	K=16	K=32	K=64
		CI-power [µW]	97.0488	132.509	130.66			
16	11	NS-power [µW]	14.8925	19.964	19.3148			
		TD-power [µW]	11.937	152.4732	149.975			
		CL-power [nW]	50.8677	67.8745	66.3543			
		CI-power	213.252	288.231	283.345	280.789		
32	24	NS-power [µW]	35.419	47.2822	47.4872	46.742		
		TD-power [µW]	248.67	335.513	332.832	327.531		
		CL-power [nW]	115.29	152.628	150.879	148.181		
		CI-power [µW]	475.887	516.225	517.858	507.231	495.271	
64	53	NS-power [µW]	83.0988	74.1009	65.7405	71.0583	77.3791	
		TD-power [µW]	558.986	590.325	583.599	578.29	572.65	
		CL-power [nW]	263.544	349.137	347.731	345.923	344.579	

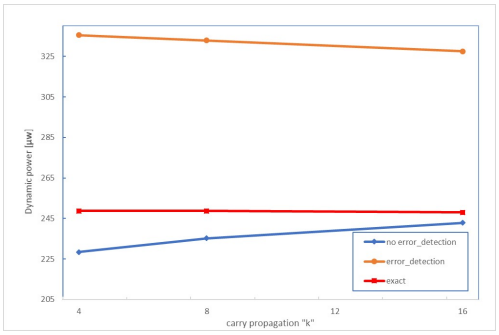
	CI-power	1030.3	1408.4	1378.6	1375.3	1365	1354.6
128 113	NS-power [µW]	187.689	237.97	244.689	244.2	240	230.896
	TD-power [µW]	1218	1646.37	1623.3	1619.5	1605	1585.5
	CL-power [nW]	3109.68	772.206	758.964	751.52	748.238	743.518

Table 4.6: Power consumption of exact and speculative adder with *error_network*. CI stands for "Cell Internal", NS for "Net Switching", TD for "Total Dynamic" and CL for "Cell Leakage".

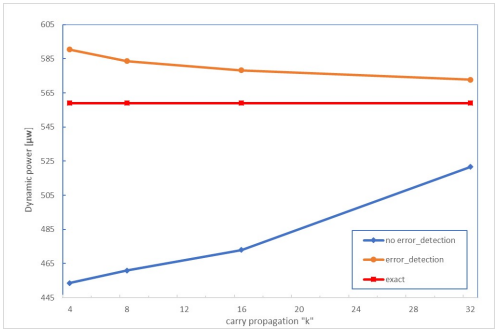
In Fig. 4.6 is seen how or the speculative adder without *error_network* the dynamic power rises with k.



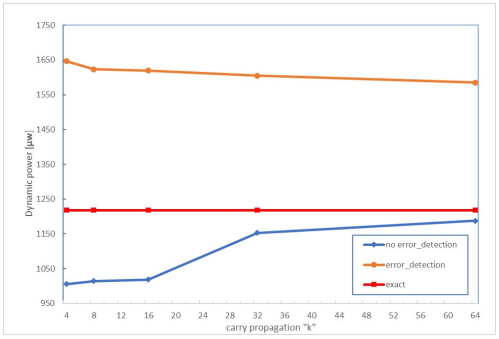
(a) Dynamic power for p=11



(b) Dynamic power for p=24



(c) Dynamic power for p=53

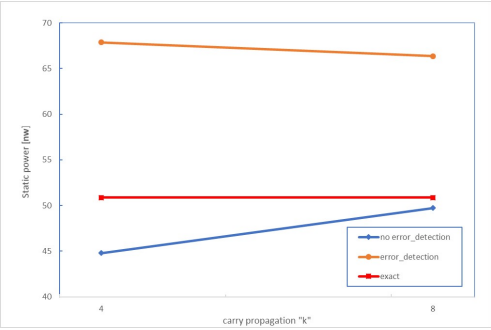


(d) Dynamic power for p=113

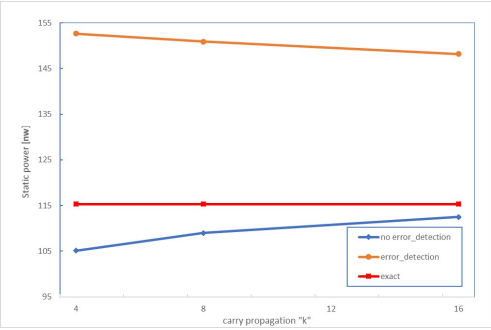
Figure 4.6: Confront between the dynamic power of exact adder and speculative adder with and without *error_network*.

This results is quite obvious because some levels are reintroduced, but the power always remains lower than the exact case. When the *error_network*

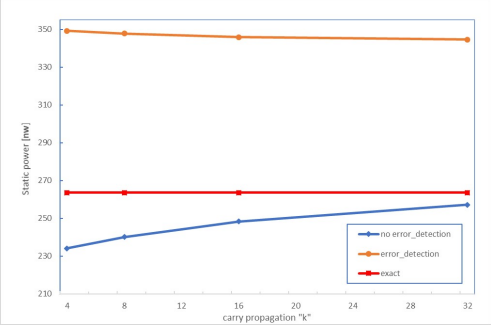
is introduced the power decrease with k, this behavior follow the one of the area and the cause is the same. The decrease of elements in the detection network cause an overall drop of switching activity.



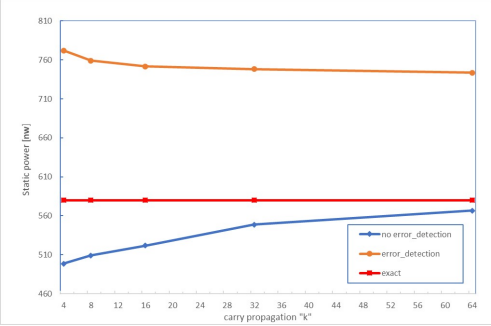
(a) Static power for p=11



(b) Static power for p=24



(c) Static power for p=53



(d) Static power for p=113

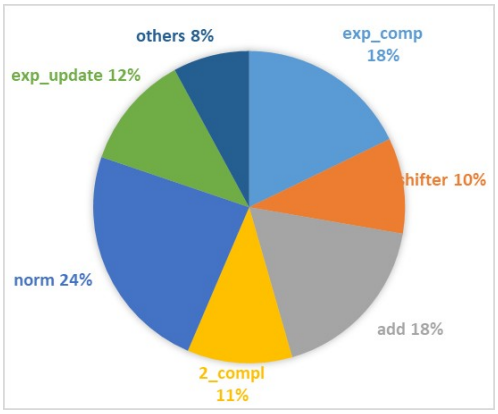
Figure 4.7: Confront between the static power of exact adder and speculative adder with and without *error_ network*.

The static power of the leakage has the same trend(Fig.4.7) but its weight in the total dynamic consumption is negligible because it is an order of magnitude smaller than the dynamic power.

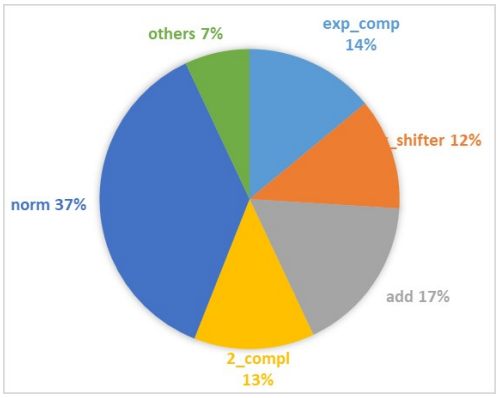
4.2 Arithmetic unit characterization

4.2.1 Pipelining

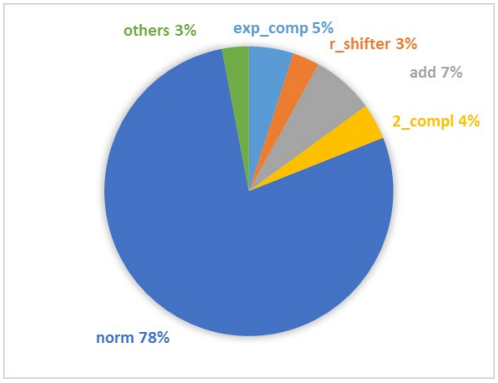
Before substituting the exact significant adder with the speculative one in the full arithmetic unit a first analysis of the architecture was performed to find the contribution to the critical path of each component. The results are inserted in pie charts (Fig.4.8) to provide a better understanding.



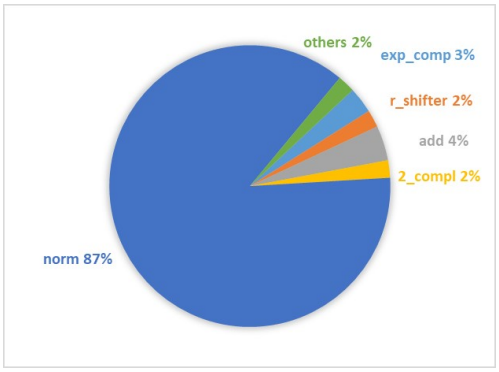
(a) Division of delay binary_16



(b) Division of delay binary_32



(c) Division of delay for binary_64



(d) Division of delay for binary_128

Figure 4.8: Main contributions to the delay of the exact floatingpoint arithmetic unit for all the IEEE754 formats.

What emerges from the charts is that starting from the binary_32 format the component that contributes most to the delay is the normalizer, its weight rises with the parallelism of data and makes up for nearly 90% of the total delay for binary_128 format.
This result can be counterproductive to the study performed because the gain in substituting the adder in the architecture is almost negligible. In the literature can be found some solutions to improve the normalizer performances, but this implies to re-design all the arithmetic unit and so this approach was discharged because it would require a separate thesis work.
Another solution that comes to mind is performing pipelining on the component normalizer increasing the latency but reducing the critical path to one comparable with the delay of the other components, with emphasis on the adder. Performing a manual pipelining was put aside for two reason: first architecture of the normalizer is described in a behavioral way in VHDL so it is not possible find how are the internal connections, secondly even if it was possible to find the internal structure the full arithmetic unit is quite complex and so find a *cut-set* is nearly impossible. In the end the problem was solved by relying on some functions provided by *Design Compiler*, thanks to them it is possible to add some spare registers at the output of the arithmetic unit and leave all the pipelining work to the program. Fist some spare registers are added at the output of the arithmetic units before the real output registers, then after a first *compile* it is issued the command *balance_registers* that moves the sequential components of the design to obtain a better cycle time. In order to avoid that the input and output register are moved and that other components besides the normalizer are pipelined, the attribute *dont_touch* is applied.
Pipe levels are added until the critical path is changes and does not include the normalizer, but only the adder and others elements. The final pipelining is resented in Tab.4.7.

Pipelining levels	
IEEE754	*Levels*
binary_16	2
binary_32	3
binary_64	7
binary_128	16

Table 4.7: Number of pipelining levels added to the arithmetic unit to decrease the normalizer critical path

4.2.2 Timing

This section continues with an analysis of the performances of the arithmetic unit using the exact adder and the two configurations of speculation for all four data parallelism of the IEEE754 standard. In Tab.4.8 and Tab.4.9 are the results of maximum achievable frequency for various architectures, this first quantity is the one that provides the most interesting outcomes with respect to the ones from the previous section.

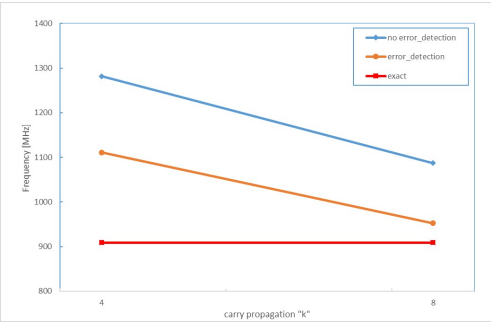
Arithmetic unit Frequency [*MHz*]						
IEEE754	*Exact*	*Speculative without error_ network*				
		K=4	K=8	K=16	K=32	K=64
binary_16	909.10	1282.05	1086.96			
binary_32	404.86	485.44	462.96	444.45		
binary_64	252.53	289.86	267.38	263.16	253.17	
binary_128	136.61	152.21	147.49	142.25	140.45	138.89

Table 4.8: Maximum frequency confront between exact and speculative arithmetic unit without *error_ network*

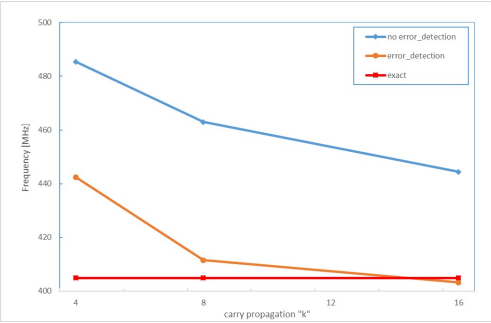
Arithmetic unit Frequency [*MHz*]						
Width	*Exact*	*Speculative with error_ network*				
		K=4	K=8	K=16	K=32	K=64
binary_16	909.10	111.11	952.38			
binary_32	404.86	442.48	411.52	403.23		
binary_64	252.53	272.48	259.74	253.8	252.525	
binary_128	136.61	150.6	145.35	141.24	139.67	138.31

Table 4.9: Maximum frequency confront between exact and speculative arithmetic unit with *error_ network*.

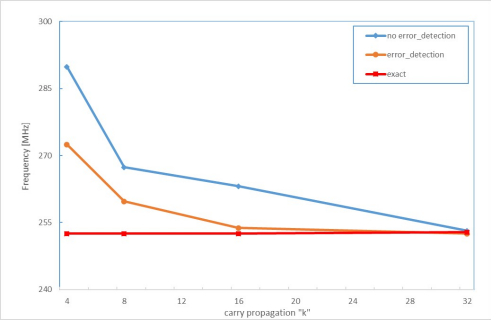
The behaviors of the unit are better visible in Fig. 4.2.2. The first thing to notice is that the arithmetic unit that uses the speculative adder without *error_ network* offers a frequency quite higher than the architecture with the exact adder, this results respect the one from the sole adder analysis. The only exception is the binary_64 case with k=32 where the frequency is the same of the exact unit, this can be justified by the different synthesis performed by *Design Compiler*. The most remarkable results concerns the architecture with the *error_ network*.



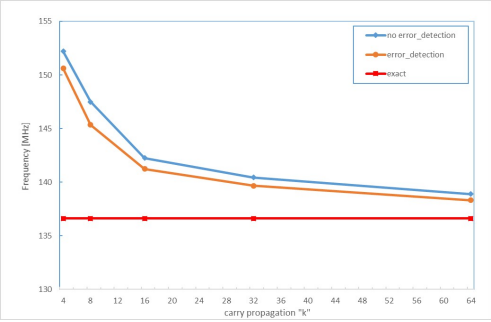
(a) Frequency binary_16



(b) Frequency binary_32



(c) Frequency for binary_64



(d) Frequency for binary_128

Figure 4.9: Confront between frequency of exact floating-point arithmetic unit and the speculative ones with and without *error_network*.

For high levels of speculation (smaller k) an higher frequency was expected, but for the other cases the results are always better or at most equal to the exact unit contrary to what happened in the single adder. Another thing to notice is that the frequency tend to approach the other speculative architecture for larger parallelism. This two results have the same explanation. For low parallelism the critical path goes through the detection network and so the results are equal to the single adder analysis, but as said before the pipe levels used to reduce the overhead of the normalizer are not applied to the other components, so when the length of data grow the delay of the components after the adder grows. When this delay overcome the one from the detection there is a shift on the critical path, this path equal for both speculative architecture because the *fast_track* is the same and the path of the corrected sum can employ two clock cycle.
This result is significant because solve the problem caused by carry in of the adder.

4.2.3 Area

Area results for the three arithmetic unit in Tab. 4.10 and Tab. 4.11 are aligned with the expectations.

The solutions without *error_network* has always a smaller area with respect to the exact case (Fig. 4.10) and increases with k, the only exception is for binary_64 with k=32 where they are equal. If the part for the error handling is introduced the speculative solution increase its area and became the one which larger surface.

Arithmetic unit Area

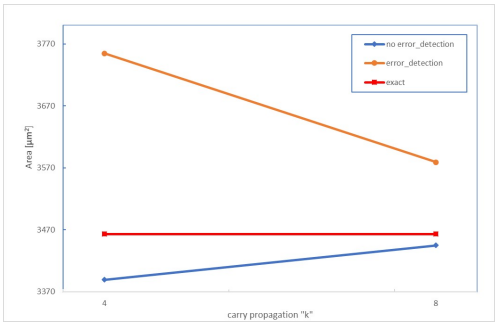
IEEE754	*Quantity*	*Exact*	*Speculative without error_network*				
			K=4	K=8	K=16	K=32	K=64
	# ports	943	913	924			
	# nets	1815	1754	1802			
	# cells	1045	997	1019			
binary_16	references	59	56	56			
	C-area [µm^2]	2154.6	2080.44	2136.64			
	NC-area [µm^2]	1308.96	1308.96	1308.96			
	T-area [µm^2]	3463.53	3389.4	3445.6			
	# ports	1967	1835	1895	1924		
	# nets	3833	3628	3717	3789		
	# cells	2125	2040	2077	2112		
binary_32	references	41	41	41	41		
	C-area [µm^2]	3577.32	3408.48	3479.4	3535.19		
	NC-area [µm^2]	4303.44	4305.24	4303.44	4303.44		
	T-area [µm^2]	7880.76	7713.72	7782.84	7838.63		
	# ports	5905	5461	5605	5737	5845	
	# nets	12757	12113	12403	12589	12756	
	# cells	7392	7142	7309	7381	7450	
binary_64	references	58	57	58	58	58	
	C-area [µm^2]	14013	13697.64	13861.8	13916	14008.32.32	
	NC-area [µm^2]	14902.56	14902.56	14902.56	14902.56	14902.56	
	T-area [µm^2]	28915.56	28600.2	28764.36	28818.56	28910.88	
	# ports	4399	3091	3415	3727	4015	4255
	# nets	26161	24202	24820	25097	25525	25788
	# cells	19464	18647	18993	19010	19186	19237
binary_128	references	187	182	186	198	186	187
	C-area [µm^2]	47991.6	4644.32	46790.28	47344.32	47437.28	47704.68

	NC-area [µm^2]	31736.16	31551.48	31778.64	3977.28	31651.92	31621.32
	T-area[µm^2]	79727.76	77995.8	78568.92	78913.08	79089.2	79326

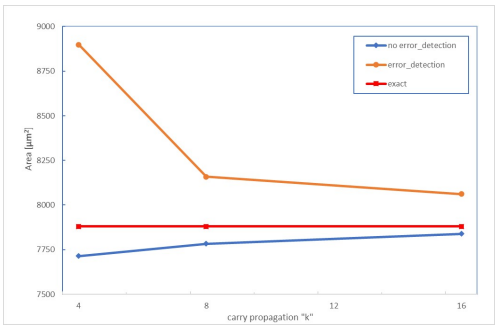
Table 4.10: Area occupancy of exact and speculative arithmetic unit without *error_ network*. C stands for "Combinational", NC for "Non-Combinational" and T for "Total".

Arithmetic unit Area							
IEEE754	*Quantity*	*Exact*	*Speculative with error_ network*				
			K=4	K=8	K=16	K=32	K=64
	# ports	943	997	997			
	# nets	1815	1903	1887			
	# cells	1045	1139	1085			
binary_16	references	59	62	62			
	C-area [µm^2]	2154.6	2426.3	2250.2			
	NC-area [µm^2]	1308.96	1328.2	1328.8			
	T-area [µm^2]	3463.53	3754.5	3579			
	# ports	1967	2143	2143	2143		
	# nets	3833	4387	4112	4098		
	# cells	2125	2534	2247	2239		
binary_32	references	41	47	46	46		
	C-area [µm^2]	3577.32	4541.04	3853.44	3762.34		
	NC-area [µm^2]	4303.44	4356.72	4305.6	4299.12		
	T-area [µm^2]	7880.76	8897.76	8159.04	8061.46		
	# ports	5905	6327	6327	6327	6327	
	# nets	12757	13329	13303	13287	13268	
	# cells	7392	7606	7578	7570	7567	
binary_64	references	58	70	69	69	69	
	C-area [µm^2]	14013	14518.08	14474.88	14456.48	14415.4	
	NC-area [µm^2]	14902.56	1499.76	15001.56	14999.76	1499.76	
	T-area [µm^2]	28915.56	29517.84	29476.44	29456.24	29415.16	
	# ports	4399	4857	4857	4857	4857	4857
	# nets	26161	27035	26890	26978	26966	26945
	# cells	19464	19997	19844	19910	19902	19893
binary_128	references	187	193	199	197	188	187
	C-area [µm^2]	47991.6	49454.28	49063.68	49081.68	49021.2	48940.56
	NC-area [µm^2]	31736.16	31784.76	31734.72	31685.88	31704.6	31726.92
	T-area[µm^2]	79727.76	81239.04	80798.4	80767.56	80725.8	80703.48

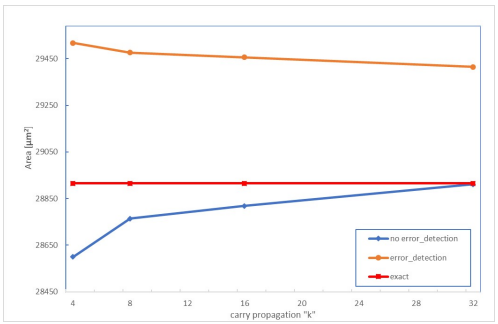
Table 4.11: Area occupancy of exact and speculative arithmetic unit with *error_ network*. C stands for "Combinational", NC for "Non-Combinational" and T for "Total".



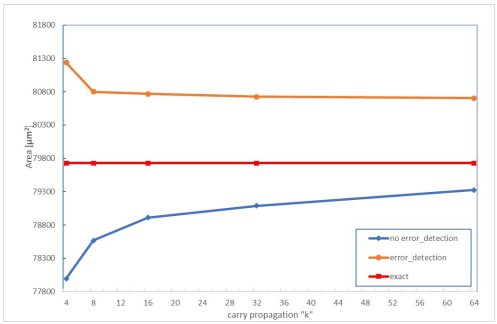
(a) Area binary_16



(b) Area binary_32



(c) Area for binary_64



(d) Area for binary_128

Figure 4.10: Confront between area of exact floating-point arithmetic unit and the speculative ones with and without *error_network*.

A final observation is that the area variation caused by the different adder choices does not have a big impact on the total area because of the huge overhead of the other components in the arithmetic unit, especially the adder.

4.2.4 Power

Arithmetic unit Power							
IEEE754	*Quantity*	*Exact*	*Speculative without error_network*				
			K=4	K=8	K=16	K=32	K=64
	CI-power [µW]	634.76	597.15	617.3			
binary_16	NS-power [µW]	47.52	42.3	45.6			

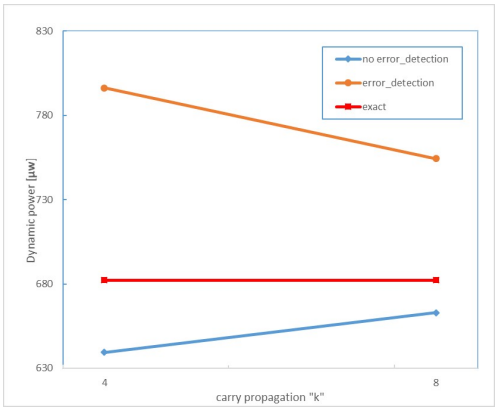
	TD-power [µW]	682.28	639.45	662.9			
	CL-power [nW]	235.4	214.2	227.6			
	CI-power [µW]	788.915	776.636	784.924			
binary_32	NS-power [µW]	54.1036	50.4171	51.5518	53.4616		
	TD-power [µW]	843.018	827.053	836.476	841.42		
	CL-power [nW]	539.667	519.153	528.718	535.751		
	CI-power	1229.2	1156	1187.8	1212.8		
binary_64	NS-power [µW]	55.1054	48.1446	54.412	55.156	56.1134	
	TD-power [µW]	1284.31	1204.14	1242.21	1267.96	1280.71	
	CL-power [nW]	2028	1919.6	2009.4	2019.5	2029.1	
	CI-power	1464.1	1435.3	1445.8	1457.6	1460	
binary_128	NS-power [µW]	75.4065	75.4031	74.8052	74.6878	75.4031	75.6894
	TD-power [µW]	1539.51	1510.11	1520.49	1533	1534.68	1535.29
	CL-power [nW]	7465.6	7277.8	7328.8	7341.9	7364.8	7449.9

Table 4.12: Power consumption of exact and speculative arithmetic unit without *error_network*. CI stands for "Cell Internal", NS for "Net Switching", TD for "Total Dynamic" and CL for "Cell Leakage".

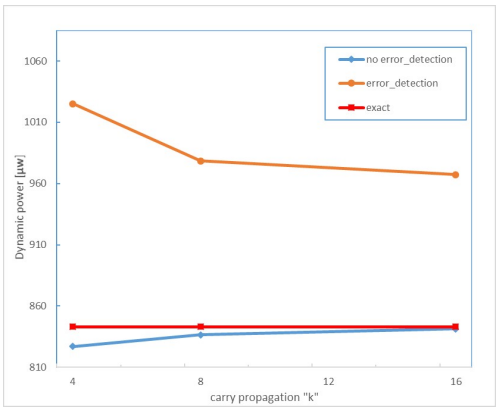
Arithmetic unit Power							
Width	*Quantity*	*Exact*	*Speculative with error_network*				
			K=4	K=8	K=16	K=32	K=64
	CI-power [µW]	634.76	703.8	679			
binary_16	NS-power [µW]	47.52	92.4	75.4			
	TD-power [µW]	682.28	796.2	754.4			
	CL-power [nW]	235.4	287.1	253.8			
	CI-power [µW]	788.915	896.228	873.279	864.653		
binary_32	NS-power [µW]	54.1036	128.941	105.164	102.587		
	TD-power [µW]	843.018	1025.17	978.443	967.24		
	CL-power [nW]	539.667	665.137	565.373	562.193		
	CI-power	1229.2					
binary_64	NS-power [µW]	55.1054	1649.3	1626	1613.1	1606.9	

	TD-power [µW]	1284.31	133.032	131.219	129.7	131.535	
	CL-power [nW]	2028	2080.6	2075.4	2069.7	2061.9	
	CI-power	1464.1					
binary_128	NS-power [µW]	75.4065	1562.4	1554.1	1553.9	1548	1546.5
	TD-power [µW]	1539.51	135.891	128.132	127.59	127.305	127.81
	CL-power [nW]	7465.6	7612	7574.4	7534.3	7531.4	7526.7

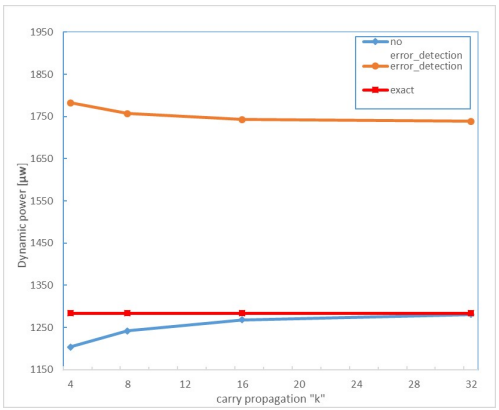
Table 4.13: Power consumption of exact and speculative arithmetic unit with *error_network*. CI stands for "Cell Internal", NS for "Net Switching", TD for "Total Dynamic" and CL for "Cell Leakage".



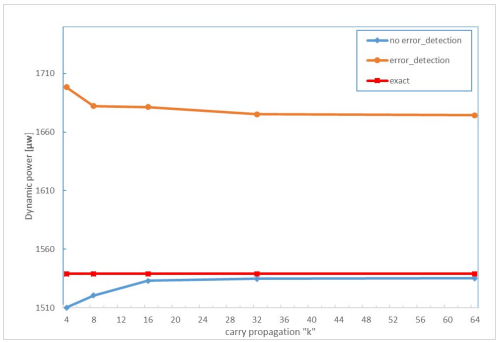
(a) Dynamic power binary_16



(b) Dynamic power binary_32



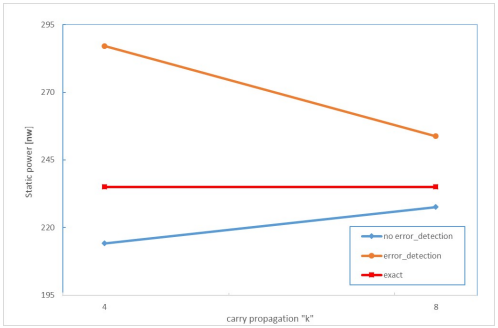
(c) Dynamic power for binary_64



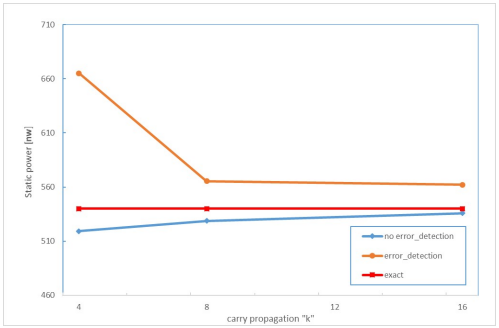
(d) Dynamic power for binary_128

Figure 4.11: Confront between dynamic power of exact floating-point arithmetic unit and the speculative ones with and without *error_network*.

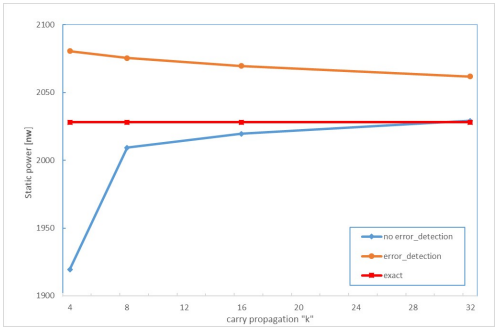
Both dynamic and power consumptions have the same behavior of the area for all the tested arithmetic units. As found before the changes caused by the various type of adders do not influence by much the total power because the overhead of the other components its really big, above all the pipe registers increase a lot the total switching activity (Fig.4.11)for binary_64 and binary_128.



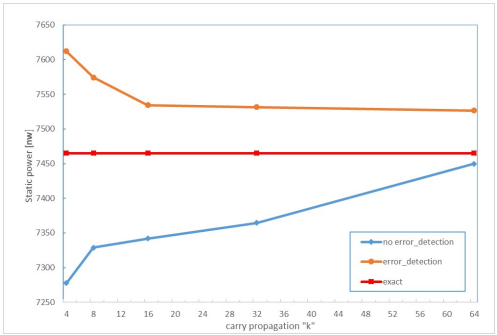
(a) Static power binary_16



(b) Static power binary_32



(c) Static power for binary_64

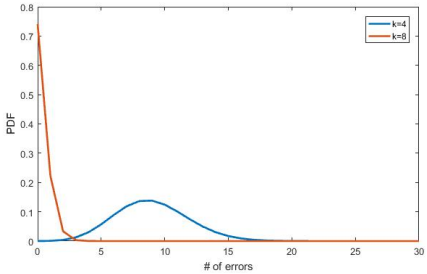


(d) Static power for binary_128

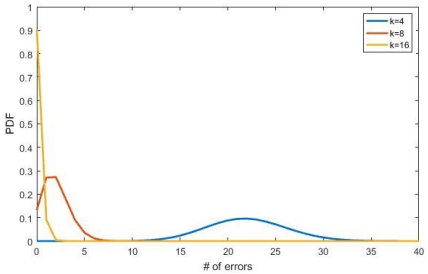
Figure 4.12: Confront between static power of exact floating-point arithmetic unit and the speculative ones with and without *error_network*.

4.3 Error evaluation

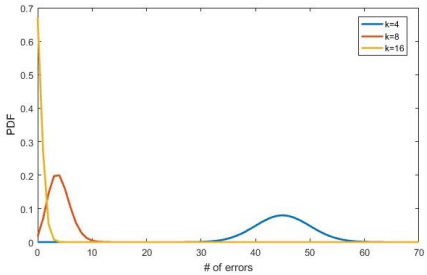
The last section of the Chapter is about the errors in the speculative architecture i.e. how many times the *fast_track* generates a wrong result and so, in the case of the arithmetic unit with the *error_network*, another clock cycle is required to provide the correct result. The analysis of the error is fundamental when the most appropriate solution must be chosen for a specific application. In Fig.4.13 are shown the diagram of the probability density function for each parallelism of the IEEE754 standard with the various levels of speculation.



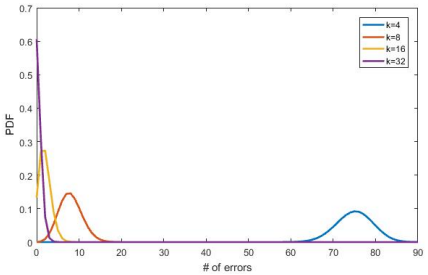
(a) Error rate binary_16



(b) Error rate binary_32



(c) Error rate binary_64



(d) Error rate binary_128

Figure 4.13: Probability density function (PDF) of the error occurrence for all the IEEE754 parallelisms with the various levels of speculation

Various conclusion can be drawn by those diagrams. First for low levels of speculation the occurrence of errors is smaller, this is quite obvious because

the architecture is closer to the exact one. Then if a confront between the various parallelism is done it can be noticed how for the same k the number of errors is bigger for higher parallelisms, also the range of the errors is wider. This means that the number is subjected to greater variation when multiple input pattern are applied. The origin of this result can be found in the longer length of the random generated input data that provides a bigger number of combinations.

By looking at the single architectures it is possible to find a validation of what has been decleared in 2.1, that is that the carry rarely propagate more than $\log_2(n)$ bit. For binary_16 the errors when k=8 is used is much smaller than the one with k=4 and it is nearly 0, same thing in binary_32 for k=16. Starting from binary_64 there were not errors for k=32 as well as k=64 in binary_128.
This does not mean that there is never an error when using this architecture. Although extremely rare there are some input patterns that can generate a wrong result, an example are the pattern used to test the correct behavior of the detection network during the simulation.

Chapter 5

Conclusions

Before drawing conclusions a small summary of the work done is provided. The final aim was to improve the performances of a arithmetic unit for floating point addition exploiting a speculative adder, so the first step was the study of which adder was better suited for this purpose. Floating point normally have an high parallelism of data and so the choice has fallen upon the type of adder better suited for this application, namely the Parallel Prefix adder. Then analyzing the performances of the possible Prefix network implementations the Han-Carlson topology was selected because it provided a better trade-off between delay, complexity and fan-out. To improve the basic adder a speculative approach was selected, the speculation made was that the internal carry did not propagate more than k-bit. Because some applications may need a result that is always correct the speculative adder must be integrated with a network that is able to detect when an error occurs and handle it, the solution was found in the Variable Latency Speculative adder. This architecture uses a *fast_ track* path that generates the speculative result and so reduces the delay, in parallel a *error_ detection* network evaluate if there are some errors in the result and if this is the case a signal is generated that makes the input remain constant for another clock cycle so that the *error_ correction* can compute the right result. The adder was then implemented in VHDL and characterized for all the IEEE754 formats and with

various level of speculation with and without the *error_network* to provide a complete study on how the performances vary.
Then the adder was implemented in the complete floating point architecture and another characterization was performed for the evaluation of the performances in the specific application. In order to have significant result a pipeline was performed to decrease the overhead of the normalizer component in the arithmetic unit. Lastly an evaluation of the error occurrence of the speculative architecture was performed because it plays a mayor role in the evaluation of the arithmetic unit performances.
The results obtained can be understood more easily if the ones of the adder are explained first. The speculative adder without the *error_network* provides always better results for what concerns delay, area and power with respect to the variable latency and exact implementation. The drawback is the possibility of a wrong result that can not be avoided, a marginal solution to this problem is using a smaller level of speculation because if the errors probability are considered, starting from a value of k=8 for almost all the parallelism, the number of errors is really small but always present.
The variable latency adder that includes the *error_network* always offers a correct result by employing an additional clock cycle, but if the errors are frequent the latency may grow too much. From the results of the characterization some mayor drawbacks of this architecture emerged. The first and most obvious are the area and power overheads caused by the addition of the *error_detection* and *error_correction* networks that needs additional components which bring a bigger static and dynamic power consumption. Under this aspect the performances improve if smaller speculation levels are used because the *error_detection* is implemented with less components. For what concerns the delay the result are not as good as the ones expected, because there is an improvement compared to the exact adder but only for high level of speculation, otherwise the delay is equal or worse than the standard adder. This behavior is caused by the *carry-in* of the adder because its presence require for the *error_detection* a value generated from the last stage of the

speculative computation and so the total delay does not benefit from the reduced delay of the detection part because it is compensated by the speculative part that for high value of k is almost similar to the exact adder.
The results from the characterization of the complete arithmetic unit are aligned with the ones described before for what concerns area and power but the interesting behavior it the one related to delay. For the architecture including the adder with the *error_network* the frequency for high levels of speculation is as expected, the things change for low level of speculation. In this case the frequency achievable is higher than the one of the exact architecture and it is close to the other speculative solution. This happens because for low value of k the critical path it is still the one through the detection network, when k grows the delay overhead of other components overtakes the *error_detection* and so the critical path its similar to the speculative architecture without *error_network*. The final results concerns the error occurrence, analyzing the density probability functions for all four standard of IEEE754 the errors became very rare starting from k=8 for binary_16 and binary_32 and k=16 for binary_64 and binary_128. This implies that this two level of speculation are the most suitable for almost all kind of application because a high improvement in all performances can be achieved with the speculation without correction and if the *error_network* is added its overhead of area and power is quite well balanced in terms of delay reduction and correctness.
In conclusion the results of this thesis work are quite satisfactory because they provide a wide analysis of the pro and cons of employing a speculative approach to improve the performances of a floating point adder that can be implemented not only in a GPU model but can be extended to all kind of application from FPGA to ASIC and it is also reusable as a starting point for future works.
Working on this thesis has proven itself quite challenging but also satisfactory because allowed me to exploit all the knowledge and skills obtained during this Master of Science in Electronic Engineering and to work on the design of digital systems, a topic that I always liked.

Bibliography

[1] Michael L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic. Society for Industrial and Applied Mathematics, Philadelphia*, 2001.

[2] *754-2008 - IEEE Standard for Floating-Point Arithmetic.* Aug 2008.

[3] Armando Aloisio and Stefano Ricci. *A framework for gpu design and simulation. Master's thesis, Politecnico di Torino*, December 2014.

[4] *AMD Accelerated Parallel Processing Technology. Reference Guide, 1.1a edition*, 2011.

[5] Stefano Capello. *A speculative approach for floating-point adders in GPU models. Master's thesis, Politecnico di Torino*, December 2016.

[6] Reto Zimmermann. *Binary Adder Architectures for Cell-Based VLSI and their Synthesis. Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, Diss. ETH NO. 12480*, 1997.

[7] Israel Koren. *Computer Arithmetic Algorithms.* 1993.

[8] Behrooz Parhami. *Computer Arithmetic: algorithms and hardware designs, Second Edition. Oxford University Press*, 2010.

[9] R. E. Ladner and M. J. Fischer. *Parallel prefix computation. J. ACM, vol. 27, no. 4*, October 1980.

[10] R. P. Brent and H. T. Kung. *A regular layout for parallel adders. IEEE Trans. Comput., vol. C-31, no. 3*, March 1982.

[11] P. M. Kogge and H. S. Stone. *A parallel algorithm for the efficient solution of a general class of recurrence. IEEE Trans. Comput., vol. C-22, no. 8*, August 1973.

[12] T. Han and D. A. Carlson. *Fast area-efficient VLSI adders. Proc. IEEE 8th Symp. Comput. Arith. (ARITH)*, May 1987.

[13] Swapna K. Gedam and Pravin P.Zode. *Parallel Prefix Han-Carlson Adder. International Journal of Research in Engineering and Applied Sciences, Vol. 02, Issue 02*, July 2014.

[14] Esposito Darjn, De Caro Davide, Napoli Ettore, Petra Nicola, and Strollo Antonio Giuseppe Maria. *Variable Latency Speculative Han-Carlson Adder. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS I: REGULAR PAPERS, VOL. 62, NO. 5*, May 2015.

[15] Liu Weiqiang, Wang Chenghua, O'Neill Máire, Lombardi Fabrizio, and Chen Linbin. *Design and Analysis of Inexact Floating-Point Adders. IEEE TRANSACTIONS ON COMPUTERS VOL. 65 NO. 1*, Jan. 2016.

[16] S. M. Nowick. *Design of a low-latency asynchronous adder using speculative completion. IEE Proc. Comput. Digit. Tech., vol. 143, no. 5*, September 1996.

[17] Verma Ajay K., Brisk Philip, and Ienne Paolo. *Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design. DESIGN, AUTOMATION AND TEST IN EUROPE*, Mar. 2008.