Methods to Evaluate Accuracy-Energy Trade-Off in Operator-Level Approximate Computing

Benjamin Barrois

▶ **To cite this version:**

Benjamin Barrois. Methods to Evaluate Accuracy-Energy Trade-Off in Operator-Level Approximate Computing. Embedded Systems. Université de Rennes 1, 2017. English. NNT : . tel-01665015v1

HAL Id: tel-01665015
https://inria.hal.science/tel-01665015v1

Submitted on 15 Dec 2017 (v1), last revised 29 Mar 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N^{o} d'ordre : 00000 ANNÉE 2017

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Traitement du Signal et Télécommunications

École doctorale MathSTIC

présentée par

Benjamin BARROIS

préparée à l'unité de recherche IRISA – Equipe CAIRN – UMR 6074
Institut de Recherche en Informatique et Système Aléatoires
Université de Rennes 1 - ENSSAT Lannion

Methods to Evaluate Accuracy-Energy Trade-Off in Operator-Level Approximate Computing

Thèse soutenue à Rennes
le 11 Décembre 2017

devant le jury composé de :

To BE DEFINED
/ *Président*

Marc DURANTON
Expert International au CEA LIST, Saclay / *Rapporteur*

Alberto BOSIO
Maître de Conférences HDR à l'Université de Montpellier, LIRMM / *Rapporteur*

Daniel MENARD
Professeur à l'INSA Rennes / *Examinateur*

Arnaud TISSERAND
Directeur de Recherche CNRS, Lab-STICC / *Examinateur*

Anca MOLNOS
Chercheur au CEA LETI, Grenoble / *Examinatrice*

Olivier SENTIEYS
Professeur à l'Université de Rennes 1 / *Directeur de thèse*

Remerciements

Mes premiers remerciements vont à mon directeur de thèse Olivier SENTIEYS, qui a cru en moi dès le master et m'a permis d'effectuer cette thèse. C'est appuyé par son expérience que j'ai pu produire les différents travaux présentés dans ce document et m'intégrer dans les riches communautés propres aux domaines de ces travaux.

Un grand merci à tous ceux qui ont collaboré aux différentes publications de cette thèse. Un grand merci tout d'abord à Rengarajan RAGAVAN pour avoir accueilli mes contributions à ses travaux, ainsi qu'à tous ceux ayant contribué aux miens : Olivier SENTIEYS tout d'abord, Karthick PARASHAR, Daniel MENARD, et Cédric KILLIAN.

Merci aux rapporteurs Marc DURANTON et Alberto BOSIO pour la relecture de ce document. Merci également à tous les membres du jury, Daniel MENARD et Anca MOLNOS, avec lesquels j'ai eu le privilège de riches échanges scientifiques et humains au travers de l'ANR Artefact, Arnaud TISSERAND, avec qui je regrette de ne pas avoir pu plus travailler pour toutes les connaissances d'arithmétique matérielles qu'il aurait pu m'apporter, et encore une fois Olivier SENTIEYS.

Un grand merci aux financements de l'Université de Rennes 1, grâce auxquels j'ai pu vivre décemment pendant ces 1187 jours de travail intense.

Un immense merci à l'équipe CAIRN pour son accueil et ce cadre fantastique de travail, toujours dans la bonne humeur, que ce soit à Lannion ou à Rennes. Des remerciements tous particuliers (dans le désordre) à Nicolas S., Pierre, Mickaël, Hamza, Baptiste, Christophe, Gabriel, Karim, Audrey, Nicolas R. et Joel, pour les bons moments à la fois dans les murs de CAIRN et à l'extérieur. Un grand merci à Angélique et Nadia pour leur grande tolérance dans les deadlines, très appréciables quand on est *très légèrement* tête-en-l'air.

Un tendre merci à mes parents de m'avoir toujours soutenu et d'avoir toujours été présents malgré la distance. Une mention spéciale pour ma mère, qui m'a donné le courage de me battre contre moi-même tout au long de cette thèse en vaincant cancer sur cancer.

Le plus grand des remerciements va à celle qui partage ma vie et m'a soutenu depuis le premier jour de cette entreprise, dans les bons et mauvais moments, et sans l'amour de qui rien de tout cela n'aurait été possible. Tendre merci Juliana, et merci d'avance pour ta présence dans les nombreuses nouvelles aventures qui suivront et qui ne seraient rien sans toi.

Abstract

During the past decades, significant improvements have been made in computational performance and energy efficiency, following Moore's law. However, the physical limits in silicon transistor size are forecast to be reached soon, and finding ways to improve the energy-performance ratio has become a major stake in research and industry. One of the ways to increase energy efficiency is the modification of number representations and data size used for computations. Today, double precision floating-point computation is a standard. However, it is now admitted that an important amount of applications could be computed using less accurate representations with no or very small impact on the application output quality. This paradigm, recently termed as Approximate Computing, has emerged as a promising approach and has become a major field of research to improve both speed and energy consumption in embedded and high-performance computing systems. Approximate computing relies on the ability of many systems and applications to tolerate some loss of quality or optimality in the computed result. By relaxing the need for fully precise or completely deterministic operations, approximate computing techniques allow substantially improved energy efficiency.

In this thesis, the error-performance trade-off relaxing the accuracy in basic arithmetic operators is addressed. After a study and a constructive criticism of the existing ways to perform approximate arithmetic operations, methods and tools to evaluate the cost in terms of error and the impact in terms of performance using different arithmetic paradigms are presented. First, after a quick description of classical floating-point and fixed-point arithmetic, a literature study of approximate operators is presented. Different techniques to create approximate adders and multipliers are highlighted by this study, as well as the problem of very variable nature and amplitude of the errors they induce in computations. Second, a system-level scalable technique to estimate fixed-point error leveraging Power Spectral Density (PSD) is presented. This technique considers the spectral nature of the quantization noise filtered across the system, leading to much higher accuracy in error estimation than PSD-agnostic methods, and lower complexity than classical propagation of error mean and variance across the whole system. Then, the problem of analytical estimation of the error of approximate operators is addressed. The variety in their behavior and logic structure makes the existing techniques either with high complexity, leading to high memory or computational cost, or with low estimation accuracy. With the Bitwise-Error Rate (BWER) propagation, we find a good compromise between estimation complexity and accuracy. Then, a pseudo-simulation technique based on approximate operators for Voltage Over-Scaling (VOS) is presented. This technique allows for VOS errors to be estimated using high-level simulation instead of extremely long and memory-costly SPICE transistor-level simulations.

Another contribution is a comparative study between fixed-point and approximate operators. To achieve this, an error and performance open-source estimation framework, called APXPERF, was developed. Embedding VHDL and C representations of several approximate operators, APXPERF automatically estimates speed, area and power at gate-level of these operators, as well as their accuracy with several error metrics. The framework can also easily evaluate the error involved by approximate operators on complex applications, and thus provide application-related metrics. In this study, we showed that, even if approximate operators can have more interesting stand-alone performance than fixed-point operators, fixed-point

arithmetic provides much higher performance and important energy savings when applied to real-life applications, thanks to much smaller error and data width. The last contribution is a comparative study between fixed-point and floating-point paradigms. For this, a new version of APXPERF was developed, which adds an extra-layer of High Level Synthesis (HLS) achieved by Mentor Graphics Catapult C or Xilinx Vivado HLS. APXPERF v2 uses a unique C++ source for both hardware performance and accuracy estimations of approximate operators. The framework comes with template-based synthesizable C++ libraries for integer approximate operators (APX_FIXED) and for custom floating-point operators (CT_FLOAT).

The second version of the framework can also evaluate complex applications, which are now synthesizable using HLS. After a comparative evaluation of our custom floating-point library with other existing libraries, fixed-point and floating-point paradigms are first compared in terms of stand-alone performance and accuracy. Then, they are compared in the context of K-Means clustering and FFT applications, where the interest of small-width floating-point is highlighted.

The thesis concludes in the strong interest of reducing the bit-width of arithmetic operations, but also in the important issues brought by approximation. First, the many integer approximate operators published with promising important energy savings at low error cost, seem not to keep their promises when considered at application level. Indeed, we showed that fixed-point arithmetic with smaller bit-width should be preferred to inexact operators. Finally, we emphasize the interest of small-width floating-point for approximate computing. Small floating-point is demonstrated to be very interesting in low-energy systems, compensating its overhead with its high dynamic range, its high flexibility and its ease of use for developers and system designers.

Résumé en Français

Au cours de ces dernières décennies, des améliorations significatives ont été faites en termes de performances de calcul et de réduction d'énergie, suivant la loi de Moore. Cependant, les limites physiques liées à la réduction de la taille des transistors à base de silicium sont en passe d'être atteintes et le solutionnement de ce problème est aujourd'hui l'un des enjeux majeurs de la recherche et de l'industrie. L'un des moyens d'améliorer l'efficacité énergétique est l'utilisation de différentes représentations des nombres, et l'utilisation de tailles réduites pour ces représentations. Le standard de représentation des nombres réels est aujourd'hui la virgule flottante double-précision. Cependant, il est maintenant admis qu'un important nombre d'applications pourrait être exécuté en utilisant des représentations de précision inférieure, avec un impact minime sur la qualité de leurs sorties. Ce paradigme, récemment qualifié de *calcul approximatif* ou *approximate computing* en anglais, apparaît comme une approche prometteuse et est devenu l'un des secteurs de recherche majeurs visant à l'amélioration de la vitesse de calcul et de la consommation énergétique pour les systèmes de calcul embarqués et de haute performance. Le calcul approximatif s'appuie sur la tolérance de beaucoup de systèmes et applications à la perte de qualité ou d'optimalité dans le résultat produit. En relâchant les besoins d'extrême précision de résultat ou de leur déterminisme, les techniques de calcul approximatif permettent une efficacité énergétique considérablement accrue.

Dans cette thèse, le compromis performance-erreur obtenu en relâchant la précision des calculs dans les opérateurs arithmétiques de base est traité. Après l'étude et une critique constructive des moyens existants d'effectuer de manière approximée les opérations arithmétiques de base, des méthodes et outils pour l'évaluation du coût en termes d'erreur et l'impact en termes de performance moyennant l'utilisation de différents paradigmes arithmétiques sont présentés. Tout d'abord, après une rapide description des arithmétiques classiques que sont la virgule flottante et la virgule fixe, une étude de la littérature des opérateurs approximatifs est présentée. Les principales techniques de création d'additionneurs et multiplieurs approximatifs sont soulignées par cette étude, ainsi que le problème de la nature et de l'amplitude très variable des erreurs induites lors des calculs les utilisant. Dans un second temps, une technique modulaire d'estimation de l'erreur virgule fixe s'appuyant sur la densité spectrale de puissance est présentée. Cette technique considère la nature spectrale du bruit de quantification filtré à travers le système, menant à une précision accrue de l'estimation d'erreur comparé aux méthodes modulaires ne prenant pas en compte cette nature spectrale, et d'une complexité plus basse que la propagation classique des moyenne et variance d'erreur à travers le système complet. Ensuite, le problème de l'estimation analytique de l'erreur produite par les opérateurs approximatifs est soulevé. La grande variété comportementale et structurelle des opérateurs approximatifs rend les techniques existante beaucoup plus complexes, ce qui résulte en un fort coût en termes de mémoire ou de puissance de calcul, ou au contraire en une mauvaise qualité d'estimation. Avec la technique proposée de propagation du taux d'erreur binaire positionnel, un bon compromis est trouvé entre la complexité de l'estimation et sa précision. Ensuite, une technique utilisant la pseudo simulation à base d'opérateurs arithmétiques approximatifs pour la reproduction des effets de la VOS est présentée. Cette technique permet d'utiliser des simulations haut niveau pour estimer les erreurs liées à la VOS en lieu et place de simulations SPICE niveau transistor, extrêmement longues et coûteuses en mémoire.

Une étude comparative entre virgule fixe et operateurs approximatifs constitue une contribution supplémentaire. Pour cette étude, un outil libre d'estimation d'erreur et de performance a été développé, ApxPerf. Embarquant des représentations en C et en VHDL d'un certain nombre d'opérateurs approximatifs, ApxPerf estime de manière automatisée la vitesse, la surface et la puissance, simulée au niveau portes logiques, ainsi que leur précision en se basant sur plusieurs métriques complémentaires. ApxPerf permet également d'évaluer de manière simple l'erreur induite par l'utilisation d'opérateurs approximatifs sur des applications complexes, en fournissant des résultats basés sur des métriques pertinentes dans le contexte de l'application. Cette étude montre que, bien que les opérateurs arithmétiques soient capables de meilleures performances comparés aux opérateurs virgule fixe lorsqu'ils sont comparés de manière atomique, l'arithmétique virgule fixe produit de bien meilleurs résultats en terme d'erreur et d'importants gains énergétiques dans un contexte applicatif, grâce notamment à une erreur équivalente obtenue avec des tailles de données bien inférieures.

L'ultime contribution de cette thèse est l'étude comparative entre les paradigmes virgule fixe et virgule flottante. Pour cela, une nouvelle version d'ApxPerf a été développée, ajoutant une couche supplémentaire de synthèse haut niveau (HLS), assurée par Catapult C de Mentor Graphics ou Vivado HLS de Xilinx. La seconde version d'ApxPerf ne prend plus en entrée qu'une unique source C++ servant à la fois pour l'estimation des performances matérielles et en termes d'erreur des opérateurs approximatifs. Cette nouvelle version embarque une bibliothèque synthétisable d'opérateurs approximatifs basée sur des templates C++, apx_fixed, ainsi qu'une bibliothèque synthétisable d'opérateurs virgule flottante simplifiés, ct_float. Des applications complexes sont également incluses, synthétisables par HLS. Après une évaluation comparative de ct_float et d'autres bibliothèques existantes, les arithmétiques virgule fixe et virgule flottante sont tout d'abord comparées en terme de performance matérielle brute, puis en terme de performance et d'erreur dans le contexte de la classification K-Means et de la transformée de Fourier rapide (FFT), où les intérêts et inconvénients des nombre flottants de petite taille sont soulignés.

La thèse conclut sur le fort intérêt lié à la réduction de la taille des opérations arithmétiques, mais aussi des problèmes apportés par cette approximation. En premier lieu, les nombreux opérateurs entiers publiés promettant d'importants gains en énergie avec un faible coût d'erreur ne semblent pas tenir leurs promesses lorsqu'ils sont considérés dans un contexte applicatif. En effet, il a été montré dans cette thèse que l'utilisation de l'arithmétique virgule fixe avec des tailles de données réduites donnait de meilleurs résultats. L'arithmétique flottante a quant à elle démontré pouvoir être intéressante même dans les systèmes à faible énergie, compensant son surcoût par sa forte dynamique, sa grande flexibilité et sa facilité d'utilisation pour les développeurs et les concepteurs de systèmes.

Contents

Contents **5**

Introduction **9**

1 Trading Accuracy for Performance in Computing Systems **13**

1.1 Various Methods to Trade Accuracy for Performance 13
1.1.1 Voltage Overscaling 13
1.1.2 Algorithmic Approximations 14
1.1.3 Approximate Basic Arithmetic 15
1.2 Relaxing Accuracy Using Floating-Point Arithmetic 15
1.2.1 Floating-Point Representation for Real Numbers 15
1.2.2 Floating-Point Addition/Subtraction and Multiplication 17
1.2.3 Potential for Relaxing Accuracy in Floating-Point Arithmetic 20
1.3 Relaxing Accuracy Using Fixed-Point Arithmetic 21
1.3.1 Fixed-Point Representation for Real Numbers 21
1.3.2 Quantization and Rounding 22
1.3.3 Addition and Subtraction in Fixed-Point Representation 27
1.3.4 Multiplication in Fixed-Point Representation 31
1.4 Relaxing Accuracy Using Approximate Operators 36
1.4.1 Approximate Integer Addition 36
1.4.1.1 Sample Adder, Almost Correct Adder and Variable Latency Speculative Adder 38
1.4.1.2 Error-Tolerant Adders 43
1.4.1.3 Accuracy-Configurable Approximate Adder 50
1.4.1.4 Gracefully-Degrading Adder 55
1.4.1.5 Addition Using Approximate Full-Adder Logic 61
1.4.1.6 Approximate Adders by Probabilistic Pruning of Existing Designs 65
1.4.2 Approximate Integer Multiplication 67
1.4.2.1 Approximate Array Multipliers 68
1.4.2.2 Error-Tolerant Multiplier 74
1.4.2.3 Approximate Multipliers using Modified Booth Encoding . . 77
1.4.2.4 Dynamic Range Unbiased Multiplier 87

1.4.3 Final Discussion on Approximate Operators in Literature 90

2 Leveraging Power Spectral Density in Fixed-Point System Refinement 93
2.1 Motivation for Using Fixed-Point Arithmetic in Low-Power Computing 93
2.2 Related work on accuracy analysis 94
2.3 PSD-based accuracy evaluation 98
2.3.1 PSD of a quantization noise 98
2.3.2 PSD propagation across a fixed-point Linear and Time-Invariant (LTI) system 99
2.4 Experimental Results of Proposed Power Spectral Density (PSD) Propagation Method 100
2.4.1 Experimental Setup 101
2.4.1.1 Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) Filters 101
2.4.1.2 Frequency Domain Filtering 101
2.4.1.3 *Daubechies* 9/7 Discrete Wavelet Transform 101
2.4.2 Validation of the Approach for LTI Systems 103
2.4.3 Influence of the Number of PSD Samples 103
2.4.4 Comparison with PSD-Agnostic Methods 104
2.4.5 Frequency Repartition of Output Error 105
2.5 Conclusions about PSD Estimation Method 106

3 Fast Approximate Arithmetic Operator Error Modeling 107
3.1 The Problem of Analytical Methods for Approximate Arithmetic 107
3.2 Bitwise-Error Rate Propagation Method 109
3.2.1 Main Principle of Bitwise-Error Rate (BWER) Propagation Method . . 109
3.2.2 Storage Optimization and Training of the BWER Propagation Data Structure 110
3.2.3 BWER Propagation Algorithm 112
3.3 Results of the BWER Method on Approximate Adders and Multipliers 113
3.3.1 BWER Training Convergence Speed 114
3.3.2 Evaluation of the Accuracy of BWER Propagation Method 114
3.3.3 Estimation and Simulation Time 119
3.3.4 Conclusion and Perspectives 120
3.4 Modeling the Effects of Voltage Over-Scaling (Voltage OverScaling (VOS)) in Arithmetic Operators 122
3.4.1 Characterization of Arithmetic Operators 123
3.4.2 Modelling of VOS Arithmetic Operators 124
3.4.3 Conclusion 130

4 Approximate Operators Versus Careful Data Sizing 133
4.1 APXPERF: Hardware Performance and Accuracy Characterization Framework for Approximate Computing 133

4.1.1 APXPERF– First Version 134
4.1.2 APXPERF– Second Version 136
4.2 Raw Comparison of Fixed-Point and Approximate Operators 140
4.3 Comparison of Fixed-Point and Approximate Operators on Signal Processing Applications 144
4.3.1 Fast Fourier Transform 144
4.3.2 JPEG Encoding 144
4.3.3 Motion Compensation Filter for HEVC Decoder 146
4.3.4 K-means Clustering 148
4.4 Considerations About Arithmetic Operator-Level Approximate Computing . . 149

5 Fixed-Point Versus Custom Floating-Point Representation in Low-Energy Computing 151
5.1 CT__FLOAT: a Custom Synthesizable Floating-Point Library 151
5.1.1 The CT__FLOAT Library 152
5.1.2 Performance of CT__FLOAT Compared to Other Custom Floating-Point Libraries 155
5.2 Stand-Alone Comparison of Fixed-Point and Custom Floating-Point 161
5.3 Application-Based Comparison of Fixed-Point and Custom Floating-Point . . . 163
5.3.1 Comparison on K-Means Clustering Application 164
5.3.1.1 K-Means Clustering Principle, Algorithm and Experimental Setup 164
5.3.1.2 Experimental Results on K-Means Clustering 167
5.3.2 Comparative Results on Fast Fourier Transform Application 168
5.4 FxP vs FlP – Conclusion and Discussion 171

Acronyms 181

Publications 183

Bibliography 190

List of Figures 191

List of Tables 195

Introduction

Handling The End Of Moore's Law

In the History of computers, huge improvements in calculation accuracy have been made in two ways. First, the accuracy of computations was gradually improved with the increase of the bit width allocated to number representation. This was made possible thanks to technology evolution and miniaturization, allowing a dramatical growth of the number of basic logic elements embedded in circuits. Second, accurate number representations such as floating-point were proposed. Floating-point representation was first used in 1914 in an electro-mechanical version of Charles Babbage's computing machine made by Leonardo Torres y Quevedo, the Analytical Engine, followed by Konrad Zuse's first programmable mechanical computer embedding 24-bit floating-point representation. Today, silicon-based general-purpose processors mostly embed 64-bit floating-point computation units, which is nowadays standard in terms of high-accuracy computing.

At the same time, important improvements of computational performance have been performed. The miniaturization, besides allowing larger bit-width, also allowed considerable benefits in terms of power and speed, and thus energy. CDC 6600 supercomputer created in 1964, with its 64K 60-bit words of memory, had a power consumption of approximatively 150 kW for a computing capacity of approximatively 500 KFLOPS, which makes 3.3 floating-point operations per Watt. In comparison, best 2017's world' supercomputer Sunway TaihuLight in China with its 1.31 Pbytes of memory, announces 6.05 GFLOPS/W [5]. Therefore, in forty years, the supercomputers energy efficiency has improved by roughly 2E9, following the needs of industry. These impressive progresses were achieved according to Moore's law, who forecast in 1965 [6] an exponential growth of computation circuit complexity, depicted in Figure 1 in terms of transistor count, performance, frequency, power and number of processing cores. From the day it was stated, this law and its numerous variations have outstandingly described the evolution of computing, but also the needs of the global market, such as computing is now inherent to nearly all possible domains, from weather forecasting to secured money transactions, including targeted advertising and self-driving cars.

However, many specialists agree on Moore's Law to end in a very near future [7]. First of all, the gradual decrease of the size of silicon transistors is coming to an end. With a Van Der Waals radium of 210 pm, today's 10 nm transistors are less than 50 atoms large, leading new issues at quantum physics scale, as well as increased current leakage. Then, the 20th century has known a steady increase of clock frequency in synchronous circuit, which represent the overwhelming majority of circuits, helping the important gain in performance. However, the

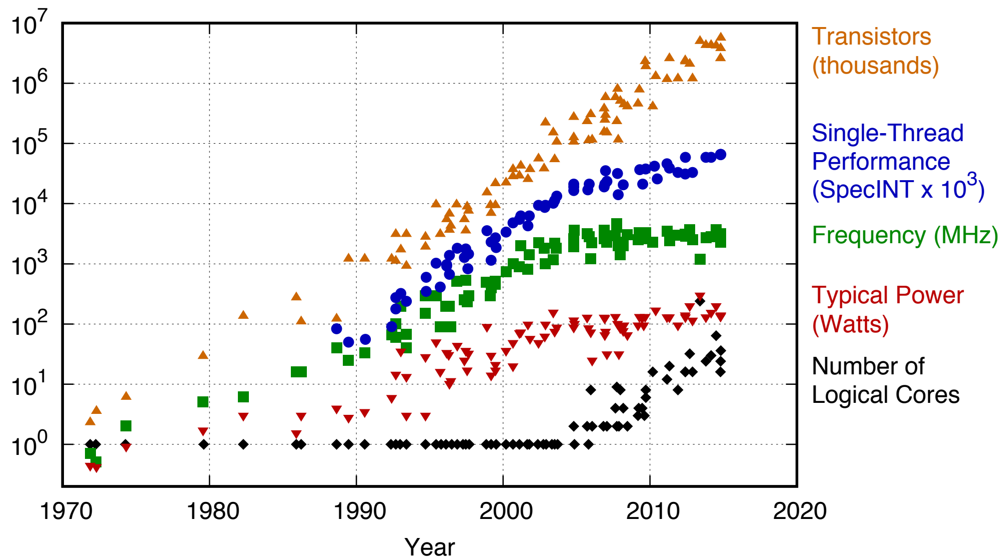


Figure 1 – 45 years of microprocessor trend data, collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten and completed by K. Rupp

beginning of the third millennium has seen a stagnation of this frequency, enforcing performance to be found in parallelism instead, single-core disappearing in favor of multi/many-core processors.

Technology being pushed to its limits and a new cutting-edge physical-layer technology not appearing to arrive soon, new ways have to be found for computing to follow the needs. Moreover, with a strong interest of industry into energy-critical embedded systems, energy-efficiency is sought more than ever. Specialists are anticipating a future technological revolution brought by the Internet Of Things (IOT), with a fast growth of the number of interconnected autonomous embedded systems [8, 9, 10]. As said above, a first performance improvement can be found in computing parallelism, thanks to multi-core/multi-thread superscalar or VLIW processors [11], or GPU computing. However, not all applications can be well parallelized because of data dependencies, leading to moderate speed-up in spite of an important area and energy overhead. A second modern way to improve performance is the use of hardware accelerators. These accelerators come in mainstream processors, with hardware video encoders/decoders such as for x264 or x265 codecs, but also in Field-Programmable Gate Arrays (FPGAs). FPGAs consist in a grid of programmable logic gates, allowing reconfigurable hardware implementation. In addition of general-purpose Look-Up Tables (LUTs), most FPGAs embed Digital Signal Processing blockss (DSPs) to accelerate signal processing computations. Hybrids embedding processors connected to an FPGA, embodied by Xilinx Zynq family, are today a big stake for high-performance embedded systems. Nevertheless, hardware and software co-design still represents an important cost in terms of development and testing.

Approximate Computing or Playing with Accuracy for Energy Efficiency

This thesis focuses on an alternative way to improve performance/energy ratio, which is relaxing computation accuracy to improve performance, as well in terms of energy/area for low-power systems, but also in terms of speed for High Performance Computing (HPC). Indeed, for many reasons, many applications can tolerate some imprecision for various reasons. For instance, having a high accuracy in signal processing applications can be useless if the input signals are noisy, since the least significant bits in the computation will be only applied on the noisy part of the signal. Also, some applications such as classification do not always have golden output and can tolerate a set of satisfying results. Therefore, it is useless to perform many loop iterations to get an output which can be considered as good enough.

Various methods applied at several levels are possible to relax the accuracy to get performance or energy benefits. At physical layer, voltage and frequency can be scaled beyond the circuit tolerance threshold with potential important energy of speed benefits, but with effects on the quality of the output that may be destructive and hard to be managed [12, 13]. At algorithmic level, several simplifications can be performed such as loop perforation or mathematical functions approximations. For instance, trigonometric functions can be approximated using COordinate Rotation DIgital Computing (CORDIC) [14] with limited number of iterations, and complex functions can be approximated by interval using simple polynomials stored in small tables. The choice and management of arithmetic paradigms also allows important energy savings while relaxing accuracy.

In this thesis, three main paradigms are explored:

- customizing floating-point operators,
- reducing bit-width and complexity using fixed-point arithmetic, which can be associated to quantization theory,
- and approximate integer operators, which perform arithmetic operations using inaccurate functions.

Floating-point arithmetic is often associated to high-precision computing with important time and energy overhead compared to fixed-point. Indeed, floating-point is today the most used representation of real numbers because of its high dynamic and high precision at any amplitude scale. However, because of more complex operations than for integer arithmetic and the complexity of the many particular cases handled in IEEE 754 standard [15], fixed-point is nearly always selected when low energy per operation is aimed at. In this thesis, we consider simplified small-bitwidth floating-point arithmetic implementations leading to better energy efficiency.

Fixed-point arithmetic is the most classical paradigm when it comes to low-energy computing. In this thesis, it is used as a reference for comparisons with the other paradigms. A model for fixed-point error estimation leveraging Power Spectral Density (PSD) is also proposed.

Finally, approximate arithmetic operators using modified addition and multiplication functions are considered. Many implementations of these operators were published this past decade, but they have never been the object of a complete comparative study.

After presenting literature about floating-point and fixed-point, a study of state-of-the-art approximate operators is proposed. Then, models for error propagation for fixed-point and approximate operators are described and evaluated. Finally, comparative studies between approximate operators and fixed-point on one side, and fixed-point and floating-point on the other side, leveraging classical signal processing applications. More details on the organization of this document are given in the next section.

Thesis Organization

Chapter 1 Approximate computing in general is presented, followed by a deeper study on the different existing computing arithmetic. After a presentation of classical floating-point and fixed-point arithmetic paradigms, state-of-the-art integer approximate adders and multipliers are presented to give an overview of the many existing techniques to lower energy introducing inaccuracy in computations.

Chapter 2 A novel technique to estimate the impact of quantization across large fixed-point systems is presented, leveraging the noise Power Spectral Density (PSD). The benefits of the method compared to others is then demonstrated on signal processing applications.

Chapter 3 A novel technique to estimate the error of integer approximate operators error propagated across a system is presented in thus chapter. This technique, based on Bitwise Error-Rate (BWER), first uses training by simulation to build a model, which is then used for fast propagation. This analytical technique is fast and requires less space in memory than other similar existing techniques. Then, a model for the reproduction of Voltage Over-Scaling (VOS) error in exact integer arithmetic operators using pseudo-simulation on approximate operators is presented.

Chapter 4 A comparative study of fixed-point and approximate arithmetic is presented in Chapter 4. Both paradigms are first compared in their stand-alone version, and then on several signal processing applications using relevant metrics. The study is performed using the first version of our open-source operator characterization framework APXPERF, and our approximate operator library APX_FIXED.

Chapter 5 Using a second version of our framework APXPERF embedding a High Level Synthesis (HLS) frontend and our custom floating-point library CT_FLOAT, a comparative study of fixed-point and small-width custom floating-point arithmetic is performed. First, the hardware performance and accuracy of both operators are compared in their stand-alone version. Then, this comparison is achieved in K-means clustering and Fast Fourier Transform (FFT) applications.

Chapter 1

Trading Accuracy for Performance in Computing Systems

In this Chapter, various methods to trade accuracy for performance are first listed in Section 1.1. Then, the study is centered on the different existing representations of numbers and the architectures of the arithmetic operators using them. In Section 1.2, floating-point arithmetic is developed. Then, Section 1.3 presents fixed-point arithmetic. Finally, a study of state-of-the-art approximate architectures of integer adders and multipliers is presented in Section 1.4.

1.1 Various Methods to Trade Accuracy for Performance

In this section, the main methods to trade accuracy for performance are presented. First, VOS is discussed. Then, existing algorithm-level transformations are presented. Finally, approximate arithmetic is introduced, to be further developed in Sections 1.2, 1.3, and 1.4 as the central element of this thesis.

1.1.1 Voltage Overscaling

The power consumption of a transistor in a synchronous circuit is linear with the frequency and proportional to the square of the voltage applied. For a same load of computations, decreasing frequency also increases computing time and the energy is the same. Thus, it is important to mostly exploit the voltage to save as much energy as possible. Nevertheless, decreasing voltage implies more instability in the transitions of the transistor, and this is why in a large majority of systems, the voltage is set above a certain threshold which ensures the stability of the system. Lowering the voltage under this threshold can cause the output of the transistor to be stuck to 0 or 1, compromising the integrity of the realized function.

One of the main issues of VOS is process variability. Indeed, two instances A and B of a same silicon-based chip are not able to handle the exact same voltage before breakdown, a given transistor in A being possibly weaker than the same in B, mostly because of Random Dopant Fluctuations (RDF) which are a major issue in nanometer-scale Very Large Scale Integration (VLSI). However, with the important possible energy gains brought by VOS, its mastering is

an important stake which is widely explored [12, 13]. Low-leakage technologies like Fully Depleted Silicon On Insulator (FDSOI) allow RDF to impact much less near-threshold computing variability. Despite technology improvements, near-threshold and sub-threshold computing needs error-correcting circuits, coming with an area, energy and delay overhead which needs to be inferior to the savings. In [16], a method called Razor is proposed to monitor at low cost circuit error rate to tune its voltage to get an acceptable failure rate. Therefore, the main challenge with VOS is its uncertainty, the absence of a general rule which would make all instances of an electronic chips equal towards voltage scaling, which make manufacturers generally turn their backs to VOS, preferring to keep a comfortable margin above the threshold.

In next Subsections 1.1.2 and 1.1.3, accuracy is traded for performance in a reproducible way, with results which are independent from hardware and totally dependent from the programmer/designer's will, and thus more likely to be used in the future at industrial scale.

1.1.2 Algorithmic Approximations

A more secure way than VOS to save energy is achieved by algorithmic approximations. Indeed, modifying algorithm implementation to make them deliver their results in less cycles or using less intensively costly functions potentially leads to important savings. First, the approximable parts of the code or algorithm must be identified, i.e. the parts where the gains can be maximized despite a moderate impact on the output quality. Various methods for the identification of these approximable parts are proposed in [17, 18, 19]. These methods are mostly perturbation-based, meaning errors are inserted in the code, and the output is simulated to evaluate the impact of the inserted error. As all simulation-based methods, these methods may not be scalable to large algorithms and only consider a limited number of perturbation types. Once the approximable parts identified by an automatized method or manually, depending on the kind of algorithm part to approximate (computation loops, mathematical functions, etc), different techniques can be applied.

One of the main techniques for reducing the cost of algorithm computations is loop perforation. Indeed, most signal processing algorithms consist in quite simple functions repeated a high number of times, e.g. for Monte Carlo simulations, search space enumeration or iterative refinement. In these three cases, a subset of loop iterations can simply be skipped, yet returning good enough results [20]. Using complex mathematical functions such as exponentials, logarithms, square roots or trigonometric functions is very area, time and energy-costly compared to basic arithmetic operations. Indeed, accurate implementations may require large tables and long addition and multiplication-based iterative refinement. Therefore, in applications using intensively these mathematical functions, releasing accuracy for performance can be source of important savings. A first classical way to approximate functions is polynomial approximation, using tabulated polynomials representing the function in different ranges. In the context of approximate computing, iterative-refinement based mathematical approximations are also very interesting since they allow loop perforation discussed previously. Reducing the number of iterations can be applied to CORDIC algorithms, very common for trigonometric function approximations [14]. Several efficient approximate mathematical functions were proposed for specialized hardware such as FPGAs [21] or Single Instruction Multiple Data (SIMD) hard-

ware [22].

1.1.3 Approximate Basic Arithmetic

The level of approximation we focus on in this thesis is the approximation of basic arithmetic operations, which are addition, subtraction and multiplication. These operations are the base for most functions and algorithms in classical computing, but also the most energy-costly compared to other basic CPU functions such as register shifts or binary logical operators, meaning that a gain in performance or energy on these operations automatically induces an important benefit for the whole application. They are also statistically intensively used in general-purpose CPU computing: in ARM processors, ADD and SUB instructions are the instructions the most used after $LOAD$ and $STORE$. The approximation of these basic operators is explored along two different angles in this thesis. On the one hand, approximate representations of numbers is discussed, more precisely the approximate representation of real numbers in computer arithmetics, using floating-point and fixed-point arithmetic. On the other hand, approximate computing using modified functions for integer arithmetical functions of addition, subtraction and multiplication is explored used in fixed-point arithmetic and compared to existing methods.

Approximations using floating-point arithmetic are discussed in Section 1.2, approximations using fixed-point arithmetic in Section 1.3 and approximate integer operators are presented in Section 1.4.

1.2 Relaxing Accuracy Using Floating-Point Arithmetic

Floating-point (Floating-Point (FlP)) representation is today the main representation for real numbers in computing, thanks to a potentially high dynamic, totally managed by the hardware. However, this ease of use comes with relatively important area, delay and energy penalties. FlP representation is presented in Section 1.2.1. Then, FlP addition/subtraction and multiplication are described in Section 1.2.2. Ways to relax accuracy for performance in FlP arithmetic is then discussed in Section 1.2.3.

1.2.1 Floating-Point Representation for Real Numbers

In computer arithmetic, the representation of real numbers is a major stake. Indeed, most powerful algorithms are based on continuous mathematics, and their accuracy and stability is directly related to the accuracy of the number representation they use. However, in classical computing, an infinite accuracy is not possible since all representations are contained in a finite bit width. To address this issue, having a number representation as accurate for very small numbers and very large numbers is important. Indeed, large and small numbers are dual, since multiplying (resp. dividing) a number by another large number is equivalent to dividing (resp. multiplying) by a small number. Giving the same relative accuracy to numbers whatever their amplitude is can only be achieved giving the same impact to their most significant digit. In decimal representation, this is achieved with scientific notation, representing the significant

value of the number in the range $[1, 10[$, weighted by 10 elevated to a certain power. FlP representation in radix-2 is the pendant of the scientific notation for binary computing. The point in the representation of the number is "floating" so the representative value of the number (or $mantissa$) represents a number in $[1, 2[$, multiplied by a power of 2. Given an M-bit mantissa, a signed integer exponent of value e, often represented in biased representation, and a sign bit s, any real number between limits defined by M and E the number of bits allocated to the exponent e can be represented with a relative step depending on M by:

$$(-1)^s \times m_{M-1}.m_{M-2}m_{M-3} \cdots m_1 m_0 \times 2^e.$$

With this representation, any number under this format can be represented using $M+E+1$ bits as showed in Figure 1.1. A particularity of binary FlP with a mantissa represented in $[1, 2[$ is that its Most Significant Bit (MSB) can only be 1. Knowing that, the MSB can be left implicit, freeing space for one more Least Significant Bit (LSB) instead.

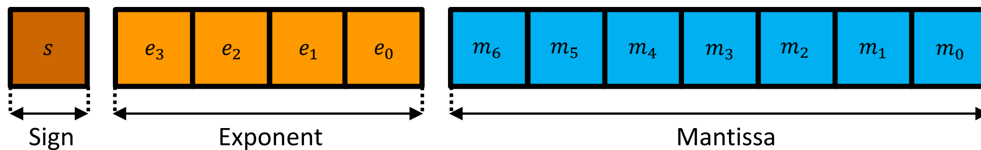


Figure 1.1 – 12-bit floating-point number with 4 bits of exponent and 7 bits of mantissa

Nevertheless, automatically keeping the *floating* point at the right position along computations requires an important hardware overhead, as discussed in Section 1.2.2. Managing subnormal numbers (numbers between 0 and the smallest positive possible representable value), the values 0 and infinity also represent an overhead. Despite this additional cost, FlP representation is today established as the standard for real number representation. Indeed, besides its high accuracy and high dynamic, it has the huge advantage of leaving the whole management of the representation to the hardware instead of leaving it to the software designer, significantly diminishing developing and testing time. This domination is sustained by IEEE 754 standard, last revised in 2008 [15], which sets the conventions for floating-point number possible representation, subnormal numbers management and the different cases to be handled, ensuring a high portability of programs. However, such a strict normalization implies:

- an important overhead for throwing flags for the many special cases, and even more important overhead for the management of these special cases (hardware or software overhead),
- and a low flexibility in the width of the mantissa and exponent, which have to respect the rules of Table 1.1 for 32, 64 and 128-bit implementation.

As a first conclusion, the constraints imposed to FlP representation by IEEE 754 normalization imply a high cost in terms of hardware resource, which highly counterbalance its accuracy benefits. However, as discussed in Section 1.2.3, taking liberties with FlP can significantly increase the accuracy/cost ratio.

Precision	Mantissa width	Exponent width	Max decimal exponent	Exponent bias
Single precision	24	8	38.23	127
Double precision	53	11	307.95	1023
Quadruple precision	113	15	4931.77	16383

Table 1.1 – IEEE 754 normalized floating-point representation

1.2.2 Floating-Point Addition/Subtraction and Multiplication

As discussed later in Section 1.3, integer addition/subtraction is the simplest arithmetic operator. However, in FlP arithmetic, it suffers from a high control overhead. Indeed, several steps are needed to perform the FlP addition:

- First, the difference of the exponents is computed.
- If the difference of the exponents is superior to the mantissa width, the biggest number is directly issued (this is the $far\ path$ of the operator – one of the numbers is too small to impact the addition).
- Else, if the difference of the exponents is inferior to the mantissa width, one of the inputs' mantissas must be shifted so bits of same significance are facing each other. This is the $close\ path$, by opposition with the $far\ path$.
- The addition of the mantissas is performed.
- Then, rounding is performed on the mantissa, depending on the dropped bits and the rounding mode selected.
- Special cases are then handled (zero, infinity, subnormal results), and the output sign.
- Then, mantissa is shifted so it represents a value in $[1, 2[$, and the exponent is modified depending on the number of shifts.

FlP addition principle is illustrated in Figure 1.2 taken from [23]. More control can be needed, depending on the implementation of the FlP adder and the specificities of the FlP representation. For instance, management of the implicit 1 implies to add 1s to the mantissas before addition, and an important overhead can be dedicated to exception handling.

For cost comparison, Table 1.2 shows the performance of 32-bit and 64-bit FlP addition, using ac_float type from Mentor Graphics, and 32-bit and 64-bit integer addition using ac_int type, generated using the HLS and power estimation process of the second version of APX-PERF framework described in Section 4.1, targeting 28nm FDSOI with a 200 MHz clock and using $10,000$ uniform input samples. FlP addition power was estimated activating the close path 50% of the time. These results show clearly the overhead of FlP addition. For 32-bit version, FlP addition is $3.5\times$ larger, $2.3\times$ slower and costs $27\times$ more energy than integer addition. For 64-bit version, FlP addition is $3.9\times$ larger, $1.9\times$ slower and costs $30\times$ more energy. The

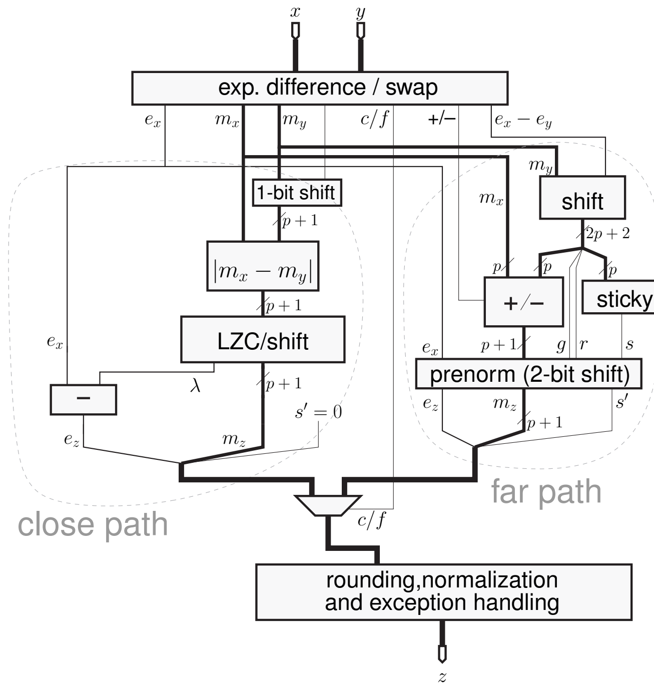


Figure 1.2 – Dual-path floating-point adder [23]

	Area (μm^2)	Total power (mW)	Critical path (ns)	Power-Delay Product (fJ)
32-bit ac_float	653	4.39E−4	2.42	1.06E−3
64-bit ac_float	1453	1.12E−3	4.02	4.50E−3
32-bit ac_int	189	3.66E−5	1.06	3.88E−5
64-bit ac_int	373	7.14E−5	2.10	1.50E−4

Table 1.2 – Cost of FlP addition vs integer addition

overhead seems to be roughly linear with the size of the operator, and the impact of numbers representation is highly impacting the performance. However, it is showed in Chapter 5 that this high difference shrinks when the impact of accuracy is taken into account.

FlP multiplication is less complicated than addition as only a low control overhead is necessary to perform the operation. Input mantissas are multiplied using a classical integer multiplier (see Section 1.3), while exponents are simply added. At worse, a final $+1$ on the exponent can be needed, depending on the result of the mantissas multiplication. The basic architecture of a FlP multiplier is described in Figure 1.3 from [23]. Obviously, all classical hardware over-

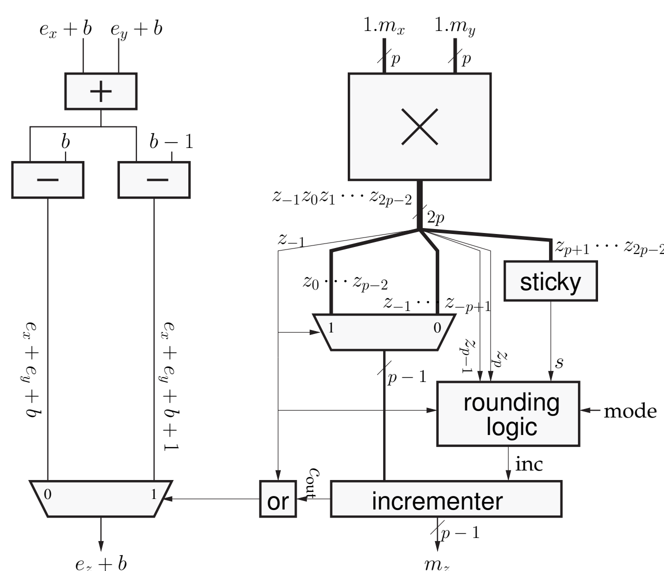


Figure 1.3 – Basic floating-point multiplication [23]

heads needed by FlP representation are necessary (rounding logic, normalization, management of particular cases). Table 1.3 shows the difference between 32-bit and 64-bit floating-point multiplication using Mentor Graphics ac_float and 32-bit and 64-bit fixed-width[1] integer multiplication using ac_int data type, with the same experimental setup than discussed before for the addition.

A first observation on the area shows that the integer multiplication is 48% larger than FlP version for 32-bit version, and 37% larger for 64-bit version. This difference is due to the smaller size of the integer multiplier in the FlP multiplication, since it is limited to the size of the mantissa (24 bits for 32-bit version, 53 bits for 64-bit version). Despite the management of the exponent, the overhead is not large enough to produce a larger operator. However, if the overhead area is not very large, 32-bit FlP multiplication energy is $11\times$ higher than the integer multiplication energy, while 64-bit version is $37\times$ more energy-costly. It is interesting to note that the difference of energy consumption between addition and multiplication is much more important for integer operators than for FlP. For 32-bit version for instance, integer multiplica-

[1]An operator is considered as fixed-width when its output has the same width as its inputs. In the considered multiplication case, half of the output LSBs is truncated.

	Area (μm^2)	Total power (mW)	Critical path (ns)	Power-Delay Product (fJ)
32-bit ac_float	1543	8.94E−4	2.09	1.87E−3
64-bit ac_float	6464	6.56E−3	4.70	3.08E−2
32-bit ac_int	2289	6.53E−5	2.38	1.55E−4
64-bit ac_int	8841	1.84E−4	4.52	8.31E−4

Table 1.3 – Cost of floating-point multiplication vs integer multiplication

tion consumes $4.7\times$ more energy than integer addition, while this factor is only $1.4\times$ for 32-bit FlP multiplier compared to 32-bit FlP adder. Therefore, using multiplication in FlP computing is relatively less penalizing than for integer multiplication, typically used in Fixed-Point (FxP) arithmetic.

1.2.3 Potential for Relaxing Accuracy in Floating-Point Arithmetic

There are several possible opportunities to relax accuracy in floating-point arithmetic to increase performance. The main one is simply to use word-length as small as possible for the mantissa and the exponent. With normalized mantissa in $[1, 2[$, reducing the word-length corresponds to pruning the LSBs, which comes with no overhead. Eventually, rounding can be performed at higher cost. For the exponent, the transformation is more complicated if it is represented with a bias. Indeed, if e is the exponent width, an implicit bias of $2^e - 1$ applies to the exponent in classical exponent representation. Therefore, reducing the exponent to a width e' means that a new bias must be applied. The original exponent must be added $2^{e'} - 2^e$ (< 0) before pruning MSBs, implying hardware overhead at conversion. The original exponent must represent a value in $\left[-2^{e'-1} + 1, 2^{e'-1}\right]$ to avoid overflow. In practice, it is better to keep a constant exponent width to avoid useless overhead and conversion overflows which would have a huge impact on the quality of the computations, even if they are scarce.

A second way to improve computation at inferior cost is to play with the implicit bias of the exponent. Indeed, increasing the exponent width increases the dynamic towards infinity, but also the accuracy towards zero. Thus, if the absolute maximum values to be represented are known, the bias can be chosen so it is just large enough to represent these values. This way, the exponent gives more accuracy to very small values, increasing accuracy. However, using a custom bias means that the arithmetical operators (addition and multiplication) must consider this bias in the computation of resulting exponent, and the optimal bias along computation may diverge to $-\infty$. To avoid this, if the original $2^e - 1$ exponent bias is kept, exponent bias can be simulated by biasing the exponents of the inputs of each or some computations using shifting. For the addition, biasing both inputs adding $2^{e_{\text{in}}}$ to the exponent implies that the output will also be represented biased by $2^{e_{\text{in}}}$. For the multiplication, the output will be biased by $2^{e_{\text{in}}+1}$.

Keeping an implicit track of the bias along computations allows to know any algorithm output bias, and eventually to perform a final rescaling of the outputs.

Finally, accuracy can be relaxed in the integer operators composing FlP operators, i.e. the integer adder adding mantissas in FlP addition close path, and the integer multiplier in the FlP multiplication. Indeed, they can be replaced by the approximated adders and multipliers described in Section 1.4 to improve performance relaxing accuracy. However, as the most part of the performance cost is in control hardware more than in integer arithmetic part, the impact on accuracy would be strong for a very small performance benefit. The same approximation can be applied on exponent management, but the impact of approximate arithmetic would be huge on the accuracy and is strongly unadvised.

More state-of-the-art work on FlP arithmetic is developed in Section 5.1, more particularly on HLS using FlP custom computing cores.

1.3 Relaxing Accuracy Using Fixed-Point Arithmetic

Aside from FlP, a classical representation for real numbers is Fixed-Point (FxP) representation. This Section presents generalities about FxP representation in Section 1.3.1, then presents the classical models for quantization noise in Section 1.3.2. Finally, hardware implementations of addition and multiplication are respectively listed and discussed in Sections 1.3.3 and 1.3.4.

1.3.1 Fixed-Point Representation for Real Numbers

In fixed-point representation, an integer number represents a real number multiplied by a factor depending on the implicit position of the point in the representation. A real number x is represented by the FxP number x_{FxP} represented on n bits with d bits of fractional part by the following equation:

$$x_{\mathrm{FxP}} = \left| x \times 2^d \right|_r \times 2^{-d},$$

where $|\cdot|_r$ is a rounding operator, which can be implemented using several functions such as the ones of Section 1.3.2. The representation of a 12-bit two's complement signed FxP number with a 4-bit integer part is depicted in Figure 1.4.

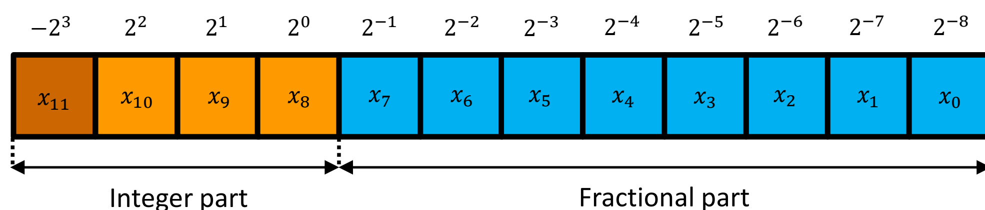


Figure 1.4 – 12-bit fixed-point number with 4 bits of integer part and 8 bits of fractional part

In two's complement representation, the MSB has a negative weight, such as a binary

number $x_{\text{bin}} = \{x_i\}_{i \in [0,n-1]}$ represents the integer number x_{int} the following way:

$$x_{\text{int}} = x_{n-1} \times \left(-2^{n-1}\right) + \sum_{i=0}^{n-2} x_i \times 2^i$$

Therefore, the two's complement n-bit FxP number represented by x_{FxP} with d-bit fractional part is worth:

$$\begin{aligned} x_{\text{FxP}} &= x_{\text{int}} \times 2^{-d} \\ &= x_{n-1} \times \left(-2^{n-d-1}\right) + \sum_{i=0}^{n-2} x_i \times 2^{i-d}. \end{aligned}$$

1.3.2 Quantization and Rounding

Representing a real number in FxP is equivalent to transforming a number represented on an infinity of bits to a finite word-length. This reduction is generally referred as *quantization of a continuous amplitude signal*. Using a FxP representation with a d-bit fractional part implies that the step between two representable values is equal to $q = 2^{-d}$, referred as *quantization step*. The process of quantization results in a quantization error defined by:

$$e = x_q - x, \tag{1.1}$$

where x is the original number to be quantified and x_q the resulting quantified number.

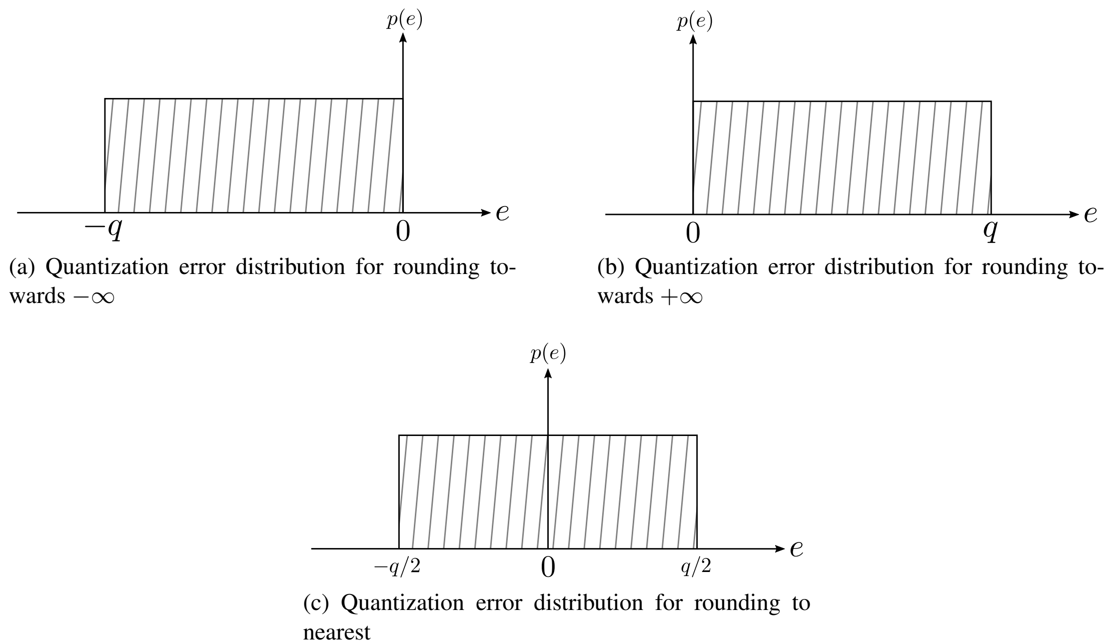
Quantization of continuous amplitude signal is performed differently depending on the rounding mode chosen. Several classical rounding modes are possible:

1. Rounding towards $-\infty$ (RD). The infinity of bits dropped are not taken into consideration. This results in a negative quantization error e and is equivalent to truncation.

2. Rounding towards $+\infty$ (RU). Again, the infinity of bits dropped are not taken into consideration, and the nearest superior representable value is selected. This is equivalent to adding q to the result of the truncation process.

3. Rounding towards 0 (RZ). If x is negative, the number is rounded to $+\infty$. Else, it is rounded to $-\infty$

4. Rounding to Nearest (RN). The nearest representable value is selected – if the MSB of the dropped bits is 0, then x_q is obtained by rounding to $-\infty$ (truncation) – else x_q is obtained by rounding towards $+\infty$. The special value where the MSB of the dropped part is 1 and all the following bits are 0 can lead whether to rounding up or down depending on the implementation. Choosing one or another case does not change anything in the error distribution in the case of continuous amplitude signal, which is also the case for discrete case, as said below.



(a) Quantization error distribution for rounding towards $-\infty$

(b) Quantization error distribution for rounding towards $+\infty$

(c) Quantization error distribution for rounding to nearest

Figure 1.5 – Distribution of continuous signal quantization error for rounding towards $\pm\infty$ and rounding to nearest

The quantization error produced by rounding towards $\pm\infty$ and to nearest discussed above are depicted in Figure 1.5. RZ method has a varying quantization error distribution depending on the sign of the value to be rounded. As described in [24] and [25], the additional error due to the quantization of a continuous signal is uniformly distributed between its limits ($[-q, 0]$ for truncation, $[0, q]$ for rounding towards $+\infty$ and $[-q/2, q/2]$ for rounding to nearest) and statistically independent on the quantized signal. Therefore, the mean and variance of the error are perfectly known in these cases and are indexed in Table 1.4. Thanks to its independence to the signal, quantization error can be seen as an additive uniformly distributed white noise q such as depicted in Figure 1.6. This representation of quantization error is the base of FxP representation error analysis discussed in the next Chapter.

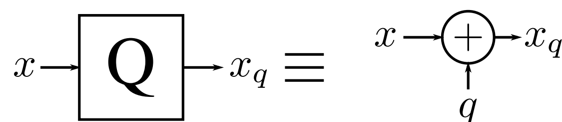
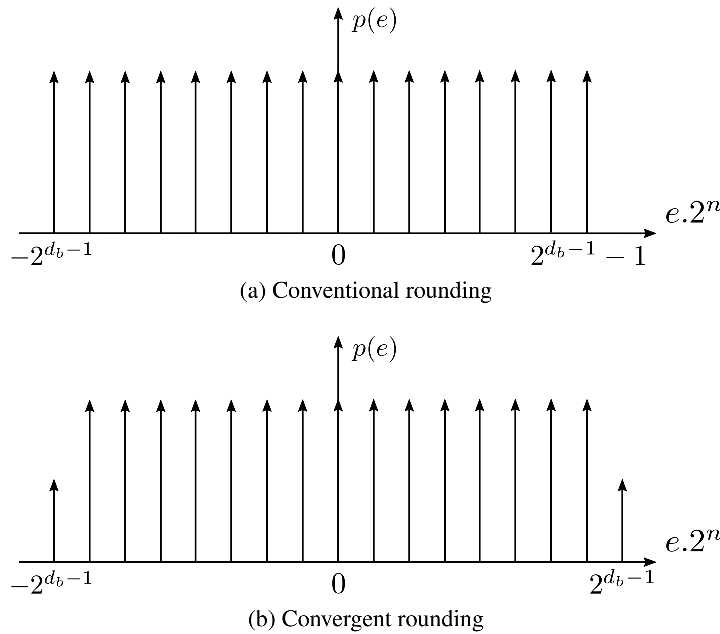


Figure 1.6 – Representation of FxP quantization error as an additive noise

The previous paragraphs describe the properties of the quantization of a continuous signal. However, in FxP arithmetic, it is necessary to reduce the bit width along computations to avoid a substantial growth of the necessary resources. Indeed, as discussed in Section 1.3.4, an integer

multiplication needs to produce an output which width is equal to the sum of its inputs to get a perfectly accurate computation. However, the LSBs of the result are often not significant enough to be kept, and so a reduction of data width must be performed to save area, energy and time. Therefore, it is often necessary to reduce a FxP number x_1 with d_1-bit fractional part to another FxP number x_2 with d_2-bit fractional part, where $d_2 < d_1$. This reduction leads to a discrete quantization error distribution, depending on the number of bits dropped $d_b = d_1 - d_2$. This distribution is still uniform for RD, RU and RN rounding methods, but has a different bias. Moreover, for RN method, this bias is not 0, and depends on the direction of rounding chosen when the MSB of the dropped part is 1 and all the other dropped bits 0. This can lead to divergences when accumulating a large number of computations.



(a) Conventional rounding

(b) Convergent rounding

Figure 1.7 – Comparison of quantization error distribution of conventional rounding and convergent rounding

To overcome this possible deviation, Convergent Rounding to Nearest (CRN) was proposed in [26]. When the special case cited above is met, the rounding is once performed toward $+\infty$, and once towards $-\infty$. This way, the quantization error distribution gets centered to zero. Figure 1.7 shows the effect of CRN on the error distribution compared to simpler RN. On Figure 1.7a, the discrete uniform distribution of RN method has a negative bias of $\frac{q}{2}\left(2^{-d_b}\right)$. As the highest error occurs for the special *in between* case, distributing this error using alternatively RD and RU paradigms balances the error, lowering by half the highest negative error and moving its impact to a new spike removing the bias as showed on Figure 1.7b. However, this compensation slightly increases the variance of the quantization error.

The values of the mean μ_e and variance σ_e^2 of RD, RU, RN and CRN rounding methods

are listed in Table 1.4. As a reminder, RZ method is mixing RD and RU, and thus the final distribution of error strongly depends on the distribution of the signal around 0.

	Continous Amplitude		Discrete Amplitude	
	μ_e	σ_e^2	μ_e	σ_e^2
RD	$-\frac{q}{2}$	$\frac{q^2}{12}$	$-\frac{q}{2}\left(1-2^{-d_b}\right)$	$\frac{q^2}{12}\left(1-2^{-2d_b}\right)$
RU	$\frac{q}{2}$	$\frac{q^2}{12}$	$\frac{q}{2}\left(1-2^{-d_b}\right)$	$\frac{q^2}{12}\left(1-2^{-2d_b}\right)$
RN	0	$\frac{q^2}{12}$	$\frac{q}{2}\left(2^{-d_b}\right)$	$\frac{q^2}{12}\left(1-2^{-2d_b}\right)$
CRN	0	$\frac{q^2}{12}$	0	$\frac{q^2}{12}\left(1-2^{-2d_b+1}\right)$

Table 1.4 – Mean and variance of quantization error depending on the rounding method and type of signal

The implementation of rounding methods discussed above depend of two parameters. Indeed, when quantizing x_1 with d_1-bit fractional part to x_2 with d_2-bit fractional part, where $d_2 < d_1$, only the following information is needed:

- the *round bit*, which is the value of the bit indexed by $d_1 - d_2 - 1$ of x_1,
- and the *sticky bit*, which is a logical *or* applied to the bits $\{0, ..., d_1 - d_2 - 2\}$ of x_1.

The extraction of *round* and *sticky* bits is illustrated in Figure 1.8. The horizontal stripes in x_1 correspond to the *round bit*, and the tilted stripes to the bits implied in the computation of the *sticky bit*. Here, both are worth 1, and the rounding logic outputs 1, which can correspond to RU, RN, or CRN. The possible functions performed by the different rounding functions which can be implemented in rounding logic block of Figure 1.8 are listed in Table 1.5. It is important to notice that for RD method, the value of round and sticky bits have no influence on the rounding direction. For RN method, if the default rounding direction is towards $+\infty$ when round/sticky bits are 1/0, then the value of the sticky bit does not influence the rounded result. If it is up, the sticky bit has to be considered. Therefore, some hardware simplifications can be performed for RD and RN (down case) methods, by just dropping the unused bits.

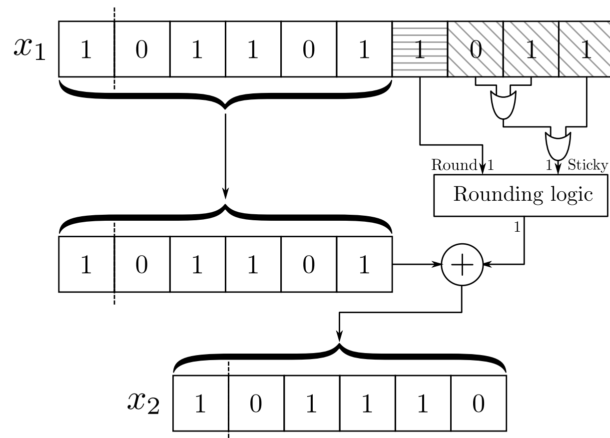


Figure 1.8 – Example of quantization and rounding of a 10-bit fixed-point number to a 6-bit fixed-point number

Round bit	Sticky bit	RD	RU	RN	CRN
0	0	−	−	−	−
0	1	−	+	−	−
1	0	−	+	Always − or always +	Alternatively − and +
1	1	−	+	+	+

Table 1.5 – Rounding direction depending on the value of round and sticky bits

1.3.3 Addition and Subtraction in Fixed-Point Representation

The addition/subtraction in FxP representation is much simpler than the one of FlP described in Section 1.2.2. Indeed, FxP arithmetic is entirely based on integer arithmetic. Adding two FxP numbers can be performed in 3 steps:

1. Aligning the points of the two numbers, shifting one of them (software style) or driving their bits to the right input in the integer adder (hardware design style).

2. Adding the inputs using an integer adder.

3. Quantizing the output using methods of Section 1.3.2.

In this section, we will consider the addition (respectively subtraction) of two signed FxP numbers x and y with a total bit width of resp. n_x and n_y, a fractional part width of d_x and d_y and an integer part width of $m_x = n_x - d_x$ and $m_y = n_y - d_y$. In the rest of this chapter, a n_x-bit FxP number x with m_x-bit integer part will be noted $x(n_x, m_x)$.

To avoid overflows or underflows, the output $z(n_z, m_z)$ of the addition/subtraction of x and y must respect the following equation:

$$m_z = \max(m_x, m_y) + 1. \tag{1.2}$$

Moreover, an accurate addition/subtraction must also respect:

$$d_z = \max(d_x, d_y). \tag{1.3}$$

The final process for FxP addition of $x(6,2)$ and $y(8,3)$ returning $z(9,4)$ then quantized to $z_q(6,4)$ is depicted by Figure 1.9. For the subtraction $x - y$, the classical way to operate is to compute $y' = -y$ before performing the addition $x + y'$. In two's complement representation, this is equivalent to performing $y' = \overline{y} + 1$, where $\overline{y}$ is the binary inverse of y. The inversion is fast and requires small circuit, and adding 1 can be performed during the addition step of Figure 1.9 (after shifting) to avoid performing one more addition for the negation.

As a first conclusion about FxP addition/subtraction, the cost of the operation mostly depends on two parameters: the cost of shifting the input(s), and the efficiency of integer addition. From this point, we will focus on the integer addition, which represent the majority of this cost.

The integer addition can be built from the composition of 1-bit additions, taking each three inputs – the input bits x_i, y_i and the input carry c_i and returning two outputs – the output sum bit s_i and the output carry c_{i+1}. This function is realized by the Full Adder (FA) function of Figure 1.10 which truth table is described in Table 1.6.

The simplest addition structure is the Ripple Carry Adder (RCA), built by the direct composition of FAs. Each FA of rank i takes the input bits and the input carry of rank i. It returns the output bit of rank i and the output carry of rank $i + 1$, which is connected to the following full adder, resulting in the structure of Figure 1.11. This is theoretically the smallest possible area for an addition, with a complexity of $O(n)$. However, this small area is counterbalanced

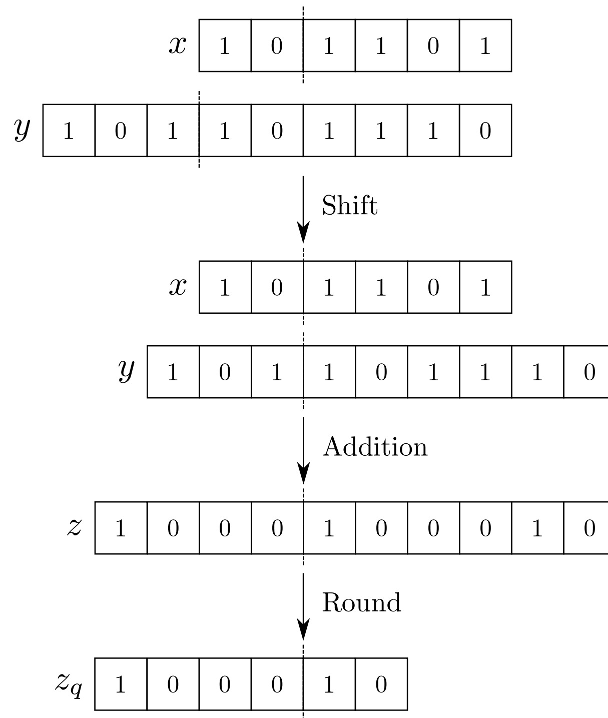


Figure 1.9 – Fixed-point addition process of $x(6, 2)$ and $y(8, 3)$ returning $z(9, 4)$ quantized to $z_q(6, 4)$

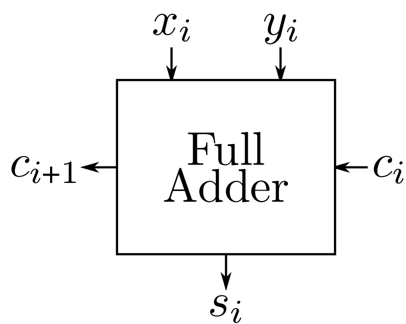


Figure 1.10 – One-bit addition function – Full adder (or compressor 3:2)

x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 1.6 – Full adder truth table

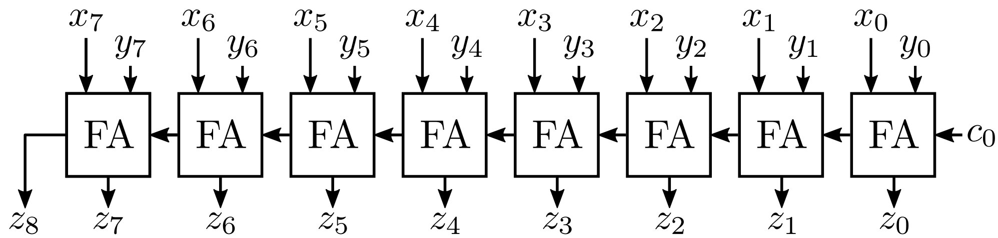


Figure 1.11 – 8-bit Ripple Carry Adder (RCA)

by a high delay also in $O(n)$, while the theoretical optimum is $O(\log n)$ [27]. Therefore, RCA is only implemented when area budget is critical.

A classical improvement issued from the RCA is the Carry-Select Adder (CSLA). The CSLA is composed of elements containing two parallel RCA structures, one taking 0 as input carry and the other taking 1. When the actual value of the input carry is known, the correct result is selected. This pre-computation (or prediction) of the possible output values increases speed, which can reach at best a complexity of $O(\sqrt{n})$ when the variable widths of the basic elements are optimal, while more than doubling the area compared to classical RCA. An 8-bit version of CSLA is depicted in Figure 1.12, with basic blocks of size 2-3-3 from LSB to MSB. Therefore, the resulting critical path is 3 FAs and 3 multiplexers 2-1, from input carry to output carry, instead of 8 FAs for RCA. It is important to note that CSLA structure can be applied to any addition structure such as the ones described below, and not only RCA, which can lead to better speed performance.

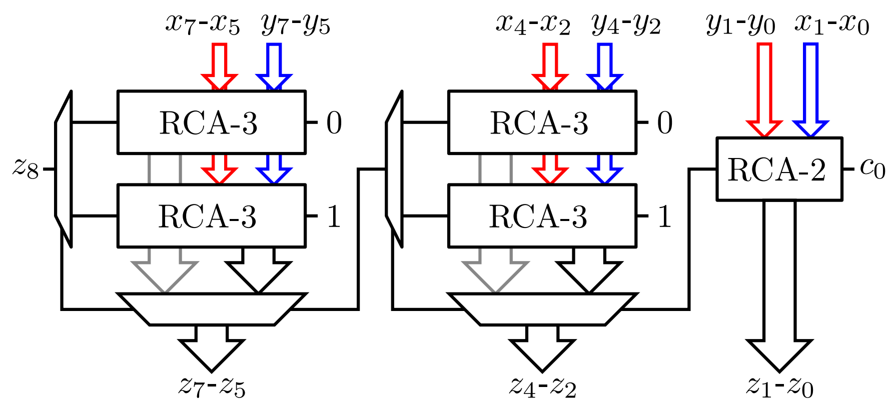


Figure 1.12 – 8-bit RCA-based Carry-Select Adder (CSLA) with 2-3-3 basic blocks

As already stated, the longest path in the adder starts from the input carry (or the input LSBs) and ends at the output carry (or output MSB). Therefore, propagating the carry across the operator as fast as possible is a major stake as long as high speed is required. It can whether be done duplicating hardware like for CSLA, but it can also be achieved by prioritizing the carry propagation. In FA design, the output is computed together with the output carry. In Carry-

Lookahead Adder (CLA) design, carry propagation is performed by an independent circuit so the carries at the MSBs position do not need to wait for all outputs of inferior rank to be computed to be available. For this, two peculiar values need to be calculated at each bit position: the $generate$ and $propagate$ bits g_i and p_i defined by:

$$\begin{aligned} p_i &= x_i \oplus y_i, \\ g_i &= x_i \wedge y_i. \end{aligned} \quad (1.4)$$

Using these values obtained with very small circuitry, the carry of rank i is extracted from the carry of previous rank by the following relation:

$$c_i = g_{i-1} \vee (c_{i-1} \wedge p_{i-1}) . \quad (1.5)$$

Then by recurrence, any carry signal can be deduced knowing any carry signal of inferior rank and all $propagate$ and $generate$ bits of intermediate rank. The addition output bit z_i is then simply deduced by the following relation:

$$z_i = p_i \oplus c_i . \quad (1.6)$$

For instance, knowing c_i, the four following carries are defined by the equations:

$$\begin{aligned} c_{i+1} &= g_i \vee (c_i \wedge p_i) , \\ c_{i+2} &= g_{i+1} \vee (g_i \wedge p_{i+1}) \vee (c_i \wedge p_i \wedge p_{i+1}) , \\ c_{i+3} &= g_{i+2} \vee (g_{i+1} \wedge p_{i+2}) \vee (g_i \wedge p_{i+1} \wedge p_{i+2}) \vee (c_i \wedge p_i \wedge p_{i+1} \wedge p_{i+2}) , \\ c_{i+4} &= g_{i+3} \vee (g_{i+2} \wedge p_{i+3}) \vee (g_{i+1} \wedge p_{i+2} \wedge p_{i+3}) \vee (g_i \wedge p_{i+1} \wedge p_{i+2} \wedge p_{i+3}) \\ &\quad \vee (c_i \wedge p_i \wedge p_{i+1} \wedge p_{i+2} \wedge p_{i+3}) . \end{aligned} \quad (1.7)$$

The direct translation of these equations into hardware leads to faster carry generation compared to RCA, but also leads to an important area overhead.

However, parallel prefix adders, which are based on CLA paradigm, show better area performance. The main idea is to propagate g and p bits to get equivalent couples (p', g') which turns the computation of any c_i in Equation 1.5 independent of c_{i-1}, so we finally get:

$$\begin{aligned} c_i &= g'_{i-1}, \\ z_i &= p'_i \oplus c_i \\ &= p'_i \oplus g'_{i-1}, \end{aligned} \quad (1.8)$$

and so all outputs can be computed in parallel. This equivalent representation is obtained thanks to a series of 4:2 compressors, extracting an equivalent couple (p'_i, g'_i) from couples (p_i, g_i) and (p_j, g_j) where $j < i$, performing:

$$\begin{aligned} p'_i &= p_j \wedge p_i, \\ g'_i &= g_i \vee (g_j \wedge p_i) . \end{aligned} \quad (1.9)$$

The details and mathematical proof of this method are available in [28]. The use of this (p, g) 4:2 compressor has led to the creation of several parallel prefix adders, such as the Brent-Kung

Adder (BKA) and the Kogge-Stone Adder (KSA) [29], respectively depicted for their 16-bit versions by Figures 1.13 and 1.14. The parallel adders are the fastest adders with a delay complexity of $O(\log n)$, but with a superior area ($2n - \log_2(n) - 2$ compressors for BKA and $n\log_2(n) - n + 1$ compressors for KSA but with lower fan-out).

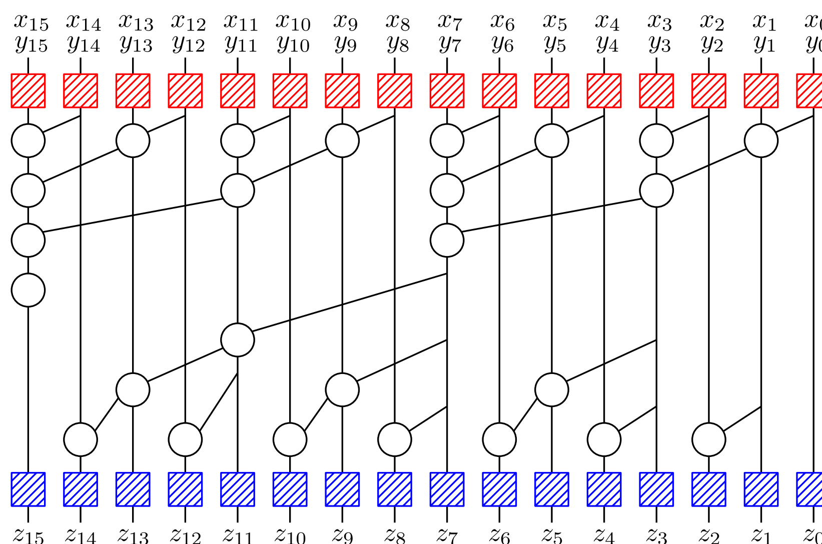


Figure 1.13 – 16-bit BKA. Red-striped square converts (x_i, y_i) to (p, g) (Equation 1.4) and blue-striped square converts (p, g) to z_i (Equation 1.8). Circles represent (p, g) compressors.

As a conclusion about integer adders used for FxP addition, several addition structures exist. This section only presents the main principles, many other instances based on these principles do exist, many being described in [28]. What is important to observe is that integer adders have an area of minimum complexity $O(n)$ and time complexity $O(\log n)$. However, both these complexity cannot be achieved by a same structure. Reaching the minimum time complexity implies parallelism and so larger area, whereas the smallest area implies longer critical path and so higher delay. In the next section, the carry-save addition method is presented in the context of summand grid reduction in multiplication. This is why it was not handled in this section.

1.3.4 Multiplication in Fixed-Point Representation

As for addition, FxP multiplication is performed by integer multiplication. Unlike addition, no alignment of the inputs is necessary. Given the multiplication of two FxP numbers $x(m_x, d_x)$ and $y(m_y, d_y)$, the result $z(m_z, d_z)$ must respect for its integer part:

$$m_z = m_x + my \tag{1.10}$$

to avoid under/overflows. Moreover, if there must be no loss of accuracy, the fractional part must also respect:

$$d_z = d_x + dy. \tag{1.11}$$

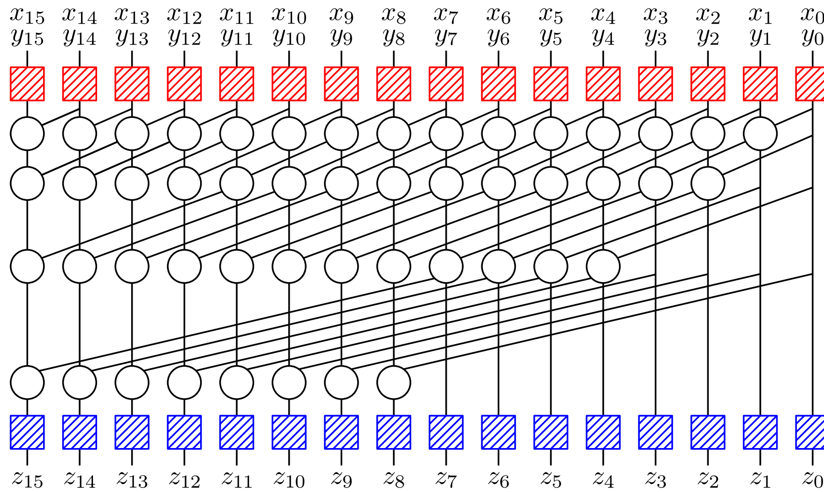


Figure 1.14 – 16-bit KSA. Red-striped square converts (x_i, y_i) to (p, g) (Equation 1.4) and blue-striped square converts (p, g) to z_i (Equation 1.8). Circles represent (p, g) compressors.

Thus, for the accurate multiplication of n-bit inputs, a $2n$-bit result is returned. Therefore, compared to addition where only 1 more bit is necessary, multiplication is a potential source for high resource needs downstream, which definitely justifies the necessity of quantizing numbers along computations, as presented in Section 1.3.2.

Integer multiplication can be split in two phases – generation of summand grid, and summand grid addition, leading to the scheme showed in Figure 1.15. Compared to higher-base,

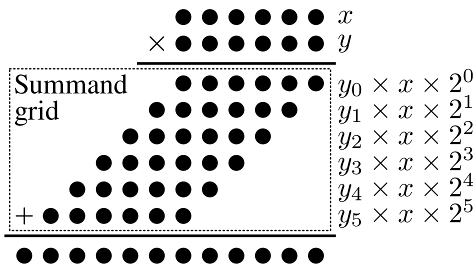


Figure 1.15 – General integer multiplication principle applied on 6-bit input

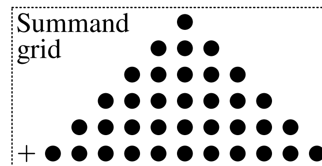


Figure 1.16 – General visualization of 6-bit multiplication summand grid

binary multiplication is much simpler. Indeed, only two values are possible for all summands, 0 or the value of the multiplicand, which can leads to major simplifications. Indeed, the generation of each line of the summand grid of an n-bit multiplier can be performed by n 2-to-1-multiplexers selecting whether the input bits are x_i or 0, controlled by the value of the bit y_j corresponding to the current line. Therefore, the most expensive part of the multiplier in terms of resources is the carry-save reduction of the summand grid to reduce it to a final addition.

The summand grid can be visualized by an $n/2$-stage triangle, as showed on Figure 1.16. The reduction of the tree is achieved by several stages of FA and Half Adder (HA), until only two lines are left. A HA can be seen as a simplified FA (see Section 1.3.3) with only two inputs instead of three. A HA is built with only two logic gates instead of five for FA. FAs perform a 3-2 compression illustrated by Figure 1.17 and HAs a 2-2 transformation as in Figure 1.18.

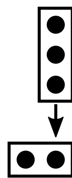


Figure 1.17 – Full adder compression – requires 5 gates

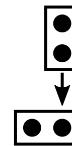
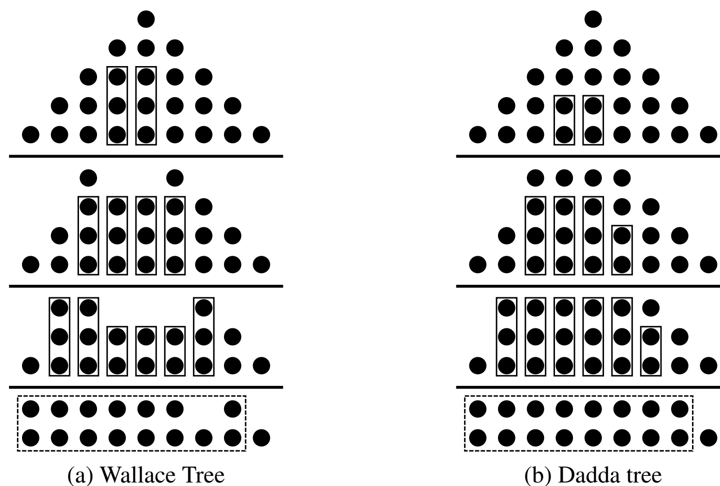


Figure 1.18 – Half adder transformation – requires 2 gates

Thus, the complexity of the multiplier in terms of speed and area depends on how the summand grid reduction is organized. Different classical methods to build reduction trees exist in the literature, most famous ones being Wallace tree [30] and Dadda tree [31]. Wallace tree reduces the partial product bits as early as possible, whereas Dadda tree reduces them as late as possible. This leads to two different kinds of architectures, Wallace tree being the fastest, whereas Dadda tree implementation is smaller. Figure 1.19 shows the difference between Wallace and Dadda trees in a 5-bit multiplier context. Wallace tree requires 9 FAs and 3 HAs (51 gates) before final 8-bit addition, whereas Dadda tree needs 8 FAs and 4 HAs (48 gates).



(a) Wallace Tree (b) Dadda tree

Figure 1.19 – Wallace and Dadda trees schemes applied to 5-bit multiplication summand grid reduction. The dashed rectangle corresponds to final 8-bit addition. Both require three stages, but Wallace tree takes 51 gates while Dadda tree only requires 48.

Computing multiplications based on partial product reduction such as with Wallace or

Dadda trees is a good compromise between speed and area. Indeed, only a few reduction stages are necessary for the reduction before the final addition (2 stages for 8-bit multiplication, 6 stages for 16-bit and 8 stages for 32-bit), which represents an acceptable overhead. Tree multipliers have a delay complexity in $O(\log n)$.

A particular sort of tree multiplier is the array multiplier. It is made with one-sided Carry-Save Adder (CSA) reduction tree (less efficient than distributed trees), which makes it slower ($O(n)$) and theoretically larger than previously discussed multipliers. And the final computation is performed by a RCA, which is the slowest possible adder as discussed in Section 1.3.3. However, it is very interesting in VLSI design thanks to its regularity which implies small wires ensuring a compact layout. This regularity also implies fine-grained pipelining possibilities. Figure 1.20 is a 6-bit signed array multiplier. The signed version is obtained using modified Baugh-Wooley two's-complement technique which consists in inverting the MSBs of all partial products except the last one which has all its bits inverted except the MSB. On Figure 1.20, AFA (resp. AHA) corresponds to a FA (resp. HA) whose inputs x_i and y_i are combined by an AND cell, NFA corresponds to a FA which inputs x_i and y_i are combined by a $NAND$ cell.

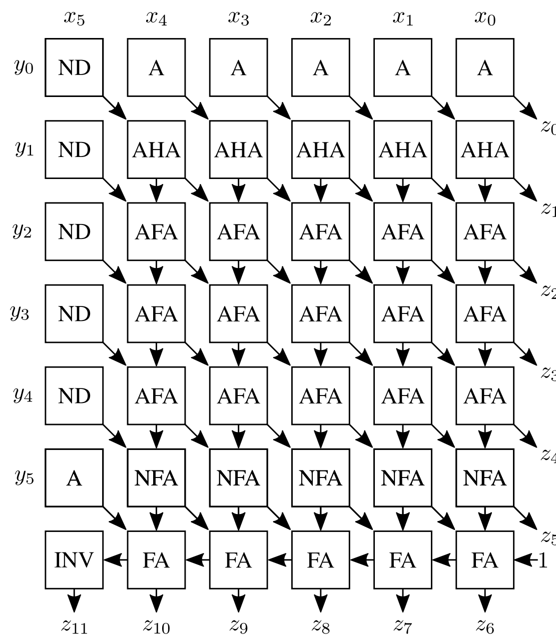
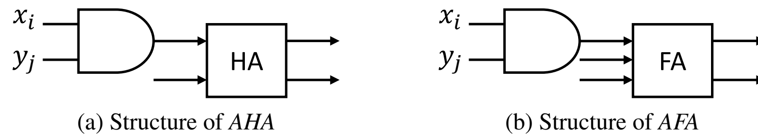


Figure 1.20 – 6-bit signed array multiplier – *AFA* (resp. *AHA*) corresponds to a FA (resp. HA) which inputs x_i and y_i are combined by an *AND* cell, *NFA* corresponds to a FA which inputs x_i and y_i are combined by a *NAND* cell. *AHA* and *AFA* structure are depicted in Figure 1.21.

Previously discussed multipliers are fast but their area is important. E.g, array multiplier area complexity is $O\left(n^2\right)$, whereas it is possible to reach an $O(n)$ complexity with a sequential multiplier such as the one presented on Figure 1.22. An n-bit sequential multiplier needs a n-bit right-shift register for input y controlling a multiplexer selecting x or 0. The output of



(a) Structure of *AHA*

(b) Structure of *AFA*

Figure 1.21 – Structures of *AHA* and *AFA*

the multiplexer is then accumulated in the MSB half of another register shifting one bit right at each new addition. Therefore, the whole computation is only performed by an adder which can be one of the several presented in Section 1.3.3, chosen whether for delay or area performance.

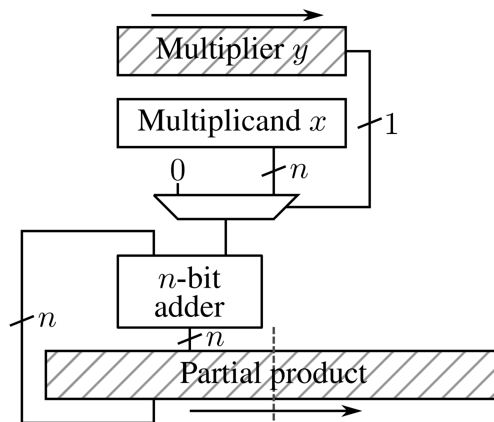


Figure 1.22 – Sequential multiplier – boxes hatched in grey are right-shift registers

In this section, several ways to manage computation time or area of the summand grid were presented. However, improvements can be done upstream in order to reduce the initial size of the summand grid. The most classical and efficient way is to apply modified Booth encoding on y. In Radix-4 Booth encoding, y is encoded so only $\lfloor n/2 \rfloor + 1$ lines in the summand grid are generated instead of n. However, this implies circuitry overhead for the encoding. Indeed, n-bit y operand needs to be transformed into $\lfloor n/2 \rfloor + 1$ actions to perform on x for partial product generation. These actions consists in a multiplication of x by an element of the set $\{-2, -1, 0, 1, 2\}$, which translates into 0 to 2 left shifts during the partial products generation, as well as possible negation. The decision of the corresponding action is driven by the rules of Table 1.7.

Higher radix encoding such as Radix-8 Booth encoding do exist, but increasing the radix leads to much higher encoding complexity and a high cost due to a more important number of shifts or dense wiring of input bits x_i, which tend to cancel the benefits of the reduction of the number of partial products. However, Radix-4 or Radix-8 Booth encoding techniques tend to be faster than classical Radix-2 multiplication, especially for large bit widths, which imply a large summand grid.

$y_{n\text{-}1}\ y_{n\text{-}2}\ y_{n\text{-}3}\ \cdots\ y_5\ y_4\ y_3\ y_2\ y_1\ y_0\ 0$

$Y_{\lfloor n/2 \rfloor+1}\quad Y_2\quad Y_1\quad Y_0$

y_i	y_{i-1}	y_{i-2}	$Y_{\lfloor i/2 \rfloor}$	Operation
0	0	0	0	$+0$
0	0	1	1	$+x$
0	1	0	1	$+x$
0	1	1	2	$+2x$
1	0	0	-2	$-2x$
1	0	1	-1	$-x$
1	1	0	-1	$-x$
1	1	1	0	$+0$

Table 1.7 – Radix-4 modified Booth encoding – Y_j is a radix-4 number which is not represented in the encoded. Only the corresponding operation is performed during partial product generation.

In this section, several multiplication techniques and optimizations were presented. Their efficiency strongly depends on the techniques used to generate and reduce the summand grid. Generally, for an n-bit multiplication, tree multiplication is $O(\log n)$ fast, with or without Booth encoding, whereas array multiplier is $O(n)$ fast and sequential multiplier $O(n \log n)$ fast. However, the sequential multiplier has an area in $O(n)$ and the array multiplier in $O(n^2)$, whereas Wallace or Dadda tree multipliers have an intermediate area. Array multiplier, despite being the largest and not the fastest, has compact layout possibilities and fine-grained pipelining possibilities. The following section present approximate operators which try to overcome the limitations of addition and multiplication complexity, often taking the previously presented operators as a work basis.

1.4 Relaxing Accuracy Using Approximate Operators

In Section 1.3, hardware integer addition and multiplication were presented in the context of FxP arithmetic. These operators are accurate, meaning they always return a mathematically correct value. However, many applications do not need calculations to be perfectly accurate since a degraded output can be tolerated. This is why these past decades, many researchers tried to break performance limitations of accurate arithmetic by proposing an important number of approximate operators. Section 1.4.1 presents a collection of previously published approximate adders and Section 1.4.2 presents some multipliers.

1.4.1 Approximate Integer Addition

If adders are the most basic arithmetic operators, they nevertheless are intensively used in all applications. Moreover, they are also directly used in some multiplier architectures and in many

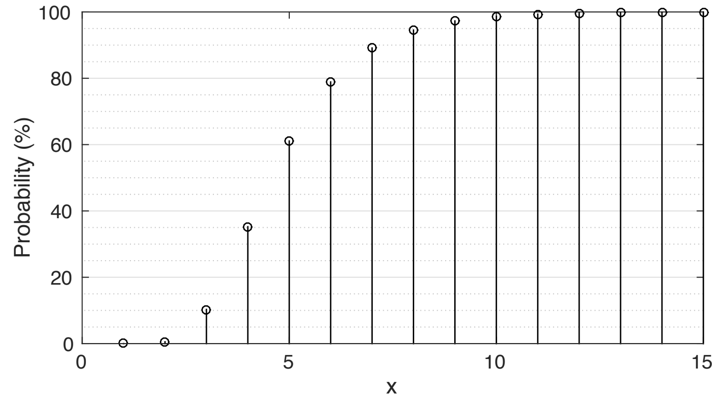


Figure 1.23 – Probability for the longest carry chain of a 64-bit adder to be inferior to x as a function of x

implementations of more complex functions, such as exponential, logarithms, functions using CORDIC algorithm, etc. Improving their latency, area or power consumption is therefore a big stake for all arithmetic operator design field. For an n-bit adder, optimal delay complexity is $O(\log n)$ and area complexity $O(n)$ [27]. It is thus impossible to find perfectly accurate adders getting below these complexities. For breaking these barriers, approximate adders were created. This section presents a non-exhaustive list of them.

As stated in Section 1.3.3, the critical path of an n-bit addition is located between the input LSBs (considering the adder has no input carry) and the z_n output, which can be considered as the adder's output carry. Therefore, the best source of improvements in addition is based on the ability to break the path of this critical carry chain. Indeed, most of the time during addition, the whole carry chain is unused and yet limiting the frequency of the adder and spending energy in glitch. It is showed in [32] that the probability for a series of n coin tosses, the longest run of heads that does not exceed x is the series $A_n(x)$, defined by:

$$A_n(x) = \begin{cases} 2^n & \text{if } n \leq x, \\ \sum_{0 \leq j \leq x} A_{n-1-j}(x) & \text{otherwise.} \end{cases} \tag{1.12}$$

Using this series, the longest carry chain with respectively a probability of 99% and 99.99% is given by Table 1.8. E.g. for a 256-bit adder, the longest propagation of a carry will be 20 bits with only 0.01% chance for it to be longer. It can also be noticed that the probability for the longest carry chain not to exceed a given x has a fast growth, as illustrated by Figure 1.23, which shows the probability for the longest carry chain for a 64-bit adder to be inferior or equal to x as a function of x. This probability explicitly shows that cautiously breaking carry chains only causes few chances for the result to be false. However, breaking a long carry chain is likely to cause a strongly erroneous output since an error with the weight of the MSBs of the carry chain can be performed. A balance must therefore be found between the occurrence of errors and their amplitude. Given an adder of width n, the probability of having a correct result

Bitwidth	**Longest run of 1's with 99% probability**	**Longest run of 1's with 99.99% probability**
64	11	17
128	12	18
256	13	20
512	14	21
1024	15	22
2048	16	23

Table 1.8 – Bounds on the longest run of 1's with high probability

limiting the carry chain to x is given by:

$$P(n,x) = \left(1 - \frac{1}{2^{x+2}}\right)^{n-x-1}, \tag{1.13}$$

which also has a fast growth with x.

1.4.1.1 Sample Adder, Almost Correct Adder and Variable Latency Speculative Adder

In [33], S.-L. Lu proposes the *Sample Adder*, which will be denoted as Lu's Parallel Adder (LPA) from this point. Based on the previous remarks about the potential of breaking the carry chains, an LPA of width n is parameterized by k, the maximum width of the transmitted carry chain. Based on the structure of a parallel prefix adder, the LPA is made of k rows of k-bit carry-chain computing blocks (or inferior to k for boundary blocks) applied on propagate and generate circuits. The output carry of each block is transmitted to the corresponding rank, and the final sum is computed along with the corresponding sum bit. The example of a 16-bit LPA with $k = 4$ is represented in Figure 1.24. First, a conversion of (x_i, y_i) to (p, g) is performed, and carries values are computed in parallel. Finally, the sum bits x_i, yi and the computed carries c_i are added to get the output sum bit at each position. The structure of LPA makes it delay constant for a given k, independently from n. The author claims the adder to be faster and smaller than KSA and Han-Carlson Adder (HCA), with a constant delay complexity in $O(k)$ and an area complexity in $O(n)$.

A functionally similar adder denoted as Almost Correct Adder (ACA) is proposed in [34]. Indeed, the same principle of limiting the transmitted carry chain to a same k for each position is used. However, the implementation is different. To understand it, the *kill* bit must be added to generate-propagate scheme, giving Equations 1.14. The kill signal is set when both inputs are 0. When this situation occurs, an hypothetical input carry can not be transmitted.

$$\begin{aligned} p_i &= x_i \oplus y_i, \\ g_i &= x_i \wedge y_i, \\ k_i &= (\neg x_i) \wedge (\neg y_i). \end{aligned} \tag{1.14}$$

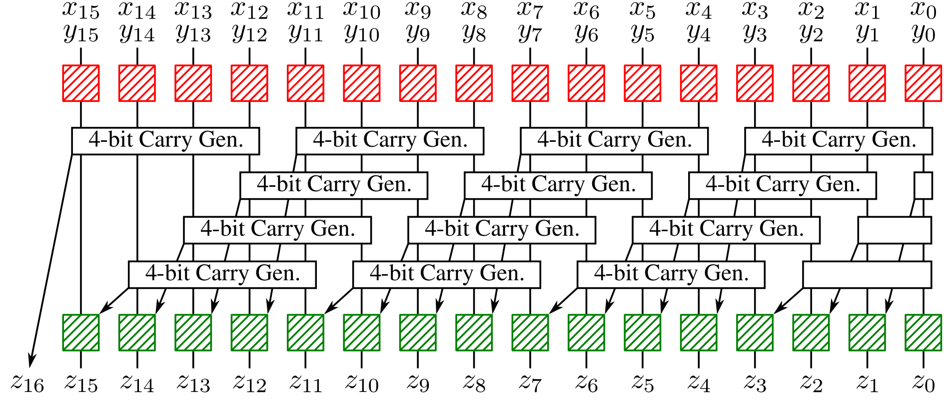


Figure 1.24 – 16-bit Sample Adder (LPA) with $k = 4$ – Red-striped square converts (x_i, y_i) to (p, g) (Equation 1.4) and green-striped square converts sum bit $x_i \oplus yi$ and c_i to output z_i.

Considering k_i, we get:

$$\begin{aligned} c_i &= \begin{cases} 0 & \text{if } k_i = 1, \\ 1 & \text{if } g_i = 1, \\ c_{i-1} & \text{otherwise } (p_i = 1). \end{cases} \\ s_i &= a_i \oplus b_i \oplus c_{i-1}. \end{aligned} \tag{1.15}$$

Using Equations 1.15, a matrix recursion can be found to express c_i as a function of any of its carry predecessors:

$$\begin{aligned} \begin{pmatrix} c_i \\ 1 \end{pmatrix} &= \begin{pmatrix} p_i & g_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix} = M_i \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix}, \text{ and by recursion,} \\ \begin{pmatrix} c_i \\ 1 \end{pmatrix} &= M_i M_{i-1} \cdots M_{i-k+2} M_{i-k+1} \begin{pmatrix} c_{i-k} \\ 1 \end{pmatrix}. \end{aligned} \tag{1.16}$$

Therefore, knowing the adder output carry c_{n+1} implies propagating, generating or killing carries from first carry c_0, performing $n - 1$ simple binary matrix products (performed operations are only logical OR and AND). ACA proposes to reduce this chain by limiting this series of matrix products to a given number, taking into account the low probability for the existence of a long carry chain. For instance, a 32-bit operator with a maximum considered carry chain of 8 taken into account for each output bit calculation will produce incorrect results only for cases where the longest carry chain should be greater to 8, which occurs for only 2.4%.

To summarize, an n-bit ACA with x-bit restricted carry chain will have to consider $n - x + 1$ carry chains of size x instead of a single $n - 1$-bit carry chain. However, successive x-bit carry chains have $x - 1$ bits recovering with their direct neighbour carry chains. As a consequence, an organization for a fast calculation of carry chains matrix $M_{i:i-x+1}$ is possible as showed in Figure 1.25, where it is applied to a 16-bit ACA with 6-bit carry chains. $M_{i:i-x+1}$ is the matrix product $\prod_{j=0}^{x-1} M_{i-j}$. By construction, an n-bit ACA with a k-bit considered carry chain has an area complexity $O(n \log k)$ and a time complexity $O(\log k)$. In [32], it is showed that the

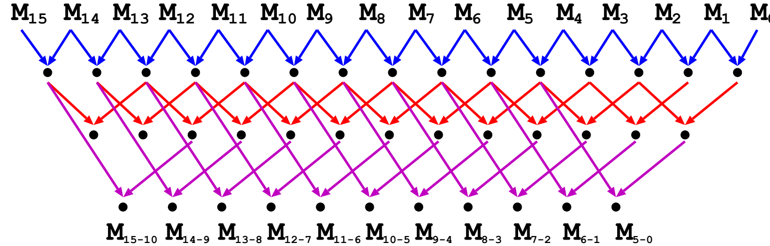


Figure 1.25 – Distribution of calculations for carry propagation matrix products [34]

expectation for the longest chain of ones for an n-bit sequence is $\log n - 2/3$, which in our case makes k proportional to $\log n$ for equal performance. Therefore, the final complexity of an n-bit ACA is $O\left(n \log \log n\right)$ for space complexity, which is near-linear even for relatively high values of n, and $O\left(\log \log n\right)$ for time complexity. Hence, the theoretical space complexity limit for accurate adder is nearly reached, whereas time complexity is exponentially beaten, so ACA can be considered as a fast approximate adder.

As previously mentioned, LPA and ACA produce the same function with different hardware layout. Figure 1.26 sums up the effect of approximation on the output on an 8-bit LPA or ACA with $k = 2$. Each colored rectangle takes the inputs considered in the computation of the corresponding output bit(s). Most approximate adders can have their function fully described by this type of figure, except for particular ones such as $ETAI$ described in Section 1.4.1.2 or configurable adders such as $AC2A$ mentioned in Section 1.4.1.3.

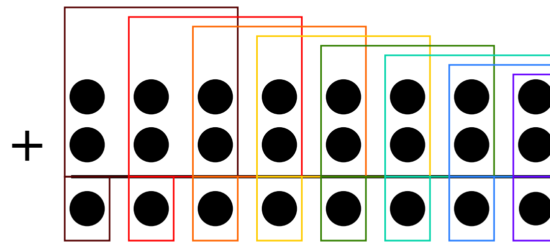
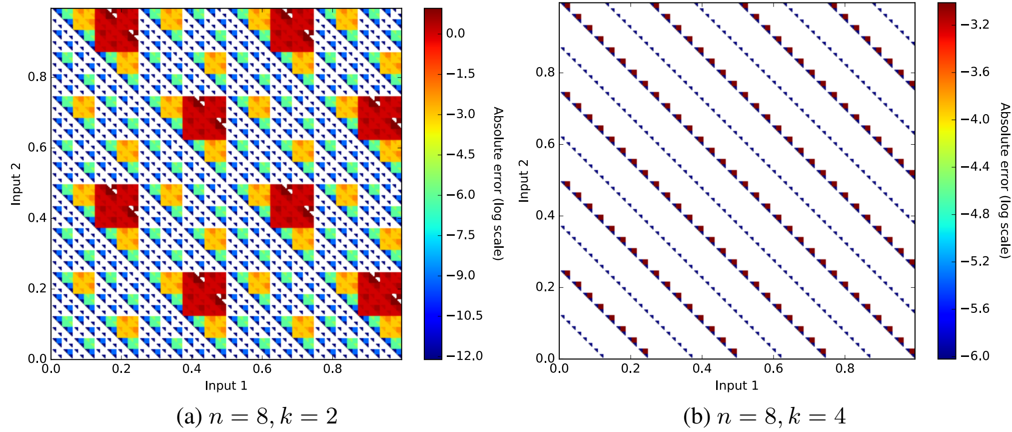


Figure 1.26 – Consideration of carries in LPA and ACA output computation for an 8-bit adder with $k = 2$ – Each color shows the inputs considered in the computation of the corresponding output bit.

Figure 1.27 shows the error maps for 8-bit ACA for $k = 2$ and $k = 4$ in log scale. The error map is the amplitude of error given all possible combinations of inputs x (input 1) and y (input 2). The white zones correspond to the inputs leading to no error, i.e. the inputs implying a carry chain inferior to k. As expected, $k = 4$ leads to scarcer error than $k = 2$. The error map is here represented using FxP-represented inputs with a 1-bit integer part. The theoretical highest possible error is therefore equal to $2 - q = 2 - 2^{1-n}$. For LPA and ACA, it is interesting to see that the error map seems to be fractal, which shows the structural different between the nature

of their error and the uniform nature of quantization noise issued by FxP.



(a) $n = 8, k = 2$ (b) $n = 8, k = 4$

Figure 1.27 – Error maps of 8-bit LPA and ACA adders for different values of k – White zones correspond to accurate calculations.

As pointed out, the ACA adder proposes high benefits in terms of delay with small area sacrifice compared to classical accurate adders. Moreover, it is possible to choose the deepness of the approximation by selecting the length of the maximal considered carry chain for each output bit. As reducing this length is source of error, an architecture going against the first interest of approximate computing is proposed in [34], the Variable Latency Speculative Adder (VLSA). This adder is totally accurate, but based on ACA. The method consists in the following steps:

1. calculate the sum using an (n, k) ACA, choosing k such as a relatively low number of errors occurs,
2. detect if an error has occurred, i.e. if a carry chain is bigger than k, and
3. if an error occurred, correct the sum in order to obtain an exact result.

This method, which provides a correct adder with variable latency from one unit to two, is only interesting if the sum of the ACA and the error recovery system does not exceed an equivalent state-of-the-art adder. An error recovery system which uses a maximum of the ACA structure to detect and correct the error is proposed in [34]. The error detection mechanism has to consider all chains of length $k + 1$ instead of k for the ACA. This leads to an $O(\log n)$ time complexity, which is higher than (n, k) ACA time complexity, but still more efficient than a traditional adder by two thirds according to the author. The correction system is an n/k-bit Carry Look-Ahead (CLA) block, which returns carries that were missed by the ACA because of a too long carry chain. This mechanism has about the same time-efficiency than the corresponding ACA, so the critical path will be the error detection mechanism. The schematic of the resulting architecture is given in Figure 1.28.

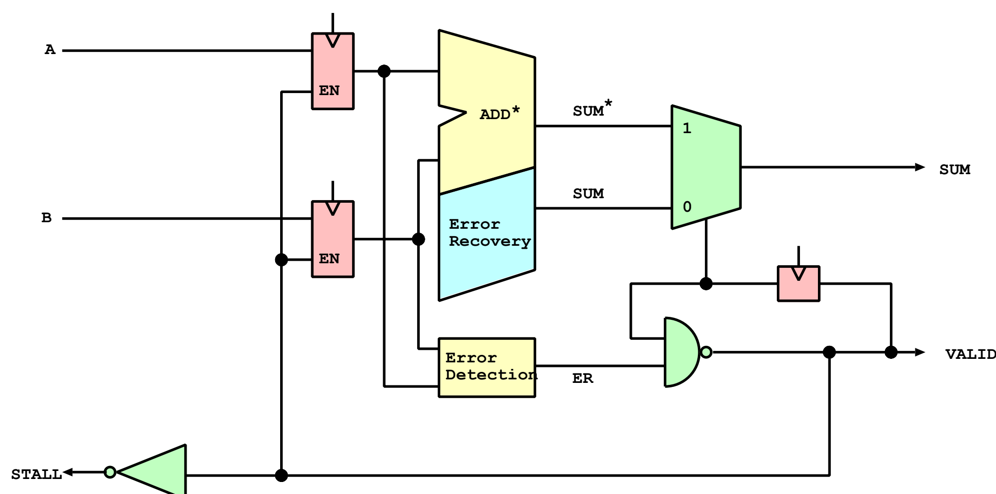


Figure 1.28 – Hardware implementation of VLSA [34]

Tests comparing ACA, ACA with error detection, ACA with error detection and recovery and a traditional fast adder provided by DesignWare are provided by [34]. ACA adders are optimally sized for an accuracy of 99.99% following Table 1.8 values. Results are showed on Figure 1.29. For ACA with no error detection and recovery, we can see a clear benefit in delay compared to traditional adder. They are both near-linear, but the proportional coefficient is much smaller for ACA. In terms of area, ACA is about 25% smaller than the traditional adder. For the ACA with error recovery, it can be noticed that it is nearly as fast as a traditional fast adder. Though, the error correction only occurs for 0.01% of computations. The average delay of corrected ACA is 0.9999 multiplied by the error detection delay, which is about $2/3$ of the traditional adder according to first graph of Figure 1.29. In terms of area, the ACA and its error recovery are together about $1.5\times$ larger than an optimal exact adder, but its complexity is linear.

To conclude about ACA:

- ACA takes advantage of deep carry propagation scarcity.
- It performs scarce errors depending on its design, but with a potential high amplitude, especially when a carry chain is incompletely considered on a bit of high significance.
- It has a near-linear delay even for high values of n, with a very low proportional coefficient compared to a fast adder.
- It covers a 25% smaller area compared to this same fast adder.
- Its structure allows efficient error detection and correction as proposed by VLSA structure, which constitutes a variable latency accurate adder.

In many more recent papers, ACA is still used as a reference for comparison with other approximate operators. Error detection and error correction proposed by VLSA can easily be

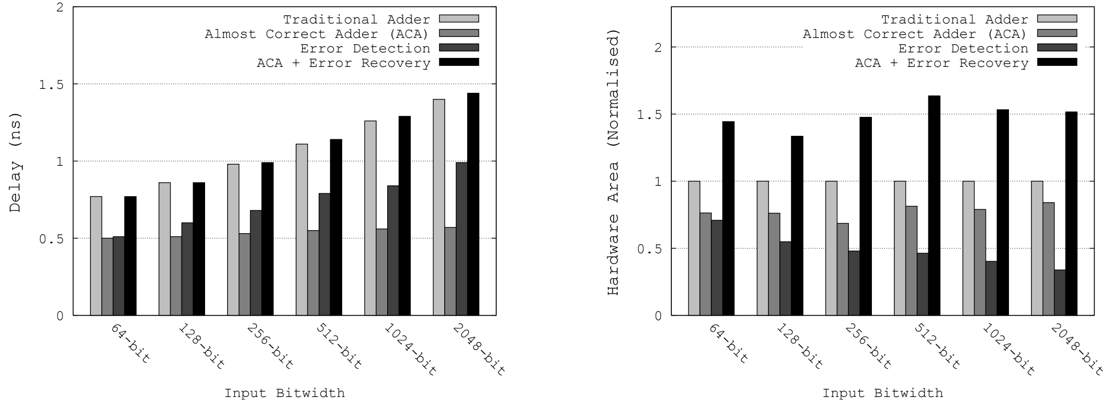


Figure 1.29 – Delay and area results for ACA with different bitwidth [34]

applied to any other approximate operator, and is a very interesting accurate structure for systems which can allow variable number of cycles for an operation. In this context VLSA can be seen as a 2-cycle addition, which can nearly always be bypassed in 1 cycle.

1.4.1.2 Error-Tolerant Adders

Between 2009 and 2010, Zhu proposed four approximate adders:

- Error-Tolerant Adder type I (ETAI) [35],
- Error-Tolerant Adder type II (ETAII)[36],
- Error-Tolerant Adder type II Modified (ETAIIM) [36], and
- Error-Tolerant Adder type IV (ETAIV) [37].

In [35], the first Error-Tolerant Adder (ETA) is presented, then referred as ETAI in the subsequent iterations [36, 37]. Its principle is simple: the most significant part (MSB side) of the adder is an accurate adder, and the least significant part (LSB side) is approximated with Algorithm 1. The inputs x_i and y_i of the approximate part are read from their MSB. When both are equal to 1, the calculations are stopped and all bits from rank i to the LSB are set to 1. This mechanism is depicted in Figure 1.30.

ETAI is made for fast approximation. Its goal is to round up the result as fast as possible when a $generate$ signal is met in the approximate part, i.e. both input bits are 1. In this way, the carry which should have been generated towards the MSB is compensated at best by maximizing the lower weight bits that have not been treated yet, that is to say all the bits from the carry generation to the LSB. The propagation of these 1s to the right is performed by a control block which was designed for propagating the information as fast as possible when the $two\text{-}1s$ case occurs, using a control signal. This block is composed of two types of sub-blocks:

- Control Signal Generating Cells of type I (CSGCI), which takes as inputs a couple (a_i, b_i) and the output control signal of rank $i + 1$ noted CTL_{i+1}, and

Algorithm 1 Computation of ETAI approximate part

```
i ← MSB_inac                                  ▷ index of the inaccurate part MSB
two_ones ← False
while i >= 0 and two_ones = False do
    if x_i = 1 and y_i = 1 then
        two_ones ← True
        z_i ← 1
    else
        z_i ← x_i ∨ y_i
    end if
end while
while i ≥ 0 do
    z_i ← 1
    i ← i − 1
end while
```

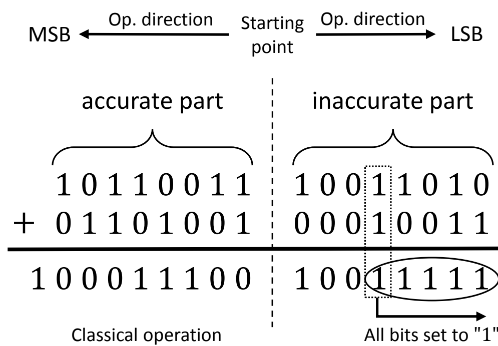


Figure 1.30 – Principle of ETAI

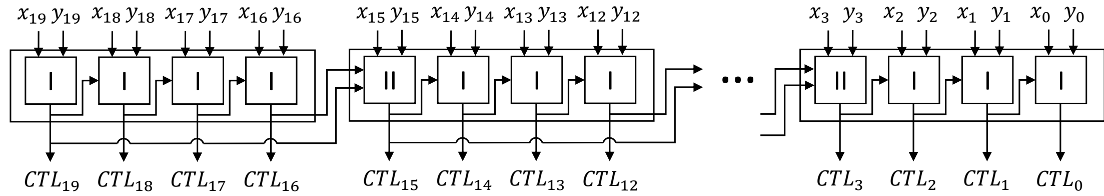


Figure 1.31 – ETAI control block [35]

Block	**Output value** CTL_i
CSGCI	$CTL_{i+1} \vee (x_i \wedge y_i)$
CSGCII	$CTL_{i+1} \vee CTL_{i+k} \vee (x_i \wedge y_i)$

Table 1.9 – Logical equations of CSGC type I and II

- Control Signal Generating Cells of type II (CSGCII), which is similar to CSGCI, but takes in addition as input another control signal of rank $i + k$, where k is fixed at design time.

CSGCII allows for the control signal to be short-circuited. This way, for an $n+m$-bit ETAI(n, m) with n-bit accurate part and m-bit approximate part, the critical 1s propagation is $k + 1 < m$, where k is the spacing between two CSGCII in the control block. The architecture of the control block is graphically described in Figure 1.31, for a 20-bit approximate part ETAI. CSGCI and CSGCII are simple logic blocks easily described by their binary logic in Table 1.9.

Once the control bits are generated/propagated for the whole approximate part, calculations are performed by a carry-free addition block, composed of a logic function called Modified XOR (MXOR) and presented at transistor-level logic in [35]. Studying the truth table of this new logic block reveals that it actually is a 3-input OR gate. Hence, each output z_i of the approximate part is given by $z_i = \mathtt{MXOR}(x_i, y_i, z_i) = x_i \vee y_i \vee CTL_i$.

The accuracy of ETAI is studied introducing Minimum Acceptable Accuracy (MAA) and Acceptance Probability (AP). MAA is a fixed value defining what is the desired minimum accuracy compared to an exact computation for the result, expressed as a percentage. AP is the probability for a given MAA to be reached by the operator. Simulation results are given by Figure 1.32. The first graph is obtained for several 16-bit ETAIs with 10^4 simulation sets of inputs, the second one has unknown simulation size parameters, but all ETAIs are designed with an approximate part representing 75% of the adder width. For 99% of MAA, ETAI$(8, 8)$ gives quite a high AP (about 99%), but when the approximate part length grows, AP dramatically drops when MAA increases. The second graph shows that for small operators, a difference of 1% on the MAA provokes a high drop in AP. For longer operators, the difference of AP for different MAA gets tinier. Errors performed by an ETAI(n, m) can be high in amplitude (nearly 2^{m-1}). ETAI is an adder which often performs errors, often with a low amplitude but sometimes with relatively high amplitude. Moreover, it has very bad performance for the addition of low amplitude numbers.

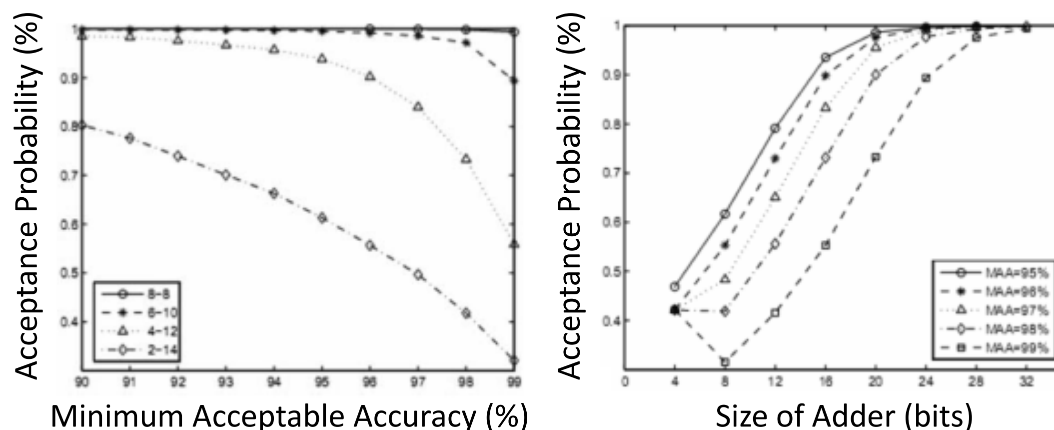


Figure 1.32 – ETAI accuracy simulation results [35]

Simulation results for the timing and delay performance of ETAI compared to most classical accurate adders are given in Table 1.12 [37] for $0.18\mu m$ CMOS process, for a 100 MHz frequency. In this table, CSK refers to Carry-Skip Adder (see [28]). The details about the size of these adders is not given. 100 sets of inputs were used for simulation, which is a bit low for giving an average of all different possible reactions of ETAI. These results give advantage to ETAI towards classical correct adders on this metric, with Power-Delay Product (PDP) savings up to more than 80% compared to carry-select implementation. Once again, results must be moderated, since the experimental conditions are not very clear (including size of adders, size of approximate part, value of CSGCII spacing parameter k).

Another fast approximate adder, ETAII, is presented in [36]. The structure of this adder is based on the same idea as LPA and ACA previously presented, i.e. shortening carry propagation paths. Indeed, carry propagation chain is cut in more little sub-chains of equal size. But contrary to ACA, every output bits do not take the same input propagated carry chain size. In the structure of ETAII given Figure 1.33, it can be noticed that each sub-block of size X is calculated using an exact adder, but taking into account only the carries generated inside and the propagated output carry of its predecessor carry generator sub-block. For carry generation, CLA blocks are implemented and for the sum generator, classical RCAs are used to minimize area. Well-designing the ETAII is entirely about finding the proper size for the carry propagation chains. The author studied the AP of 32-bit ETAII given different carry generator blocks sizes m. The results are available in Table 1.10. Simulations were led using 10^4 sets of inputs, which is quite low again for the adder width, so the results must be taken with caution. For a 32-bit adder, a high AP can be reached for a high MAA with quite a low carry chain length. E.g., more than 97% of tested inputs lead to an accuracy superior to 99% compared to the accurate value. Contrary to ETAI, ETAII has no relative accuracy disparity comparing low amplitude and high amplitude input sets. Simulation in [36] showed that AP for a given MAA is similar for ETAII whatever the range of inputs is, at least for $n/m = 4$. Comparison

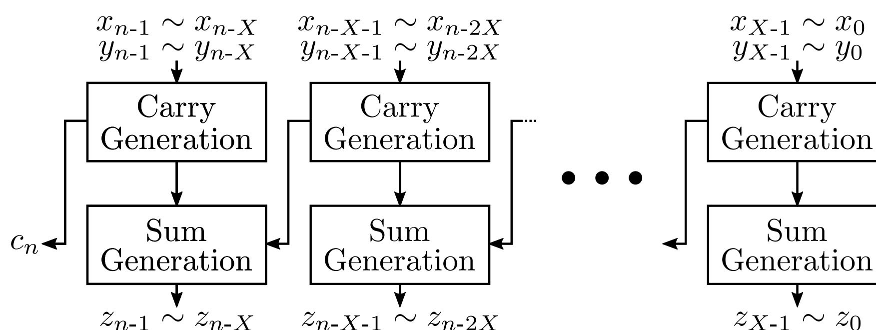


Figure 1.33 – Hardware implementation of ETAII [36]

MAA (%) \ **X**	1	2	4	8	16
100	0.0118	0.2204	0.8306	0.9961	1.0
99	0.5238	0.7927	0.9725	0.9985	1.0
98	0.5419	0.8075	0.9795	0.9985	1.0
97	0.5534	0.8119	0.9815	0.9995	1.0

Table 1.10 – AP as a function of MAA and carry propagation block size for 32-bit ETAII

of accuracy between an ETAI (with unknown parameters) and a 32-bit ETAII with 4-bit carry propagation block is given by the author. For instance, for a 99% MAA and inputs in the integer range $[\![0, 2^8]\!]$, ETAII presents a 97% AP against only 52% for ETAI.

In order to improve ETAII accuracy, a modified version is proposed in [36], the ETAIIM. Indeed, ETAII has a periodic structure, with the same substructure for high significance and low significance output bits, which makes it give as much importance to LSB part than to MSB part, which is generally unwise. In order to give more importance to MSB, ETAIIM takes the same structure as ETAII, but with a longer carry propagation chain for most significant bits. Figure 1.34 represents a 32-bit ETAIIM, with 4-bit carry propagation for all LSB and a 12-bit MSB carry propagation chain. Such a structure induces a longer critical path, corresponding to this longer carry chain and the corresponding sum generator. In this way, the previously described ETAIIM has a 99.9% AP for a 99% MAA against 97.0% AP for the same MAA for the corresponding ETAII. Hardware simulations results are presented in Table 1.12 [37], comparing classical correct adders to ETAII and ETAIIM. The only difference between ETAII and ETAIIM is the delay which is 64% higher for ETAIIM, but both operate with the same power.

ETAIV is presented in [37]. Its principle is the same as ETAII and ETAIIM, shortening the carry chain. However, ETAIV presents longer carry chains than ETAII and ETAIIM for an identical delay, in exchange for a higher energy cost. Each carry generator block is divided into two parts:

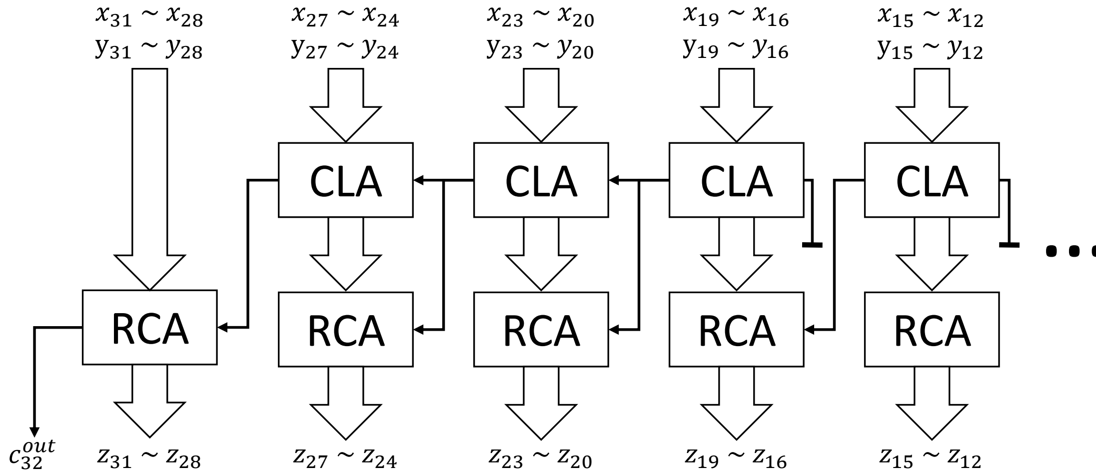


Figure 1.34 – Hardware implementation of ETAIIM

- the LSB carry generator is strictly identical than the one met in ETAII and ETAIIM and
- the MSB carry generator is composed of two parallel carry generator blocks: one taking value 0 (GND) as input carry, the other taking 1 (VDD). In this way, the two exhaustive possibilities for the concerned partial carry propagation are calculated. The good one is chosen thanks to a 2-bit multiplexer controlled by the LSB carry generator output carry.

The partial block diagram of ETAIV is depicted in Figure 1.37. The author performed simulations on 10^4 sets of inputs and accuracy results are given in Table 1.11, where X represents the number of bits of each sum generator (and thus each carry generator). ETAIV provides better results in terms of AP for a given MAA than ETAII (considering the same block size X).

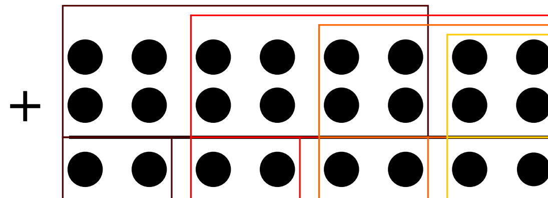


Figure 1.35 – Consideration of carries in ETAIV output computation for an 8-bit adder with $X = 2$ – Each color shows the inputs considered in the computation of the corresponding output bit.

The error maps for 16-bit ETAIV with respectively blocks of width $X = 2$ and $X = 3$ are depicted in Figure 1.36. For $X = 2$, triangular patterns are clearly visible, corresponding to areas where error is less. Generally, the error is quite high since many carries are not transmitted. For $X = 3$, the errors seem more homogeneous and with lower amplitude in average.

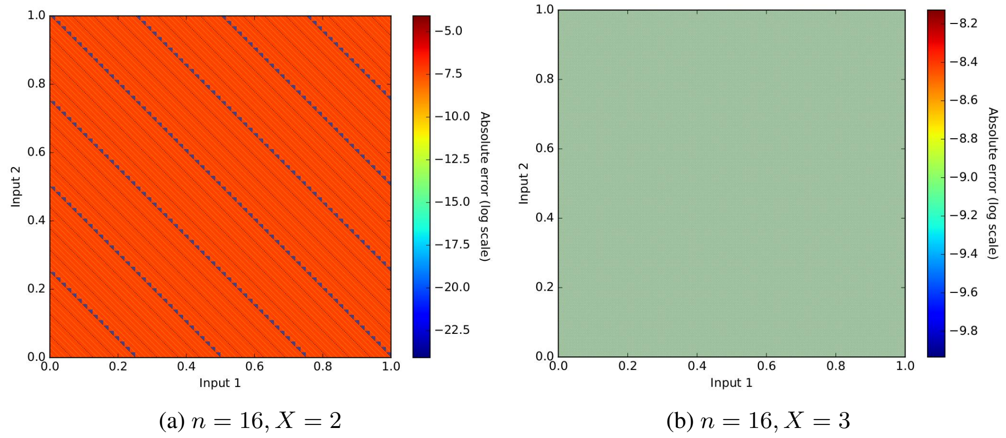


Figure 1.36 – Error maps of 16-bit ETAIV for different values of X

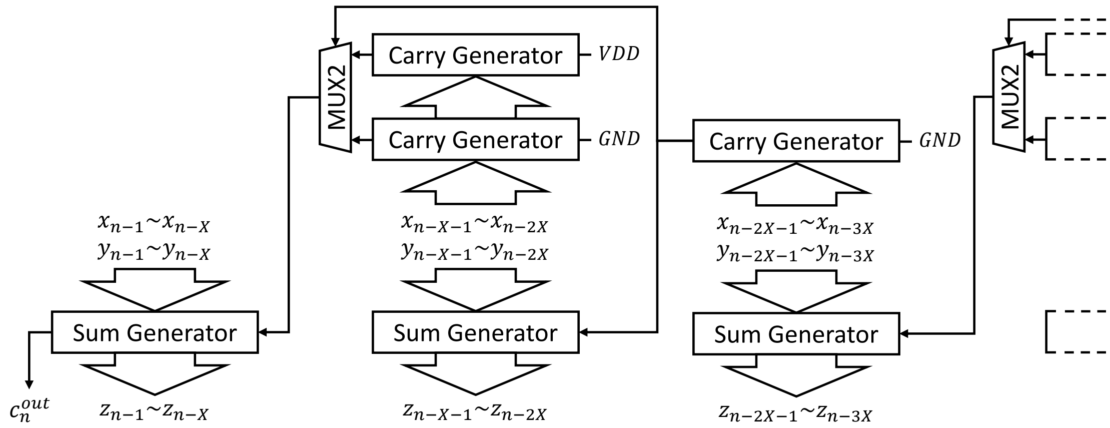


Figure 1.37 – Hardware implementation of ETAIV

MAA (%) \ X	1	2	4	8	16
100	0.0430	0.4136	0.9136	0.9985	1.0
99	0.6466	0.8848	0.9917	1.0	1.0
98	0.6695	0.9063	0.9966	1.0	1.0
97	0.6749	0.9127	0.9976	1.0	1.0

Table 1.11 – AP as a function of MAA and x for 32-bit ETAIV

Type	**Power (mW)**	**Delay (ns)**	**PDP (pJ)**	**Transistor (count)**	**PDP × transistor count**
RCA	0.22	4.04	0.89	896	797
CSK	0.46	2.90	1.33	1728	2298
CSL	0.60	3.06	1.84	2176	4004
CLA	0.51	2.37	1.21	2208	2672
ETAI	0.13	2.29	0.30	1006	302
ETAII	0.24	0.85	0.20	1372	274
ETAIIM	0.24	1.39	0.33	1372	453
ETAIV	0.25	1.03	0.26	1444	375

Table 1.12 – Simulation results for Error-Tolerant Adders [37]

Hardware simulation results for ETAIV are given in Table 1.12. Because of its longer critical paths, ETAIV needs slightly more power than ETAII and ETAIIM and has a 21% greater delay. The authors also show by simulation that the design has a good accuracy even for low amplitude inputs. It is a good alternative to ETAIIM, if a little amount of area can be traded for delay reduction.

With ETAI, ETAII and their modified versions ETAIIM and ETAIV, four subsequent approximate adders are proposed. ETAI is original by its reversed carry-propagation approximation, but is very inaccurate and has very poor performance for low amplitude inputs. However, it is a very low energy adder thanks to its low delay. ETAII, ETAIIM and ETAIV have a very different nature from ETAI, but they are very close the one from the others in their principle. The most accurate is ETAIV, followed by ETAIIM and ETAII. However, the fastest is ETAII, then ETAIV and ETAIIM. ETAII is the most energy-efficient, ETAIV coming second. The four operators have linear area complexity, but ETAI is the smallest, followed by ETAII and ETAIIM tied, not far from ETAIV. However, the authors compare these operators to some classical adders, but not to state-of-the-art fast adders such as KSA [29] or Ladner-Fischer Adder (LFA) [38].

1.4.1.3 Accuracy-Configurable Approximate Adder

In [39], Kahng proposes an Accuracy-Configurable Adder (AC2A). This operator is able to perform additions on different levels of accuracy on an unique implementation using a series of error correction systems which can be activated or deactivated thanks to power gating techniques. The approximate part of an n-bit AC2A is composed of $n/k-1$ recovering k-bit exact adders, whose only the MSB part is kept for the final result, as illustrated by Figure 1.38. Hence, errors are generated only when a carry is not propagated to the input of one of these sub-adders. As a functional point of view, AC2A resembles ETAIV, except that the sub-blocks recover by a half instead of $2/3$ and that the accuracy is tunable at run time on AC2A. The approximate computation part has a delay complexity of

$$O\left(\log_2 k + 1\right),$$

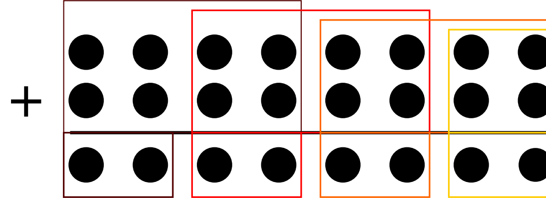


Figure 1.38 – Consideration of carries in AC2A output computation for an 8-bit adder with $k = 2$ – Each color shows the inputs considered in the computation of the corresponding output bit.

	k = 2	**k = 2**	**k = 3**	**k = 4**	**k = 5**
delay	0.5	0.65	0.75	0.83	0.89
area	0.87	1.05	1.12	1.15	1.12
dynamic power	0.44	0.68	0.84	0.95	1.00
pass rate	0.554	0.829	0.942	0.982	0.995

Table 1.13 – Estimated parameters of the approximate computation part of a 16-bit AC2A relatively to a conventional 16-bit CLA

an area complexity of

$$O\left((n-2k)\left(\log_2 k+1\right)\right),$$

and a dynamic power complexity of

$$O\left((n-2k)\left(\log_2 k+1\right)^2\right).$$

This means delay complexity beats the optimal delay for an accurate adder, whereas area complexity is slightly above the optimal accurate adder. Estimations of minimal delay, area, dynamic power and pass rate ($1 - \text{error rate}$) as a function of the value of k for the approximate computation part of a 16-bit AC2A compared to a conventional CLA is given in Table 1.13. For $k \leq 6$, the proposed computation part is more power-efficient than the classical CLA in terms of dynamic power, but its area is larger when $k \geq 2$. It can thus be assumed that its static power is superior to the one of CLA. When k decreases, the minimal delay of the proposed operator decreases, but the error rate increases because of the induced larger number of unconsidered propagated carries. Therefore, a trade-off must be found between delay and pass rate.

To characterize AC2A approximate part more completely, two metrics are used in [39]:

$$\text{ACC}_{amp} = 1 - \frac{|R_c - R_e|}{R_c}, \text{ and} \tag{1.17}$$

$$\text{ACC}_{inf} = 1 - \frac{B_e}{B_w}, \tag{1.18}$$

where R_c and R_e are the respective values of the correct and approximate results, B_e the number of erroneous bits in the approximate results and B_w the output bit-width. Therefore,

	CLA	LPA	AC2A	ETAI	ETAIIM
area (μm^2)	910	1356	923	876	678
delay (ps)	280	210	200	200	260
pass rate (%)	100	99.2	94.1	10.0	97.0
$\mathbf{ACC_{amp}}$	1.000	0.998	0.997	0.999	0.999
$\mathbf{ACC_{inf}}$	1.000	0.999	0.993	0.694	0.996
EDC area overhead (%)	N.A.	75	28	N.A.	15

Table 1.14 – Comparison of 16-bit AC2A approximate part with $k = 4$ with CLA and other approximate adders

$\texttt{ACC}_{amp}$ measures the relative amplitude of the error, 1.0 representing a perfect accuracy and the value decreasing with the error. $\texttt{ACC}_{inf}$ represents the proportion of correct bits, 1.0 representing a correct output bit sequence. Considering these metrics, comparisons of 16-bit AC2A approximate part with $k = 4$, ETAI [40] and ETAIIM [36] described in Section 1.4.1.2, LPA [33] described in Section 1.4.1.1 and an accurate CLA are given in Table 1.14. In this table, EDC stands for error detection and correction system. The best delay is obtained for AC2A and ETAI, but ETAI is 38% less area-costly than AC2A. However, ETAI pass rate and $\texttt{ACC}_{inf}$ are very low. In terms of area, AC2A is beaten by ETAIIM, which is also more accurate considering $\texttt{ACC}_{amp}$ and $\texttt{ACC}_{inf}$ metrics and pass rate, but with a delay overhead of 30%. LPA is nearly as fast as AC2A and also more accurate, but as it is based on a parallel prefix structure, its area is superior by 47%. The required area overhead for error detection and correction is 75% for LPA, whereas it is only 28% for AC2A and 15% for ETAIIM. Error detection and correction method is discussed below. A first conclusion from these results is that AC2A is a fast adder with a good balance between accuracy and area.

Figure 1.39 [39] gives comparative results for AC2A adder, using the metrics $\texttt{ACC}_{amp}$ and $\texttt{ACC}_{inf}$, varying voltage from $0.6V$ to $1.0V$. In these graphs, AC2A is refered to as $ACA\ adder$ (not to be confused with Almost-Correct Adder [34]), and $Lu's\ adder$ refers to LPA [33], both presented in Section 1.4.1.1. AC2A shows an interesting resistance to VOS. Indeed, for $\texttt{ACC}_{amp}$ metric, only ETAI achieves a better resistance, but it has extremely bad results with $\texttt{ACC}_{inf}$ metric, whereas AC2A beats every other tested operator, closely followed by ETAIIM. What can be concluded is that AC2A has a shorter critical path than the tested adders, and ensures a good accuracy in terms of error amplitude as well as a low Bit Error Rate (BER).

As mentioned before, AC2A calculation errors occur when at least one of the sub-adder should have taken an input carry. Knowing this, detecting an error can be performed with a very little overhead. Correction can then be performed by transmitting the lost carry or carries to the concerned sub-adders. Just as VLSA (see Section 1.4.1.1), the entire error correction could be performed with additive cycles, but with only a small area overhead since most of the design is re-used for correction. However, unlike VLSA, AC2A proposes a configurable accuracy. Indeed, the periodical structure of the error correction system allows several levels of correction, so that an erroneous result can be partially corrected. For this, a pipelined correction design is proposed, following the principle showed in Table 1.15. During the first cycle, the

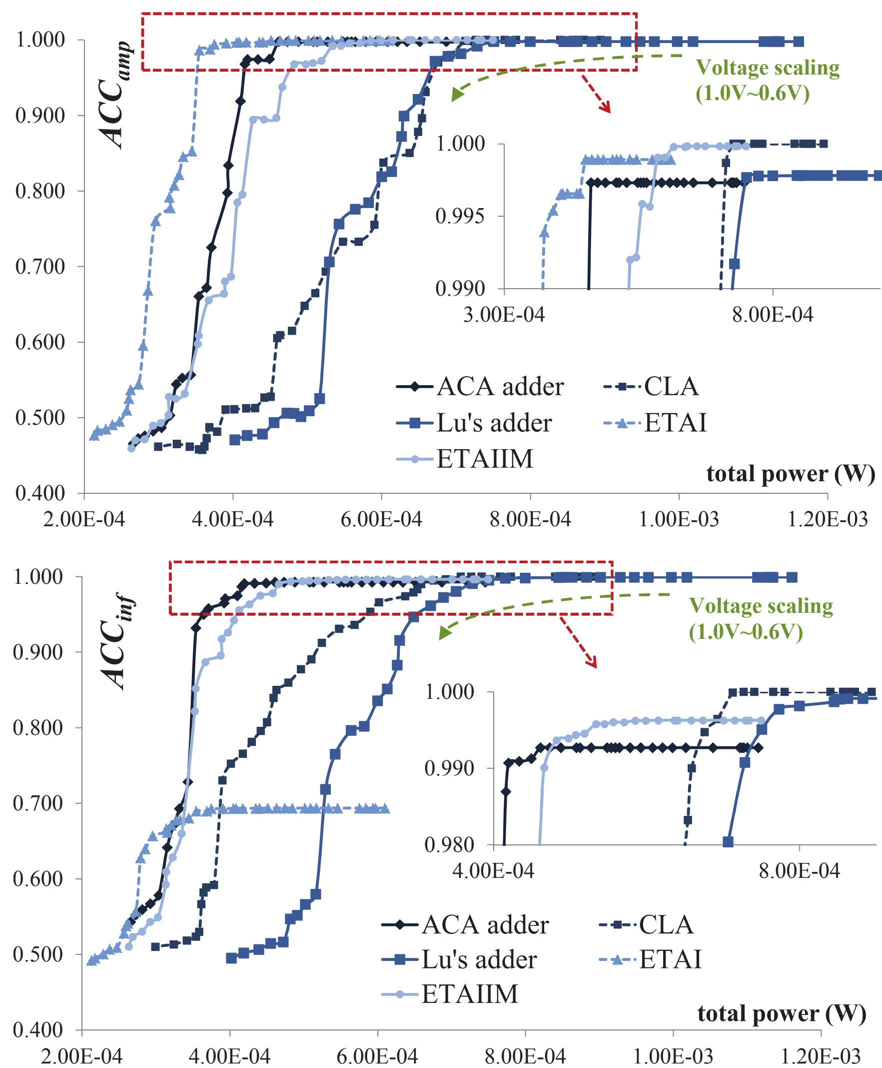
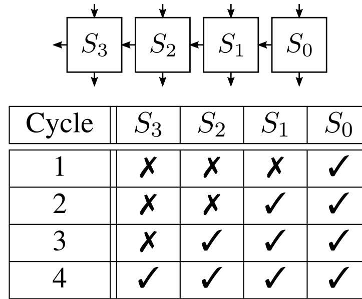


Figure 1.39 – Accuracy vs power for AC2A and other approximate adders under VOS

approximate calculation is performed. Then, each following cycle is dedicated to the successive correction of each sub-adder, from the second one counting from the LSB. By power-gating each partial error correction systems, different levels of accuracy can then be targeted. By design, calculation and correction cycles have the same theoretical maximum delay.



Cycle	S_3	S_2	S_1	S_0
1	✗	✗	✗	✓
2	✗	✗	✓	✓
3	✗	✓	✓	✓
4	✓	✓	✓	✓

Table 1.15 – Error correction cycles in a 4-block AC2A – Checkmarks means the output of block S_i is accurate after j cycles, crosses that it is inaccurate.

An example of the previously described structure is developed in [39] taking a 32-bit AC2A composed of 4 8-bit sub-adders. With such a structure, four modes can be applied:

- mode-1: no power-gating, the whole pipeline is active, and the produced result is exact,
- mode-2: only stage 4 is power-gated, only the most significant bits sub-adder is not corrected,
- mode-3: stage 3 and 4 are power-gated, only one sub-adder is corrected, which is the second one from the LSB,
- mode-4: stage 2, 3 and 4 are power-gated, and so only the approximate calculation is performed with no error correction.

Comparisons of the accuracy reached by these modes using $\texttt{ACC}_{amp}$ and $\texttt{ACC}_{inf}$ metrics, as well as power and power reduction compared to a conventional pipelined adder are give in Table 1.16. The conventional pipelined adder refers to an exact adder where the calculation is regularly performed by pipelining the calculation using its sub-adders. Therefore, the 32-bit conventional pipelined adder used as a reference also has four pipeline stages, each stage performing one-fourth of the total calculation, contrary to the proposed pipeline where the whole approximate calculation is performed on the first pipeline step. In mode-1, the conventional pipelined adder is more energy efficient than AC2A, but when using approximate modes 2 to 4, energy saving raises from 12.4 to 51.6%, with a relatively small loss of accuracy. Accuracy results of the 32-bit pipelined AC2A on SPEC 2006 benchmarks [41] detailed in [39], present generally good accuracy for every mode, even using mode-4. In this mode, $\texttt{ACC}_{amp}$ is superior to 0.99 for every test and above 0.95 for $\texttt{ACC}_{inf}$.

In order to show the advantage of configurability in terms of power, the same benches are run with a dynamic configuration of the accuracy. Even if it is not specified in [39], it can be assumed that the operating modes were chosen thanks to a succession of simulations and configuration optimizations and so there is no auto-control system. The final configurations insure

Configuration	$\mathrm{ACC_{amp}}$ (max)	$\mathrm{ACC_{inf}}$ (max)	Total power (mW)	Power reduction
mode-1	1.000	1.000	5.962	-11.5%
mode-2	0.998	0.960	4.683	12.4%
mode-3	0.991	0.925	3.691	31.0%
mode-4	0.983	0.900	2.588	51.6%

Table 1.16 – Accuracy and power consumption of 4-stage pipelined 32-bit AC2A as a function of the active mode and comparison with conventional pipelined adder

that $0.99 \leq \mathtt{ACC}_{amp} \leq 1.00$ for the first series of tests and $0.95 \leq \mathtt{ACC}_{inf} \leq 1.00$ for the second. Results in [39] show that the proposed 32-bit AC2A leads to an average of 30.0% power savings with the $\mathtt{ACC}_{amp}$ objective (44.5% at best) and 35.8% average power savings with the ACC_{inf} objective (47.1% at best) on SPEC 2006 benchmarks.

As a conclusion, AC2A proposes a pipelined accuracy-configurable adder with good accuracy performance and with potentially important energy savings thanks to partial power gating applied on the error-correction system. However, accuracy configuration must be done offline and so programming effort is increased. In terms of power consumption, AC2A is quite near to ETAI performance, which is intermediately energy-efficient (see Section 1.4.1.2), but with a much better accuracy in terms of information accuracy, meaning the number of correct bits produced. However, AC2A has two main drawbacks:

- error correction is gradually performed from the least significant sub-adders, and
- changing the error correction efficiency at run time implies a modification on the number of cycles for an addition, and variable-latency instructions are generally difficult to handle in an instructions pipeline.

Discarding these drawbacks, AC2A coupled with a good configuration can lead to important energy savings with a relatively low loss of accuracy or information.

1.4.1.4 Gracefully-Degrading Adder

This section presents a quality-configurable approximate adder denoted as Gracefully-Degrading Adder (GDA) [42]. In comparison to AC2A [39] presented in Section 1.4.1.3, GDA is meant to be an approximate adder with a better accuracy, reached with less effort. Indeed, as showed in Table 1.15, each AC2A additive correction cycle leads to better correction, the first correction cycle correcting LSB bringing much less accuracy improvement than the last one, correcting MSB. Both GDA and AC2A can reach the same maximum quality with the same effort, but the optimized operator is able to reach a very good quality with a dramatically reduced amount of effort when compared to the original one.

The proposed GDA is based on a structure very similar to ETAIV [37] described in Section 1.4.1.2. Indeed, the adder is divided into smaller chained adders whose input can be

switched whether to the upstream sub-adder output carry or to the output of a carry-in prediction block thanks to a multiplexer. For an n-bit GDA divided in four $n/4$-bit sub-adders, with $X = (X_3, X_2, X_1, X_0)$ and $Y = (Y_3, Y_2, Y_1, Y_0)$, both are $n/4$-bit subsets of inputs, and $Z = (Z_3, Z_2, Z_1, Z_0)$, $n/4$-bit subsets of outputs, the structure of the corresponding GDA is given in Figure 1.40. The configuration of GDA consists in setting up the multiplexers. If the

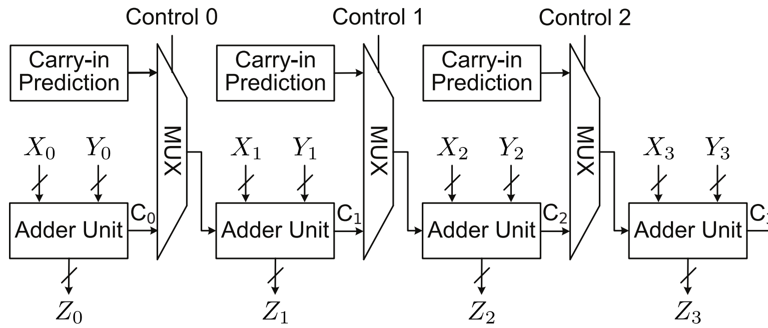


Figure 1.40 – Structure of proposed n-bit GDA composed of 4 $n/4$-bit sub-adders

upstream sub-adder output carry is chosen, then a bigger sub-adder is virtually used, causing longer delay in return for a higher accuracy. In order to obtain a faster operator, it is then better to choose carry-in prediction blocks as sub-adders input. To be prevented from an important loss of accuracy, these blocks need to predict the input carry efficiently and with the shortest possible delay, and imperatively strictly inferior to the adder unit.

Proposed carry-in prediction is based on a hierarchical scheme. Indeed, in order to have a fast and accurate prediction, many LSBs have to be considered, more precisely more than the length of the proposed GDA adder units. To achieve that, prediction computation needs to be parallelized. As for ACA (see Section 1.4.1.1), prediction is based on $propagate$ and $generate$ signals p_i and g_i, which define a recursive formula for the calculation of the carry value c_{i+1}:

$$\begin{aligned} p_i &= a_i \oplus b_i, \\ g_i &= a_i \wedge b_i, \\ c_{i+1} &= g_i \vee (p_i \wedge c_i(). \end{aligned} \tag{1.19}$$

By developing the previous equations and by assuming the longest carry chain cannot exceed t, the expression of the carry signal c_i is then

$$c_i = g_{i-1} \vee (p_{i-1} \wedge g_{i-2}) \vee \cdots \vee \left(\prod_{j=i-1}^{i-t+1} p_j \right) \wedge g_{i-t} \vee \left(\prod_{j=i-1}^{i-t} p_j \right) \wedge c_{i-t}. \tag{1.20}$$

Using Equation 1.12 in Section 1.4.1, we can prove that for a 32-bit adder, the probability for the longest carry-chain to exceed 8 is 2.43%. Therefore, assuming taking the 8 preceding bits into account for carry prediction is acceptable, the expression of the carry signal becomes

$$\begin{aligned} c_i &= c'_i \vee \left(\prod_{j=i-1}^{i-4} p_j \right) \wedge c'_{i-4}, \text{ with} \\ c'_i &= g_{i-1} \vee (p_{i-1} \wedge g_{i-2}) \vee \cdots \vee \left(\prod_{j=i-1}^{i-4} p_j \right) \wedge g_{i-4}, \text{ and} \\ c'_{i-4} &= g_{i-5} \vee (p_{i-5} \wedge g_{i-6}) \vee \cdots \vee \left(\prod_{j=i-1}^{i-4} p_j \right) \wedge g_{i-8}. \end{aligned} \tag{1.21}$$

Moreover, c'_{i-4} will propagate to c_i only if

$$\prod_{j=i-1}^{i-4} P_j = 1. \tag{1.22}$$

These equations lead to the hierarchical prediction scheme of Figure 1.41, where two 4-bit groups are watched in parallel before considering the condition showed by Equation 1.22 using AND gates.

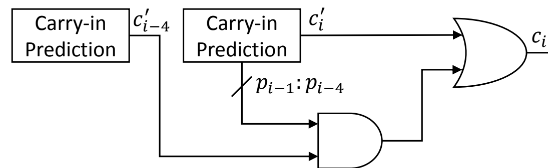


Figure 1.41 – GDA hierarchical prediction scheme

Based on this hierarchical prediction scheme, another level of configurability can be introduced. Indeed, following the scheme proposed by Figure 1.42, the number of preceding bits considered in the prediction can be set by a series of multiplexers, with a step determined by the deepness of carry-in prediction blocks. For instance, the number of carries which can be considered is whether 4, 8, $\ldots$, $4 \times k$, where k is the number of implemented unitary carry-in prediction blocks. This structure gives a high number of possibilities for the granularity of the reconfigurability and on the parallelism of the carry-in prediction.

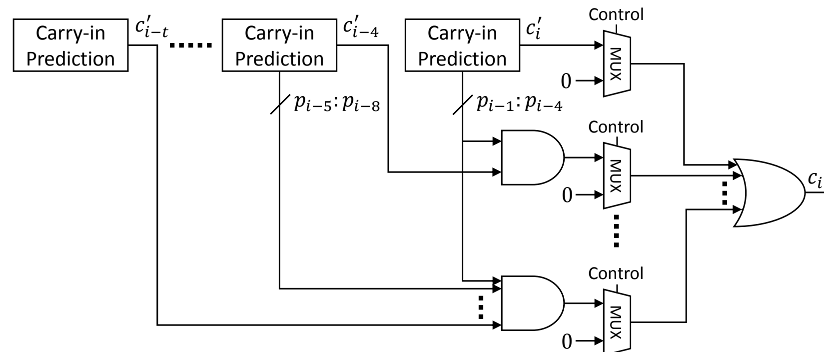


Figure 1.42 – GDA reconfigurable prediction scheme

To conclude about its structure, GDA proposes two levels of configuration:

- by its possibility to select the use of carry-in prediction or exact adding between each sub-adder, and
- by its possibility to select the deepness of carry-in prediction when this mode is active.

By structure, GDA offers fine-grained control for the level of error that can be tolerated at its output, and potentially offers a good prediction of carry despite an important area penalty for control.

In [42], the authors give very complete comparative simulation results of 32-bit GDA compared to exact adders, RCA and CLA, and other approximate adders, Variable Latency Carry Select Adder (VLCSA-1) [43] and LPA [33] described in Section 1.4.1.1, leveraging worst-case-error, error rate and average error. Results are given in Table 1.17. All simulations are based on one million randomly-generated inputs, so they have to be taken with caution, especially worst-case error. All adders are static, meaning AC2A and GDA are not generated using their reconfigurable version, the configuration is set before implementation. M_A corresponds to the mode AC2A is operating such as described in Table 1.16. For the test and for each mode, AC2A was implemented in a non-pipelined version in order to get fair area and delay measurements. M_B and M_C are the parameters for GDA. M_B indexes the number of bits for each sub-adder (respectively 4, 8, 12 or 16) and M_C the number of prediction bits (respectively 4, 8, 12 or 16). From this point and for the rest of this section, AC2A_i will denote AC2A on mode $M_A = i$, and $\text{GDA}_{(i,j)}$ will denote GDA on mode $(M_B, M_C) = (i, j)$. When comparing all exact adders, RCA, CLA, AC2A_4 and $\text{GDA}_{(4,4)}$, the two last exact adders achieve a better delay than the classical others, but with an important area and power overhead. For instance, $\text{GDA}_{(4,4)}$ is 42.3% faster than RCA, but with 44.5% area overhead and twice the operating power. Compared to AC2A_4, $\text{GDA}_{(4,4)}$ occupies 66.45% more area and consumes 7.97% more power, but is 5.93% faster. It shows than nor AC2A_4 neither $\text{GDA}_{(4,4)}$ are only efficient in terms of delay in their exact version.

Accuracy-configurable implementation of AC2A and GDA are compared in Table 1.18, showing the area for each reconfigurable operator, as well as the power and delay corresponding to all their operating modes. Delay^* corresponds to the delay when applying voltage scaling for each operator and each mode in order to have the same power consumption as the highest, occurring for $\text{GDA}_{(1,4)}$. In their reconfigurable versions, it can be observed that GDA costs 19.43% less area than AC2A. In accurate mode, GDA is only 0.90% slower than AC2A because of the higher number of multiplexers in the critical path. Results from Table 1.17 and Table 1.18 highlight the benefits of GDA compared to AC2A for an equivalent power (Delay^* rows in Table 1.18) and for each of the three metrics considered. Figure 1.43 shows the benefits of GDA, selecting the optimal accuracy configuration for each of these metrics:

- AC2A worst-case error is compared to $\text{GDA}_{(1,1)}$, $\text{GDA}_{(2,1)}$, $\text{GDA}_{(3,1)}$, $\text{GDA}_{(4,1)}$ and $\text{GDA}_{(4,4)}$.
- AC2A error rate is compared to $\text{GDA}_{(1,1)}$, $\text{GDA}_{(1,2)}$, $\text{GDA}_{(1,3)}$, $\text{GDA}_{(1,4)}$ and $\text{GDA}_{(4,4)}$.
- AC2A average error is compared to $\text{GDA}_{(1,1)}$, $\text{GDA}_{(2,2)}$, $\text{GDA}_{(3,3)}$ and $\text{GDA}_{(4,4)}$.

In conclusion, the slight delay overhead in GDA structure allows significative reductions of the error, contrary to AC2A. However, it has to be kept in mind that each of the curves of Figure 1.43 does not represent the entirety of GDA but only the optimal points. This means for instance that the ideal configuration for optimizing worst-case error is not the same for error

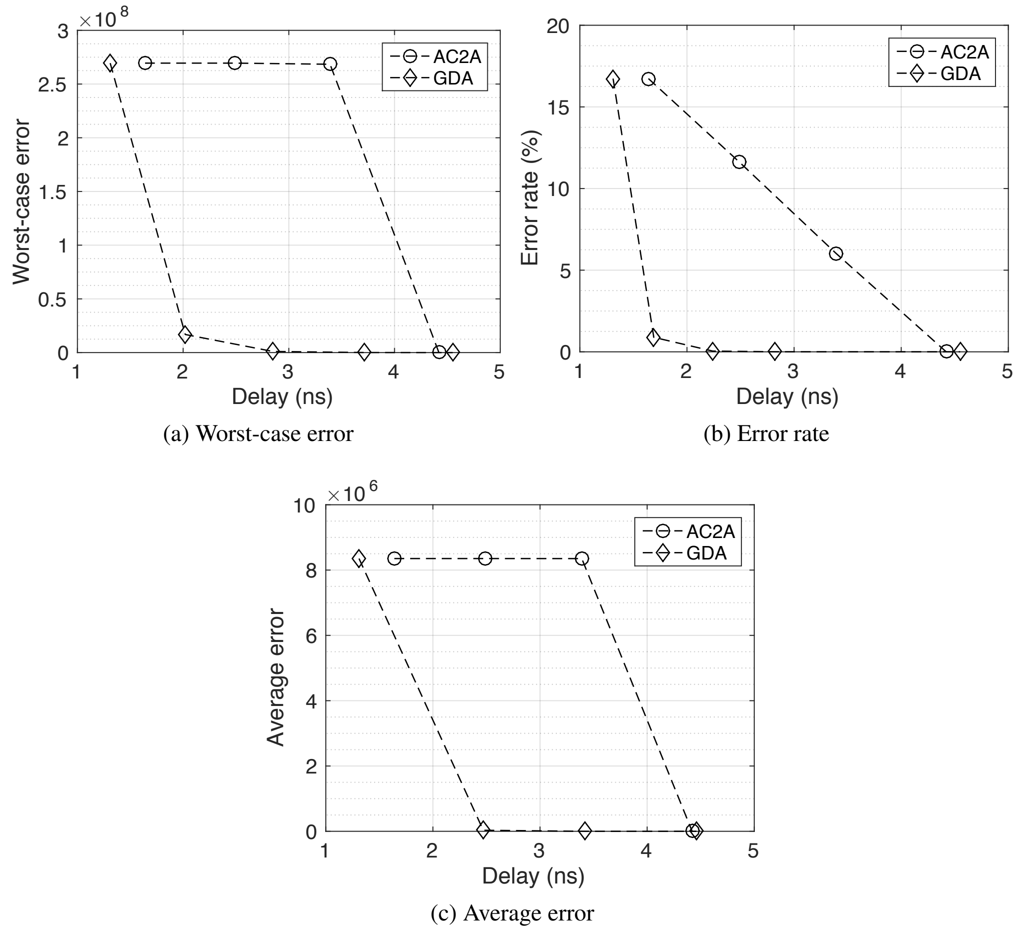
	RCA	**CLA**	**VLCSA-1**	**LPA**
Area (μm^2)	1184	1736	2686	2321
Power (W)	2.407E-05	3.106E-05	5.054E-05	3.668E-05
Delay (ns)	7.88	5.40	1.24	0.83
WCE	0	0	269,488,128	2,215,641,344
ER	0	0	16.6715%	32.2528%
AE	0	0	7,906,556	45,247,362
AC2A				
$\mathbf{M_A}$	1	2	3	4
Area (μm^2)	2200	2416	2632	2848
Power (W)	4.209E-05	4.692E-05	4.977E-05	5.259E-05
Delay (ns)	1.44	1.78	2.11	2.44
WCE	269,488,128	269,484,032	268,435,456	0
ER	16.7344%	11.6149%	6.0388%	0
AE	8,352,852	8,352,848	8,351,497	0
GDA				
$(\mathbf{M_B}, \mathbf{M_C})$	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$
Area (μm^2)	1501	1597	1677	1741
Power (W)	3.940E-05	4.383E-05	4.623E-05	4.793E-05
Delay (ns)	1.44	1.78	2.11	2.44
WCE	269,488,128	268,439,552	268,435,456	268,435,456
ER	16.7344%	0.8873%	0.0453%	0.0027%
AE	8,352,852	509,152	33,690	2,215
$(\mathbf{M_B}, \mathbf{M_C})$	$(2,1)$	$(2,2)$	$(2,3)$	$(2,4)$
Area (μm^2)	1577	1625	1689	1753
Power (W)	4.329E-05	4.567E-05	4.773E-05	4.952E-05
Delay (ns)	2.19	2.48	2.86	3.14
WCE	16,843,008	16,777,216	16,777,216	16,777,216
ER	8.8746%	0.3761%	0.0232%	0.0018%
AE	523,985	31,614	1,954	67
$(\mathbf{M_B}, \mathbf{M_C})$	$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$
Area (μm^2)	1596	1628	1660	1708
Power (W)	4.437E-05	4.597E-05	4.703E-05	4.840E-05
Delay (ns)	2.93	3.22	3.51	3.85
WCE	1,048,832	1,048,576	1,048,576	1,048,576
ER	5.9984%	0.1922%	0.0122%	0.0016%
AE	32,886	2,015	128	17
$(\mathbf{M_B}, \mathbf{M_C})$	$(4,1)$	$(4,2)$	$(4,3)$	$(4,4)$
Area (μm^2)	1615	1631	1663	1711
Power (W)	4.552E-05	4.631E-05	4.735E-05	4.871E-05
Delay (ns)	3.68	3.97	4.26	4.55
WCE	65,536	65,536	65,536	0
ER	3.0964%	0.1884%	0.0116%	0
AE	32,029	123	8	0

Table 1.17 – Comparison between 32-bit GDAs and exact and approximate static adders [42]

AC2A				
Area (μm^2)	3119			
$\mathbf{M_A}$	1	2	3	4
Power (W)	4.261E-05	4.882E-05	5.289E-05	5.696E-05
Delay (ns)	2.15	2.90	3.67	4.45
Delay* (ns)	1.64	2.49	3.39	4.42

GDA				
Area (μm^2)	2513			
$(\mathbf{M_B}, \mathbf{M_C})$	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$
Power (W)	4.679E-05	4.777E-05	5.327E-05	5.733E-05
Delay (ns)	1.58	2.01	2.41	2.82
Delay* (ns)	1.31	1.69	2.24	2.82
$(\mathbf{M_B}, \mathbf{M_C})$	$(2,1)$	$(2,2)$	$(2,3)$	$(2,4)$
Power (W)	4.590E-05	4.900E-05	5.065E-05	5.245E-05
Delay (ns)	2.49	2.87	2.96	3.11
Delay* (ns)	2.02	2.47	2.63	2.85
$(\mathbf{M_B}, \mathbf{M_C})$	$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$
Power (W)	4.742E-05	4.953E-05	5.031E-05	5.120E-05
Delay (ns)	3.41	3.78	3.88	4.03
Delay* (ns)	2.85	3.29	3.42	3.61
$(\mathbf{M_B}, \mathbf{M_C})$	$(4,1)$	$(4,2)$	$(4,3)$	$(4,4)$
Power (W)	4.879E-05	4.981E-05	5.057E-05	5.149E-05
Delay (ns)	4.33	4.70	4.80	4.95
Delay* (ns)	3.71	4.11	4.25	4.46

Table 1.18 – Accuracy-configurable implementation of AC2A and GDA [42]



(a) Worst-case error

(b) Error rate

(c) Average error

Figure 1.43 – Error vs delay for an identical power consumption for GDA and AC2A [42]

rate or average error. Therefore, there is no optimal configuration for GDA considering all metrics at the same time, and so GDA accuracy has to be configured knowing the prior metric to take into consideration.

1.4.1.5 Addition Using Approximate Full-Adder Logic

All the approximate adders previously described leverage modifications in addition function, always cutting off in different ways carry propagation, except for ETAI (see Section 1.4.1.2). In this section, the adders are built considering modifications in the Full Adder (FA) function. As developed in Section 1.3.3, FA function is the basic cell the most used in binary addition since it produces a 1-bit addition.

Several possible modifications of the FA celle are proposed in [44], and more particularly on the Mirror Adder (MA) circuit implementing FA logic (see Figure 1.44a). Indeed, being able to remove transistors in a FA cell modifying as little as possible its binary logic function

can potentially induce high benefits in terms of speed, area, and energy savings, because of the omnipresence of this cell in numerous adder designs.

First, transistors are removed one by one from the conventional MA to find a configuration where all sets of input x, y and c_{in} give in as many cases as possible the good set of outputs z and c_{out}. The best result according to the author was obtained removing 8 transistors, which leads to the resulting circuit of Figure 1.44b. Then, the final Approximate Mirror Adders (AMAs) are designed following a series of observations about FA truth table:

- The Approximate Mirror Adder type 1 (AMA1) is based on the observation that $z = \overline{c_{out}}$ for 6 cases out of 8. Therefore, the z calculation part is suppressed and z is set to $\overline{c_{out}}$ using a buffer in order to limit capacitance for latency and/or power efficiency purpose. AMA1 transistor view is showed on Figure 1.44c. A 44.5% area gain is observed by AMA1 towards traditional MA.

- The Approximate Mirror Adder type 2 (AMA2) is based on the observation that for 6 cases out of 8, $cout = x$ is verified (or $c_{out} = y$ by symmetry). Therefore, an inverter is used on x to calculate c_{out} from simplified MA of Figure 1.44b, producing the transistor view in Figure 1.44d. A 41.2% area gain is observed by AMA2 towards traditional MA, which is nearly as good as AMA1 thanks to its shorter critical path.

- The Approximate Mirror Adder type 3 (AMA3) is obtained from AMA2. In AMA2, there are 3 errors for z in its truth table. The idea of AMA3 is to reduce dependency between c_{in} and z. A good way to do it is to force $z = y$. Structuring the adder this way also insures c_{out} to be correct when z is correct. AMA3 is 66.7% smaller than traditional MA, which is much better than AMA1 and AMA2, but it comes at a cost of a much higher approximation.

Truth tables of AMA1, AMA2 and AMA3 are given in Table 1.19. They show that AMA1 globally generates less errors than AMA2, and AMA2 less than AMA3. Obviously, the case when each of these errors occurs is very important for the global accuracy. For instance, it is potentially more costly in terms of error to accidentally generate a carry on c_{out} than not to propagate a carry that should have been propagated or than to give the wrong result to z, since the error is possibly propagated to a higher significance output bit.

Because of the potential high amplitude errors that can be generated by chaining AMA, AMA-based approximate adders cannot be exclusively used in approximate adders designs to reach an acceptable output minimal accuracy. That is why the author only presents designs where only the LSB part is made of AMAs. In [44], the MSB part is composed of conventional exact FA cell instances forming a RCA. The adders produced this way are denoted as IMPrecise Adder for low-power Approximate CompuTing (IMPACT). However, in general, it is possible to use any of the accurate adder structures presented in Section 1.3.3. The designs of AMAs being fixed, the only way to modify accuracy is to modify the number of approximated LSBs. In [44], image processing tests are performed with different approximated LSB lengths and compared to FxP truncation method in terms of Peak Signal-to-Noise Ratio (PSNR), power and area savings towards exact method. Results for Discrete Cosine Transform (DCT) and

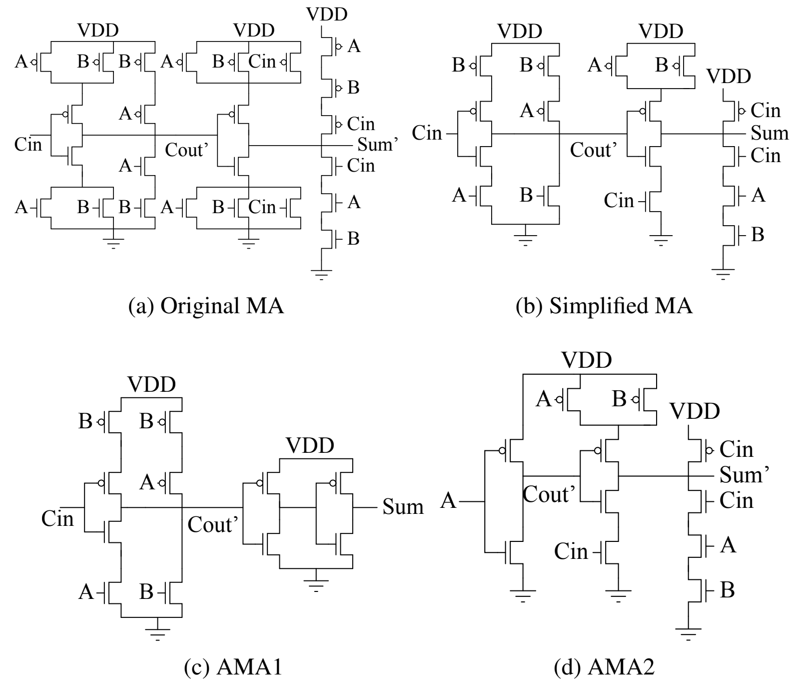


Figure 1.44 – Original and approximate MA transistor view – A and B inputs refer to x and y in the notations of this document, while Sum output refers to z.

Inputs			**Acc. outputs**		**Approximate outputs**					
x	y	c_{in}	z	c_{out}	z_1	c_{out_1}	z_2	c_{out_2}	z_3	c_{out_3}
0	0	0	0	0	1	0	0	0	0	0
0	0	1	1	0	1	0	1	0	0	0
0	1	0	1	0	0	1	0	0	1	0
0	1	1	0	1	0	1	1	0	1	0
1	0	0	1	0	1	0	0	1	0	1
1	0	1	0	1	0	1	0	1	0	1
1	1	0	0	1	0	1	0	1	1	1
1	1	1	1	1	0	1	1	1	1	1

Table 1.19 – Truth tables of accurate and approximate MA cells – Shaded cells indicate design logic errors.

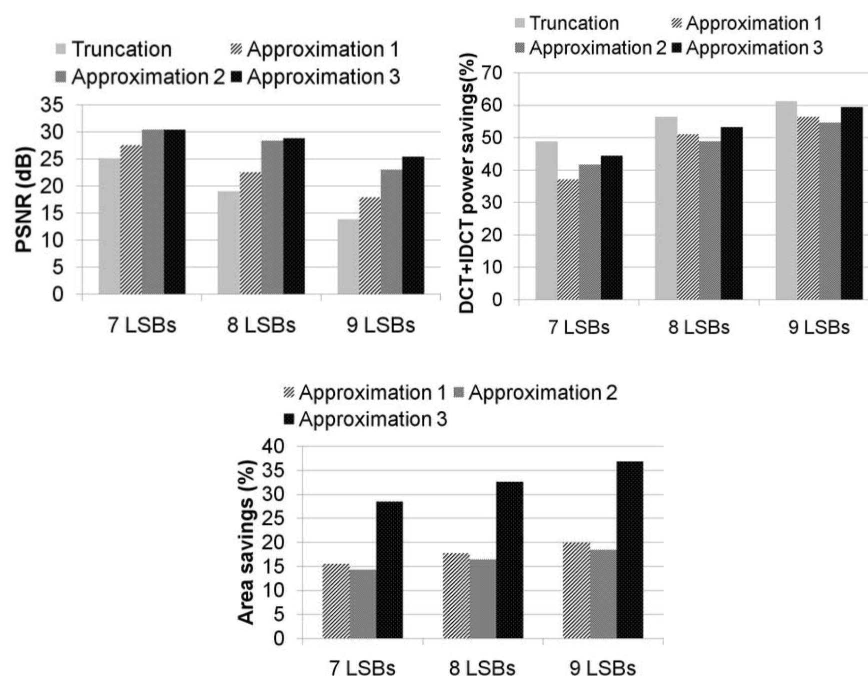


Figure 1.45 – DCT/IDCT test results for IMPACT – The savings are relative to a 20-bit accurate RCA

Inverse Discrete Cosine Transform (IDCT) for 7, 8 and 9 approximated LSBs are given in Figure 1.45, where a DCT-8/IDCT-8 process is applied to 12,288 vectors from the classical image processing image Lena. This process is composed of 8 additions for the computation of each output pixel, all operations are performed on 20 bits. The circuit voltages are set to a minimum in such a way the only allowed errors are functional and not due to VOS. More information about these operating voltages can be seen in Table III of [44]. It can be noticed that operating voltages for all AMAs and truncated adders are very similar so the results are quite fair.

Results of Figure 1.45 show that AMA3 is the most accurate in terms of PSNR, followed by AMA2, AMA1 and finally truncation. Truncation being the worse is not surprising since it virtually sets all LSBs to 0, but the order of accuracy using the three different AMAs would be expected to be the opposite since 1 corresponds to the slightest approximation and 3 to the largest one. However, no other accuracy metric is used so the nature of the performed error is unknown, though we could suspect the AMAss to perform a *salt-and-pepper* noise because of the nature of their design. Moreover, using 20-bit adders on an 8-bit image brings question about their legitimacy. In terms of power, truncation is obviously more efficient than AMA-based adders, but IMPACT shows a good power efficiency, especially for a high number of approximated LSBs. The most power efficient is AMA3, followed by AMA1 or AMA2 depending on the situation. For area results, truncation does not appear. A good estimation of

truncation area savings can be performed considering that for x truncated LSBs on a 20-bit adder, the benefits in terms of area are $\frac{x}{20} \times 100$ in percents. Therefore, we can assume area savings for 7, 8 and 9 LSBs truncation are respectively 35%, 40% and 45%, which is about 25% as high as AMA3, partially explaining energy savings results. However, results show that area savings are twice as high for AMA3 than for AMA2 and AMA1, which are quite similar. Further results implementing MPEG video compression are given in [44] and tend to confirm the results obtained for DCT/IDCT experiment.

As a conclusion, IMPACT are power-efficient approximate adders, but their accuracy compared to truncation still needs to be tested on legitimate experimental settings, which is done in Chapter 4.

1.4.1.6 Approximate Adders by Probabilistic Pruning of Existing Designs

In previous sections, all proposed adders were original designs. However, the existence of a deep literature about integer arithmetic raises the following question: *is it possible to take any adder circuit and automatically transform it to an approximate one?* The answer of this question is addressed in [45]. The idea is to prune iteratively little parts of a design in order to trade accuracy for area, power, and potentially delay. Of course, pruned elements must not be chosen randomly, and so strategies are leveraged.

Two parameters are considered for design pruning – activity and significance. Indeed, power consumption is proportional to activity, therefore it is more interesting for a high-activity-driven gate to be removed uppermost. For arithmetic operators, each output is twice as significant as its direct less significant bit, and so an error occurring at its position has twice as much impact on the output error as an error on this neighbor. Assuming this, every transistor, gate, or group of gates can be assigned a cost function which is the value of its activity multiplied by its significance. This way, they can be sorted in an increasing order to generate a priority list for pruning. The cost functions can also be modified considering all operator outputs have the same weigth. However, we will only get interested in weighted output since this is the most common situation in signal processing.

For automated design, an error target must be given as a parameter. In [45], two error parameters are allowed in the framework:

- $\text{Error Rate} = \frac{\text{Number of erroneous computations}}{\text{Total number of computations}}$.
- $\text{Relative Error Magnitude} = \frac{1}{\nu}\sum_{k=1}^{\nu}\frac{|z_k - z'_k|}{x_k}$, where ν is the number of possible sets of inputs, zk the accurate output for a given input set k and z'_k the output for the same input case considering the pruned design.

Once the error nature chosen and the target defined, probabilistic pruning optimization process follows the flowchart given in Figure 1.46.

In [45], the method is applied to parallel-prefix adders, which is convenient as they are defined by a grid of base cells (see Section 1.3.3), and thus probabilistic pruning is easy to be applied considering each of these parallel-prefix base cells to be the unitary block for the method. Figure 1.47 shows the results for a pruned 16-bit Kogge-Stone adder with a -20 dB relative error accuracy goal. For this adder, activity increases with the considered level of the

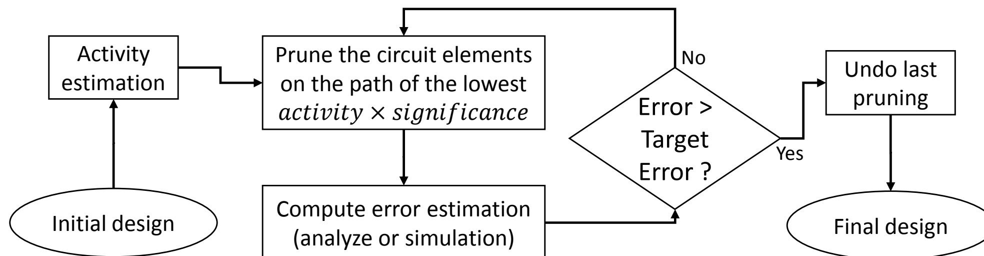


Figure 1.46 – Flowchart for probabilistic pruning optimization process [45]

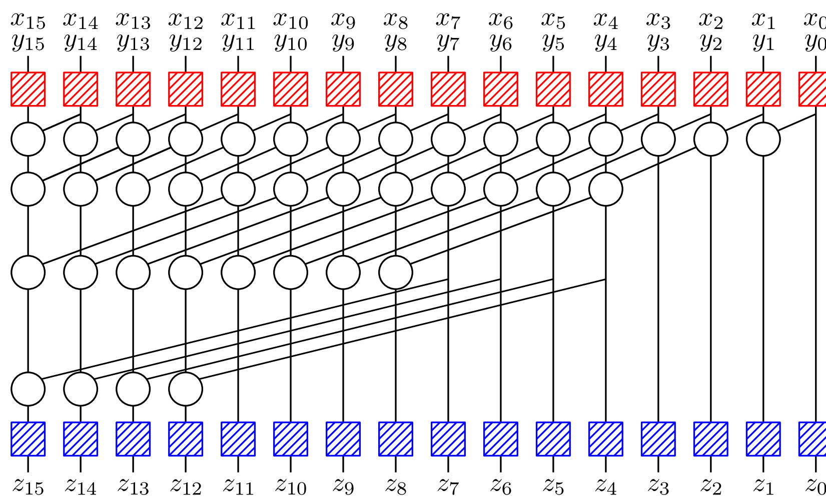


Figure 1.47 – 16-bit Weighted-Pruned Kogge Stone Adder (WPKSA) – The original KSA is depicted in Figure 1.14 in Section 1.3.3.

parallel-prefix graph, while the significance logically increases from LSB to MSB on each level. Thus, cells are pruned from the lower right corner to the upper left corner. It can be seen that to reach the -20 dB relative error, 10 cells were pruned on a total of 49, which represents more than 20% area saving.

Probabilistic pruning method on several classical adders such as RCA, CSA, KSA, HCA, LFA is applied in [45]. All error estimations were obtained by functional simulation, whose number of sets of inputs is unknown. Analytical method was used to compute activity estimation, assuming each input has a 0.5 activity. Globally, the best benefits in the Energy-Delay-Area (EDA) product are $2\times$-$7.5\times$ compared to the original adder, with a relative error of respectively 10% and $10^{-6}\%$, these best results being obtained for KSA and HCA. Results about these adders are given in Figure 1.48. The most important gains are obtained on delay, thanks to shorter critical paths. Despite a lack of detailed results, we can assume by interpolating the points of the result curves that quite an important benefit can be observed even when trading a few amount of accuracy. As an example, for a 10^{-2} relative error, a factor of more than $3\times$ is reached for the relative EDA.

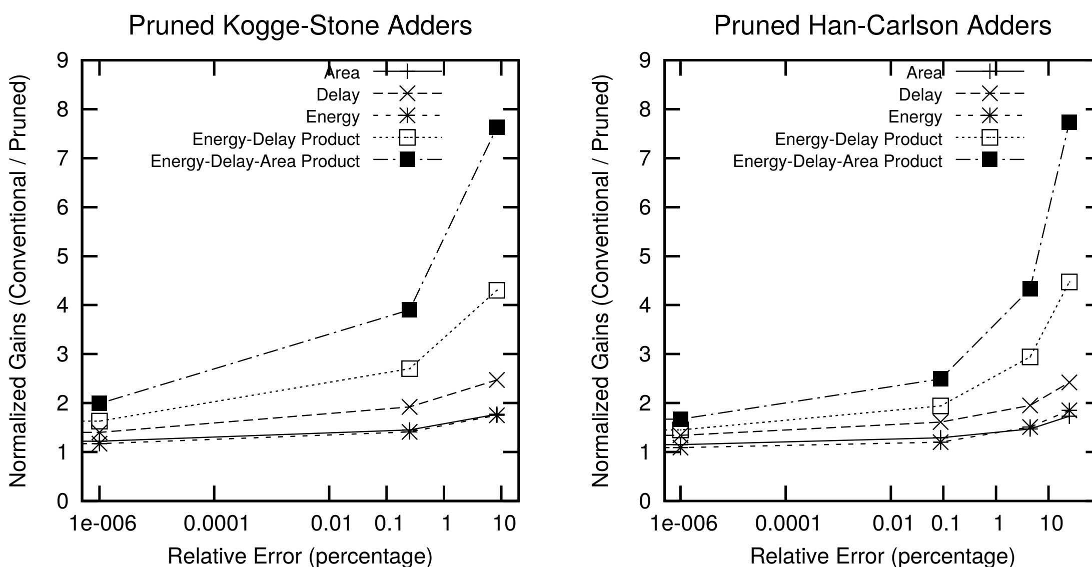


Figure 1.48 – Probabilistic pruning results for Kogge-Stone and Han-Carlson adders [45]

The main issue with this method is the use of mean relative error as a target, which does not prevent the operator from performing high amplitude errors. As an example, Weighted-Pruned Kogge-Stone Adder (WPKSA) presented on Figure 1.47 can perform an error on its MSB for a certain number of sets of inputs, causing an extremely high error, which cannot be tolerated depending on the considered application.

The list of approximate adders presented in Section 1.4.1 is far from exhaustive, many others exist in literature. However, the ones presented here are representative of the general trend of approximate adders. Most are derived from classical adders, with different ways to cut carry chains and accelerate carry propagation. Some designs come with error detection and correction circuits, leading to another kind of approximate adders, the configurable adders, which can take different accuracy targets at run time. Finally, an interesting automated method for approximate circuit generation stands out from the crowd [45], applied on adders in the original paper but which can be applied to any signal processing circuit. Next section presents a subset of the literature for approximate multipliers.

1.4.2 Approximate Integer Multiplication

In this section, a subset of literature dealing with approximate multipliers is developed. As seen previously in Section 1.3.4, the most important part of the multiplication structure is the reduction of the summand grid, performed with adders (generally CSA). Therefore, a high quantity of different multipliers can be derived using the approximate adders such as the ones presented in previous section. In this document, we will try to highlight more original multipliers leveraging more creative ideas from which they can benefit.

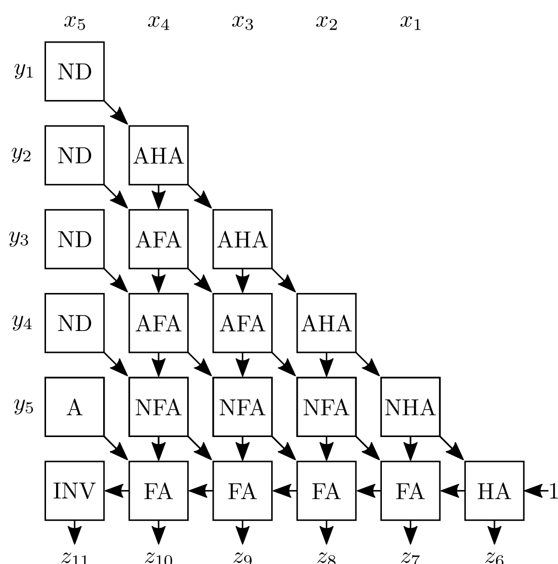


Figure 1.49 – Two's-complement signed 6-bit AAMI structure

1.4.2.1 Approximate Array Multipliers

In this section, three versions of approximate array multipliers are presented. As a reminder, accurate array multipliers are presented in Section 1.3.4. A 6-bit two's-complement signed array multiplier is depicted in Figure 1.20. Accurate array multipliers are not the fastest neither the smallest multipliers. However, their periodic structure has a compact hardware layout, thanks to short wiring, and allows for efficient pipelining. This advantage makes array multiplier one of the most used in embedded System on Chip (SoC). The three Approximate Array Multiplier (AAM) of this section are Fixed-Width Multipliers (FWMs), meaning that if their inputs are of with n, their output is also of width n, instead of $2n$, as it should be to be perfectly accurate. Only the n MSBs are kept. In practice, most multipliers are FWMs as data width is generally constant in processing units.

The first AAM will be denoted by Approximate Array Multiplier I (AAMI) and was proposed in [46]. As the two others described below, the idea is to prune the least significant cells, i.e. the ones responsible for the computation of the non-kept output LSBs. The resulting two's-complement signed operator is given in Figure 1.49. More than the half of the base cells were pruned, and the diagonal AFAs were changed into AHAs since they have one less input in AAMI version. The mathematical calculation of the error bias is given in [46]. The author also showed that the bias and the variance of the error are linear with the size of the AAMI.

The basic pruning proposed by AAMI was then improved in [47] with Approximate Array Multiplier II (AAMII) presented below and in [48] with Approximate Array Multiplier III (AAMIII). AAMII adds to AAMI a correction circuit on the diagonal to reduce the bias of error. This correction circuit is made with very few gates, since it is composed of n very simple cells on the diagonal. The first one is an AND gate, followed by $n-2$ AAO cells composed of two

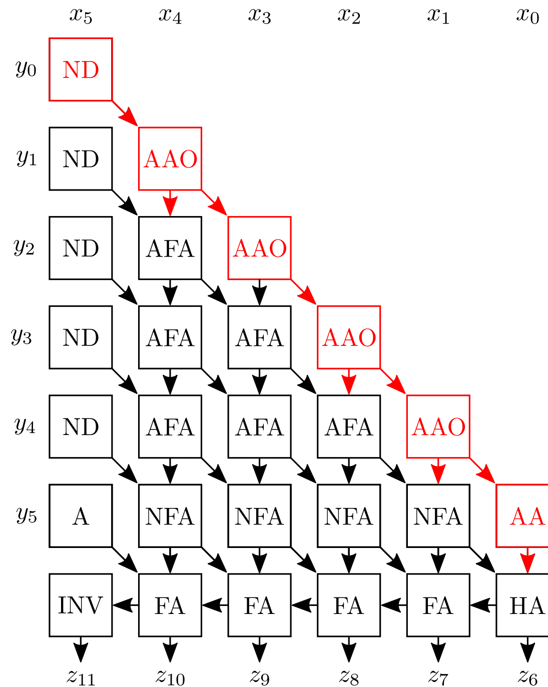


Figure 1.50 – Two's-complement signed 6-bit AAMII structure

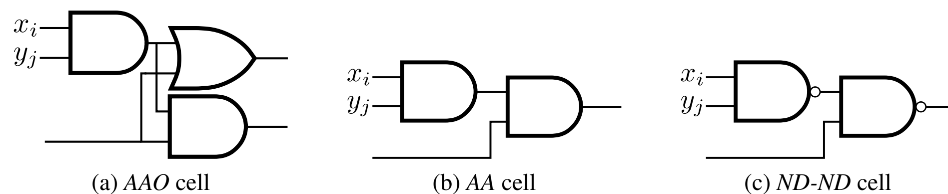


Figure 1.51 – *AAO*, *AA* and *ND-ND* cells

AND gates and an OR gate disposed as on Figure 1.51a, and the last one is a combination of two AND gates as on Figure 1.51b. In the original paper, only an unsigned operator is designed. However, as mentioned in Section 1.3.4, two's-complement signed multiplier can be obtained by simply negating the MSBs of the $n - 1$ first partial products of the summand grid and negating the whole last partial product except for the MSB. The resulting two's complement signed multiplier is depicted in Figure 1.50.

Table 1.20 [47] gives accuracy results comparing AAMI and AAMII using its maximal absolute error $\mathrm{AE}_{\mathrm{max}}$, and the AP for a given MAA, this metric being explained in Section 1.4.1.2. These results show that AAMII has a much lower maximal error than AAMI thanks to bias reduction circuit, and the maximal error is increasing slower when the size of the multiplier increases. AP is also much higher for AAMII, and the difference between both multiplier designs seems to increase with MAA. As an example, for a 16-bit multiplication,

Multiplier	Error metric	$\mathbf{n = 4}$	$\mathbf{n = 8}$	$\mathbf{n = 12}$	$\mathbf{n = 16}$
AAMI	AE_{max}	32	1,536	40,960	917,540
AAMII		16	512	8,196	196,608
AAMI	AP for MAA $= 99\%$	26.2%	50.8%	90.1%	98.9%
AAMII		94.1%	79.3%	94.8%	99.4%
AAMI	AP for MAA $= 99.99\%$	26.2%	39.1%	33.0%	35.2%
AAMII		94.1%	74.6%	62.1%	77.9%

Table 1.20 – Accuracy comparison of AAMI and AAMII [47]

77.9% of outputs will be less than 0.01% far from the expected results, against only 35.2% for AAMI.

To conclude about AAMII, this second version achieves much better performance in terms of maximal error as well as in term of acceptance probability, with a very small area and delay overhead. AAMII also has an area which is nearly half of the classical non-fixed-width array multiplier, but slightly more important than AAMI.

A third amelioration of the signed version of the AAM, referred as AAMIII, is proposed in [48]. As AAMII, the idea is to lower the bias induced by the operator truncation, but more effectively. To reach an optimal efficiency, a method to make a fixed-width array multiplier with reduced maximum error, average error and error variance with no overhead for the correction circuit compared to AAMII is presented. The minimization of the bias is exclusively performed by feeding the array multiplier's diagonal with AND and $NAND$ gates (instead of OR gates in AAMII) and adjusting using a constant input on the last FA line. In order to find the most effective configuration, an exhaustive search of the bias is first performed.

For a given set of inputs $(x_i, y_j)_{,i,j\in[\![0,n-1]\!]}$ then the best error correction term towards AAMI structure C_t can be written as [48]

$$C_t = \sum_{i=1}^{n-2} \langle a_{n-i-1} b_i \rangle^{q_{n-i-1}} + \lfloor K \rceil \tag{1.23}$$

with

$$\langle T \rangle^{q_k} = \begin{cases} T, & \text{if } q_k = 0 \\ \overline{T}, & \text{if } q_k = 1 \end{cases} \tag{1.24}$$

where the value of K depends on the inputs value. The first part of the correction term given by Equation 1.23 can be achieved by inserting simple AND and $NAND$ gates on the AAMI diagonal. When these gates are set, then the only way to affect multiplication result is to set the input bit on the last FA line to 0 or 1. When this value is set to a given value $p \in \{0, 1\}$, then, two correction term cases occur depending on the inputs, given by

$$C_t = \begin{cases} \sum_{i=1}^{n-2} \langle a_{n-i-1} b_i \rangle^{q_{n-i-1}} + \lfloor p \rceil, & \text{if } \theta < n \\ \sum_{i=1}^{n-2} \langle a_{n-i-1} b_i \rangle^{q_{n-i-1}} + \lfloor \overline{p} \rceil, & \text{if } \theta = n \end{cases} \tag{1.25}$$

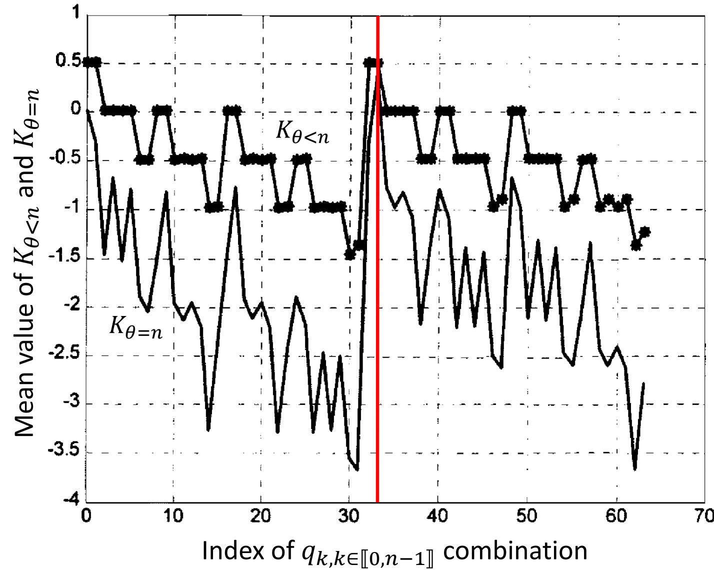


Figure 1.52 – Exhaustive search of $K_{\theta<n}$ and $K_{\theta=n}$ for a 6-bit multiplier – the vertical red line shows the index of the optimal values of $K_{\theta<n}$ and $K_{\theta=n}$

where

$$\theta = \sum_{i=0}^{n-1} \langle a_{n-i-1} b_i \rangle^{q_{n-i-1}} \tag{1.26}$$

Therefore, the values of $q_k, k \in [\![0, n-1]\!]$ need to be fixed so the compensation offset defined by the last line input is as near as possible of $p \in \{0, 1\}$ in the first case of Equation 1.25, and $\overline{p}$ in the second case. To determine this case, an exhaustive search is performed, testing all combinations of $(q_0, \dots, q_{n-1})$ for all input combinations in order to determine the best value of $K_{\theta<n}$ and $K_{\theta=n}$ to minimize the bias for each combination. Figure 1.52 [48] shows the example of an exhaustive search of $K_{\theta<n}$ and $K_{\theta=n}$ for a 6-bit multiplier. Once this exhaustive search performed, then the nearest case from $K_{\theta<n} = 0 \mathrm{or} 1$ and $K_{\theta=n} = \overline{K_{\theta<n}}$ is determined. This case defines which combination of q_k is chosen, and what is the optimal value of the last line input bit. On Figure 1.52, the vertical red line shows the index of the optimal values of $K_{\theta<n}$ and $K_{\theta=n}$.

After exhaustive experimentation, the author analytically shows that for any n-bit multiplier the optimal configuration is always:

- $q_0 = q_{n-1} = 1$ and $\forall k \in [\![1, n-2]\!], q_k = 0$, and
- $\mathrm{Last\ line\ input} = 1$ for n high enough (case-by-case simulation is needed for low values).

Taking that into account, the author gives the structure of a signed 8-bit AAMIII, as showed on Figure 1.53. *FA*, *AFA*, *NFA* and *A* cells are previously described cells. *ND-ND* cell is composed of two *NAND* gates, disposed as on Figure 1.51c.

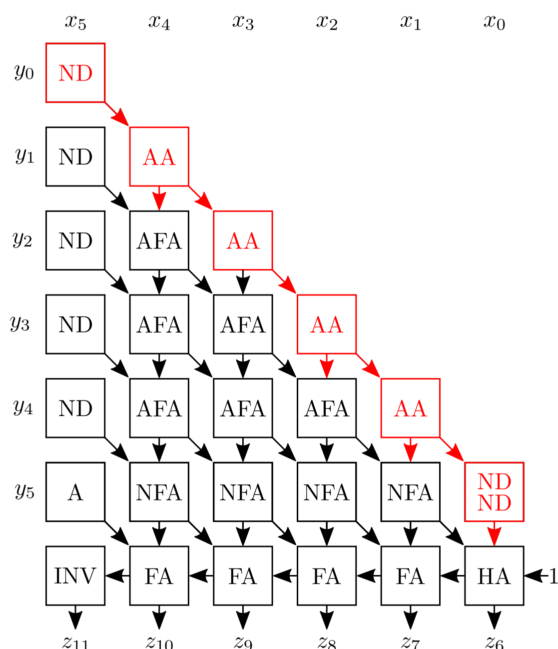


Figure 1.53 – 8-bit signed AAMIII structure

Applying the method described above, AAMIII from width 4 to 12 are generated and compared with the two previous versions in [48]. The accuracy results in terms of maximum error E_M, average error μ_e and variance of error σ_e^2 are given in Table 1.21. In this table, only signed versions of the three AAMs versions were studied. There are slight benefits with AAMIII in terms of maximal error comparing to AAMII, but the relative gains decreases when the width of the operator increases. However, the benefits in terms of mean and variance of the error, and therefore in terms of power of error, are very important and increase with the width of the operator. E.g, for a 12-bit operator, the power of error is reduced by 52.0% comparing AAMIII to AAMII, which is a huge gain considering there is no area overhead, as showed in Table 1.22. For larger operators, all AAMs area ratio decreases towards 0.5 and so the area overheads of AAMII and AAMIII towards AAMI become negligible.

The error maps of 4-bit, 8-bit and 16-bit AAMIII are visible on Figure 1.54. On these error maps, four squared areas are clearly visible. The lower-left square, corresponding to low-amplitude inputs, returns low amplitude error. The three other squares perform globally much higher error, with patterns which need to be different depending on the operator's width.

We can then conclude about AAMs that:

- AAMI has brought an interesting way to reduce by a factor of 2 the area of an n-bit fixed-width parallel multiplier, removing the least significant part of the operator.
- AAMII has proposed an improvement of AAMI by adding simple cells on the operator diagonal.

Multiplier	Error metric	n = 4	n = 6	n = 8	n = 10	n = 12
AAMI	E_M	33	193	1,281	6,145	32,769
AAMII		21	107	515	2,403	10,979
AAMIII		17	89	441	2,105	9,785
AAMI	μ_e	6.96	41.01	188.29	906.40	3,842.06
AAMII		7.20	37.27	170.46	736.62	3,065.25
AAMIII		5.17	24.07	105.96	456.14	1,907.36
AAMI	σ_e^2	39.80	788.45	22,959.01	416,043	9,204,493
AAMII		28.24	537.70	10,158.54	190,805	3,417,020
AAMIII		17.63	320.65	6,031.32	112,079	1,973,508

Table 1.21 – Accuracy comparison of signed AAM versions I, II and III [48]

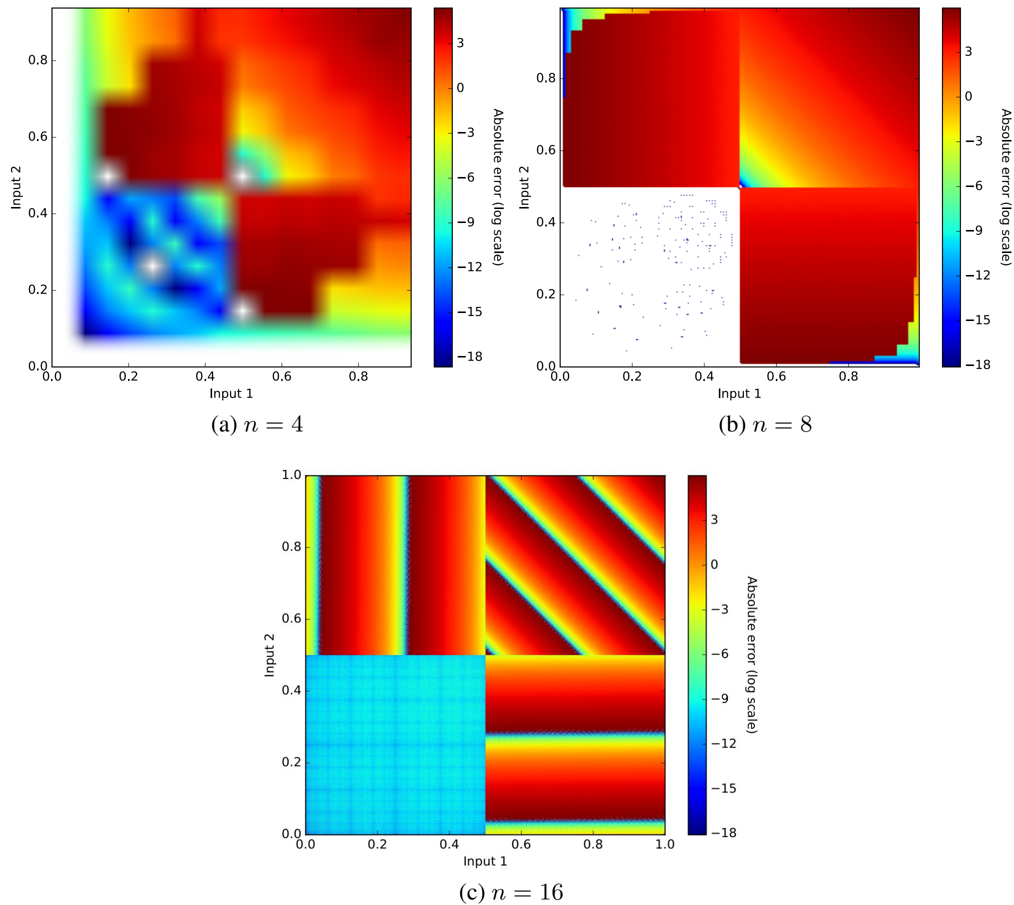


Figure 1.54 – Error maps of 4-bit, 8-bit and 16-bit AAMIII

Multiplier	$\mathbf{n = 4}$	$\mathbf{n = 6}$	$\mathbf{n = 8}$	$\mathbf{n = 10}$	$\mathbf{n = 12}$
AAMI	0.555	0.536	0.527	0.522	0.518
AAMII	0.608	0.569	0.550	0.5540	0.533
AAMIII	0.608	0.569	0.550	0.5540	0.533

Table 1.22 – Area ratio comparing to original parallel adder for AAM versions I, II and III [48]

- AAMIII has proposed an optimized modification of the diagonal cells in order to set the approximate operator bias to its minimum, achieving great gains in terms of accuracy with no area overhead.

As a consequence, AAMIII designing method is a very efficient way to build an approximate n-bit fixed-width array multiplier with a nearly-50% area reduction.

1.4.2.2 Error-Tolerant Multiplier

The Error-Tolerant Multiplier (ETM) [35] is inspired from the principle of ETAI [40] studied in Section 1.4.1.2. Indeed, it is composed of two parts:

- the MSB part, which is a conventional accurate multiplier, and
- the LSB part, which is designed for very fast approximation.

For the accurate MSB part, as an $n \times n$ multiplier has a $O(n^2)$ area complexity, dividing the accurate part digits number by a factor k insures a k^2 area saving. Therefore, the benefit of reducing the accurate multiplication part is much bigger than for adders. For the approximate LSB part, an approximation resembling ETAI, described by Algorithm 1 page 44: the inputs of the approximate part are read from MSB to LSB, performing a logical OR until two 1s are met on the same position i. When this happens, all outputs from i down to 0 are set to 1. An illustration of the process is showed in Figure 1.55.

ETM also embeds a system detecting if the MSB part of the inputs has at least one bit worth 1. This way, if there is no 1 (meaning it is a low amplitude multiplication), then the accurate multiplier is used for the multiplication of the LSB part. This way, the calculation is 100% accurate with no overhead on the design area, except for multiplexing the inputs. Therefore, contrary to ETAI, ETM performs well for low amplitude inputs. The system view of ETM can be seen on Figure 1.56. For a good use of this system, the accurate and approximate parts lengths of the multiplier need to be of equal width.

The LSB approximation part is achieved the same way as ETAI thanks to a control block (see Figure 1.31). The only difference is that all sub-blocks are CSGCI blocks (see Section 1.4.1.2), meaning the control signal is not propagated as fast as for ETAI. This choice is probably due to the fact that the approximation part is faster than the accurate multiplication part, and so there is no need to reduce the delay on the control block which would imply an area and power overhead. The accurate multiplication is performed by an array multiplier of size $n/2$.

Figure 1.55 – Example of ETM multiplication process [35]

Figure 1.56 – Structure of a 12-bit ETM [35]

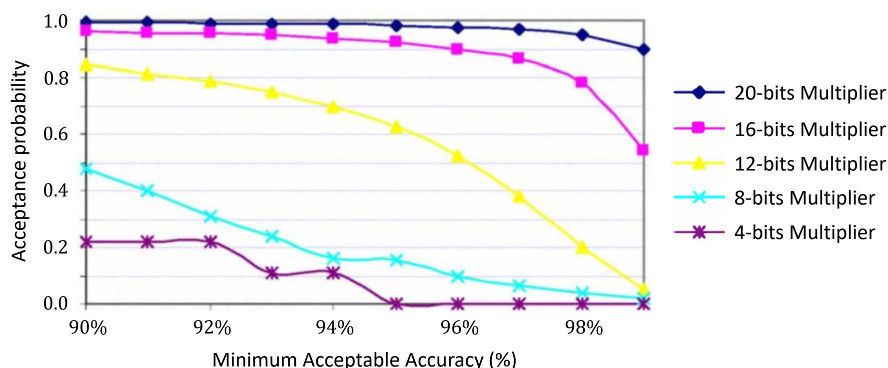


Figure 1.57 – Accuracy evaluation by simulation for ETM [35]

The evaluation of accuracy is performed using AP for a given MAA in [35]. This metric is presented in Section 1.4.1.2. Figure 1.57 presents the results for a MAA range from 90% to 99%, using ETM of input width from 4 to 20 bits. We can suppose these ETM have the same lengths $n/2$ for their accurate and approximate parts. Results were obtained by simulation of $65,000$ sets of inputs for the 20-bit multiplier, and $6,500$ for the others. Such a choice can be discussed, the number of points being objectively too low to ensure all approximation cases to be met. On the graph, what can first be noticed is that small-width ETMs have a very low accuracy, with only less than 20% AP for a 95% MAA for 8-bit ETM for instance. For larger operators such as 16-bit or 20-bit, AP seems stable above 90% MAA, but the 16-bit AP curve dramatically decreases after 97%. Results for lower MAA can be found in [35]. To conclude about accuracy, the ETM model seems quite well adapted to large operators, small ones often generating high errors relatively to their computing range. For a 99% MAA, 20-bit ETM seems to be the minimal operator for a 90% AP. Of course, the interest of large widths in approximate computing is questionable. Answers are given in Chapter 4.

Design simulations were performed on 0.18 μm CMOS process with a 10 MHz frequency. The comparison was made between a conventional 12-bit array multiplier and a 12-bit ETM with 6-bit accurate part and 6-bit approximate part. Power and delay were reported for five sets of inputs, the PDP of the five corresponding operations are given in Figure 1.58. The detailed results are given in [35]. These results show that energy consumption strongly depends on the inputs. On the five tested sets of inputs, energy consumption of ETM is 75% to 90% lower than the conventional 12-bit multiplier, though we could regret the very low number of performed tests. Looking at detailed test results, we can see there is nearly no improvement in calculation delay, whereas power improvement is quite important. In terms of area, the 12-bit ETM covers 491 μm^2 whereas parallel multiplier covers 1028 μm^2, which is more than twice. Since a 12-bit array multiplier is roughly four times as large as its 6-bit version, this means that the approximation part has an area overhead nearly equivalent to the exact part area.

To conclude about this operator, ETM proposes a n-bit approximate multiplication using a $n/2$-bit accurate multiplier, that can be used for the MSB part or LSB part of the computation depending on the input range. This allows for high energy and area savings, with quite good accuracy results for large-width operators. Once again, using a unique scalar metric does not

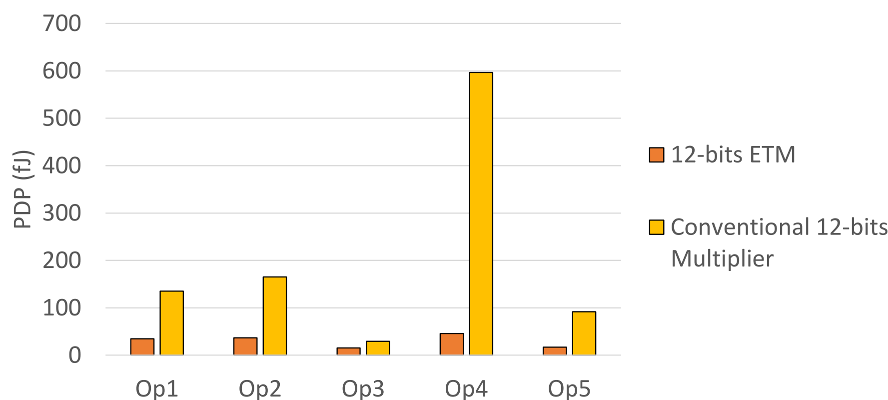


Figure 1.58 – PDP for 12-bits ETM and array multiplier [35]

give details about the nature of the error, but this operator, by nature, can perform high amplitude errors in a small number of worst-case input sets. It can be noticed that ETM has an area which is comparable to the area of an AAM version II or III (see Section 1.4.2.1), but ETM is an $n \times n \rightarrow 2n$ multiplier whereas APMs are only $n \times n \rightarrow n$. However, this does not ensure at all that ETM is more accurate than the best of them (AAMIII), and more tests would need to be performed to determine its accuracy.

1.4.2.3 Approximate Multipliers using Modified Booth Encoding

As discussed in Section 1.3.4, modified Booth encoding allows the number of partial product of the summand grid to be importantly reduced. Usual Radix-4 encoding divides the number of partial products by two for an even input width. As the critical part of a multiplier is the carry-save reduction of the partial products, using modified Booth encoding is a good potential for area, delay and power saving. Therefore, in the context of approximate multipliers in low-power applications, using this technique as a starting point is a potentially efficient way to save energy.

In [49], the authors present a fixed-width Booth multiplier with a simple error-correction system. In FxP context, fixed-width multipliers are obtained by truncating the LSB half part of the multiplication output. A classical approximation for fixed-width multipliers consists in removing the LSB half part of the partial products and to compensate the induced bias by removing an adapted constant. In [49], error compensation is performed by keeping a few recombined cells of the most significant column of the LSB part. Therefore, the usual constant output bias is replaced by an input-dependent bias.

To determine how error-compensation cells should be recombined, the authors studied α_{n-1}, the number of carries generated at rank $n-1$ as a function of β, the sum of ones in summand grid at rank $n-1$. The author statistically determines that the best choice to minimize the error is to have $\alpha_{n-1} = \beta$. Therefore, every rank $n-1$ cell in summand grid is kept without any recombination. The resulting summand grid is given by Figure 1.59. From this point, the column of rank $n-1$ in the summand grid will be referred as LP_{major} (Low Part,

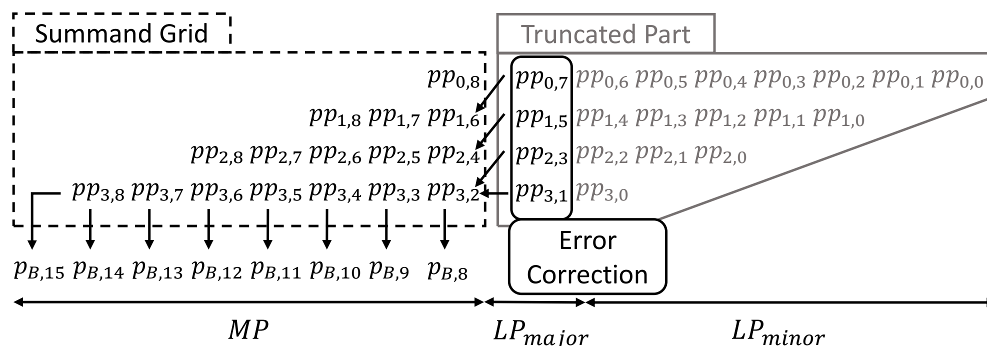


Figure 1.59 – Summand grid for an 8-bit fixed-width Booth multiplier with LP_{major}-based error correction [49]

Y_i	X_{sel}	$2X_{sel}$	NEG
-2	0	1	1
-1	1	0	1
0	0	0	0
1	1	0	0
2	0	1	0

Table 1.23 – Equivalence between Radix-4 modified-Booth-encoded symbol Y_i and control bits in partial product generation

major position) and all the lower significance elements LP_{minor} (Least significant Part, minor position). The most significant part which corresponds to the n bits which are always kept for fixed-width multipliers output calculation will be referred as MP (Most significant Part, major position). The proposed 8-bit multiplier has 46% less gates than the accurate equivalent. In [49], accuracy results are given in terms of Signal-to-Noise Ratio (SNR). For a 16-bit multiplier, the SNR is 76.64 dB. More detailed results about this multiplier are respectively given in [50] and [51], reported in Tables 1.25, 1.28 and 1.30, and denoted as Fixed-width modified-Booth-encoded Multiplier version I (FBMI).

Another fixed-width Booth multiplier with input-dependent error correction, denoted here as Fixed-width modified-Booth-encoded Multiplier version II (FBMII), is proposed in [50]. In FBMII, only the Booth encoder control outputs are considered for error correction. The idea is to perform a statistical approximation of the carries generated by the truncated part in function of the coded input to be able to design a well-adapted error compensation system. Radix-4 modified Booth-encoding is determined by radix-4 symbols Y_i. For the generation of a given partial product $pp_{i,j}$, the value of input x_j and bits X_{sel}, $2X_{sel}$ and NEG at determined by Y_i from Table 1.23 are used, and the resulting generation circuit is depicted in Figure 1.60. To get 1-bit statistics on the encoded multiplier instead of the 3 bits representing a symbol Y_i, Y_i' is

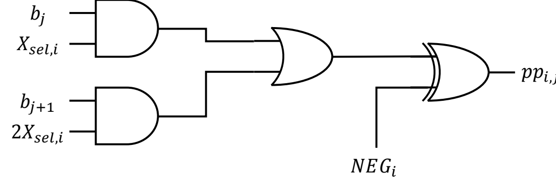


Figure 1.60 – Partial product generation

defined as

$$Y_i' = \begin{cases} 1 & \text{if } Y_i \neq 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1.27}$$

Therefore, for a given chain of symbols $\{Y_i\}_{i \in LP}$, the corresponding chain of bits $\{Y_i'\}_{i \in LP}$ can be obtained with a very small area and delay overhead performing logical OR as

$$Y_i' = X_{sel,i} \mid 2X_{sel,i}. \tag{1.28}$$

Then, statistics on the chain of bits $\{Y_i'\}_{i \in LP}$ must be performed in order to find the best way to estimate carries generated in LP group. For this, S_{LP}, the sum of all weighted partial products of LP group, is introduced and defined as

$$S_{LP} = \sum_{0 \leq 2i+j < n-1} p_{i,j} 2^{2+2i+j-n}. \tag{1.29}$$

Indeed, a good estimation of S_{LP} for a given input allows to inject in MP part a good correcting bias. For this, the exact carries of LP_{major} are propagated and the carries of LP_{minor} are estimated. $S_{LP_{minor}}$ is defined the same way as S_{LP} but restricted to LP_{minor}.

First, a method leveraging exhaustive simulation is proposed in [49]. The idea is to list all occurrences of all possibilities for $\{Y_i'\}_{i \in LP_{minor}}$ and to calculate the rounded statistical mean of $S_{LP_{minor}}$ for each of these occurrences. E.g., for $n = 10$, there are 108 ways to obtain $\{Y_i'\}_{i \in LP_{minor}} = 1, 0, 1, 1$. Amongst these 108 sequences, 52 verify $\{E[S_{LP_{minor}}]\}_r = 1$ and 56 verify $\{E[S_{LP_{minor}}]\}_r = 2$, where $\{\cdot\}_r$ is rounding operation. Therefore, the best compensation of the truncated carries generated by LP_{minor} is 2. As $\{E[S_{LP_{minor}}]\}_r$ always is between 0 and 2, two carries must be transmitted to LP_{major}, following rules of Table 1.24. For $n = 10$, the Karnaugh map representations of a_carry_0 and a_carry_1 as a function of $\{Y_i'\}_{i \in LP_{minor}}$ are given by Figure 1.61. The logic for the carry generation can be determined from this map and is given by

$$\begin{array}{lcl} \text{a_carry}_0 & = & Y_3' \mid Y_2' \mid Y_1' \mid Y_0' \\ \text{a_carry}_0 & = & Y_3'Y_2'(Y_1' \mid Y_0') \mid Y_1'Y_0'(Y_3' \mid Y_2') \end{array} \tag{1.30}$$

This circuit can be implemented using 8 basic logic gates. The relation between the value of the chain of bits $\{Y_i'\}_{i \in LP_{minor}}$ and the value $E[S_{LP_{minor}}]$ is given by

$$E[S_{LP_{minor}}] = 2^{-1} \sum_{i=0}^{n/2-2} Y_i'. \tag{1.31}$$

Rounded value	$\mathbf{a_carry_0}$	$\mathbf{a_carry_1}$
0	0	0
1	1	0
2	1	1

Table 1.24 – Representation of approximate carry values

$Y_3'Y_2' \backslash Y_1'Y_0'$	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

(a) a_carry_0

$Y_3'Y_2' \backslash Y_1'Y_0'$	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	0	1	1	1
10	0	0	1	0

(b) a_carry_1

Figure 1.61 – Karnaugh map representation of approximate carry for $n = 10$

Then, the value of the carry to propagate on LP_{major} is obtained by rounding. As $\{Y_i'\}_{i \in LP_{minor}}$ can only be 0 or 1, the maximal value for the propagated carry is $\left\{2^{-1}\left(n/2-1\right)\right\}_r$, and so the number of binary approximated carries $N_{\text{a_carry}}$ is always $\lfloor n/4 \rfloor$.

A methodology for the error correction system can then be derived:

1. For an n-bit FBMII, the number of approximated carries is $N_{\text{a_carry}} = \lfloor n/4 \rfloor$, denoted as $\text{a_carry}_0, a_carry_1, \ldots, \text{a_carry}_{N_{\text{a_carry}-1}}$.
2. $\forall i \in [\![0, N_{\text{a_carry}} - 1]\!], \text{a_carry}_i = 1 \Leftrightarrow \left(\sum_{i=0}^{n/2-2} Y_i'\right) \leq (2i+1)$.
3. Approximate carry circuit is designed by defining the compensation carry logic, e.g. using a Karnaugh map.

Using this method, accuracy results are given in [50] in terms of error mean and variance for $n = 10$ and $n = 12$, showed in Table 1.25. *Rounded* and *Truncated* respectively refer to a rounding and truncation of the output of the original accurate multiplier. FBMII proposes a better accuracy than its predecessor FBMI, but also beats truncation despite its nearly-50% area savings. With the proposed method, for bigger multiplier size, the overhead of this approximate carry generation procedure, denoted as Approximate Carry Generation Procedure version I (ACGPI), becomes too impacting in terms of area and delay, and is not suitable. To face that,

Multiplier	Error metric	$\mathbf{n = 10}$	$\mathbf{n = 12}$
Rounded	μ_e	$4.87\text{E}{-4}$	$1.22\text{E}{-4}$
Truncated		$9.66\text{E}{-4}$	$2.43\text{E}{-4}$
FBMI		$1.22\text{E}{-3}$	$3.21\text{E}{-4}$
FBMII		$6.28\text{E}{-4}$	$1.63\text{E}{-4}$
Rounded	σ_e^2	$8.01\text{E}{-8}$	$4.51\text{E}{-9}$
Truncated		$3.17\text{E}{-7}$	$1.99\text{E}{-8}$
FBMI		$8.22\text{E}{-6}$	$5.21\text{E}{-7}$
FBMII		$1.94\text{E}{-7}$	$1.33\text{E}{-8}$

Table 1.25 – Accuracy comparison for FBMI and FBMII using ACGPI

the author proposes another approximate carry generation procedure denoted as Approximate Carry Generation Procedure version II (ACGPII):

1. $\{Y_i'\}_{i \in LP_{minor}}$ is divided in groups of 3, the last group being of size 1, 2 or 3 depending of the set size.
2. Each group bits are summed using a FA (or HA or a wire for the last group).
3. At each FA output, the carry signal c is an approximate carry, and the sum signal s is summed with the ones from the other groups.
4. The process is repeated until only one sum signal is left.
5. Finally, 1 is added to the last adder.

With ACGPII, original ACGPI is slightly modified by a new higher level approximation, but the sum of the generated carries remains the same, just as showed for $n = 8$ in Table 1.26. The design for $n = 32$ is given in Figure 1.62. With only 7 FA and 1 HA, the ACGPII represents a very low overhead in term of area. Moreover, with only three levels, it only adds a small delay to the critical path.

A comparison of delays and areas using ACGPI and ACGPII for different operator sizes is provided in Table 1.27. For $n <= 10$, ACGPI is better in both domains than ACGPII, in addition to be more accurate by construction in the estimation of the carries. When the operator size grows, ACGPII gets much more efficient in delay and area. Indeed, for a 32-bit FBMII, ACGPII is 78% faster than ACGPI for 56% area benefit.

Delay and area comparisons of FBMI and FBMII using ACGPII were performed in [50]. They show that their performance is nearly the same, with a slight advantage for FBMII/ACGPII on delay for all sizes and also on area from $n = 14$. Using Synopsys tools, FBMI and FBMII show very similar power consumption at least for $n = 10$ and $n = 12$, with a bit more than 60% of the power consumption of an ideal fixed-width Booth multiplier, meaning a complete Booth multiplier with output truncation or rounding. Accuracy comparisons between rounding, truncation, FBMI and FBMII using ACGPII are given for $n = 16$ and $n = 20$ by Table 1.28. They show that proposed FBMII/ACGPII significantly beats both truncation and FBMI in terms of

	ACGPI		ACGPII	
$\mathbf{Y'_2Y'_1Y'_0}$	$\mathbf{a_carry_0}$	$\mathrm{a_carry_1}$	$\mathbf{a_carry_0}$	$\mathrm{a_carry_1}$
0 0 0	0	0	0	0
0 0 1	1	0	0	1
0 1 0	1	0	0	1
0 1 1	1	0	1	0
1 0 0	1	0	0	1
1 0 1	1	0	1	0
1 1 0	1	0	1	0
1 1 1	1	1	1	1

Table 1.26 – Approximate carry signals generated by ACGPI and ACGPII for $n = 8$

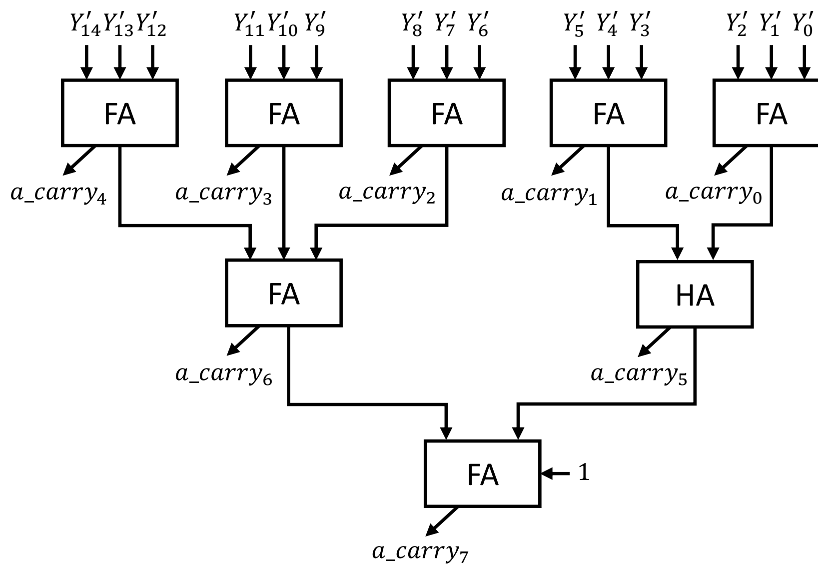


Figure 1.62 – Approximate carry generation circuit using ACGPII for $n = 32$

	Delay (ns)		Area (# of NAND gates)	
n	ACGPI	ACGPII	ACGPI	ACGPII
10	4.48	6.24	10	11
12	7.21	6.06	22	15
14	7.94	6.21	31	20
16	10.25	7.56	43	23
18	10.78	7.56	55	27
32	18.23	10.20	189	60

Table 1.27 – Comparison of delay and area of ACGPI and ACGPII [50]

Multiplier	Error metric	$\mathbf{n = 16}$	$\mathbf{n = 20}$
Rounded	μ_e	7.51E−6	4.63E−7
Truncated		1.47E−5	9.02E−7
FBMI		1.92E−5	1.15E−6
FBMII		1.07E−5	6.35E−7
Rounded	σ_e^2	1.96E−11	8.08E−14
Truncated		7.96E−11	3.21E−13
FBMI		1.92E−10	8.24E−13
FBMII		6.38E−11	2.44E−13

Table 1.28 – Accuracy comparison for FBMI and FBMII using ACGPII

error mean and variance. As a matter of fact, FBMII/ACGPII has a mean square error which is 68.1% inferior to FBMI for $n = 16$ and 69.8% for $n = 20$.

To conclude about FBMII, this fixed-width Booth multiplier proposes much better performance than FBMI with no area, power or delay overhead, thanks to improved error compensation systems which are ACGPI and ACGPII. The first one consists in a statistical analysis of the carries generated in the truncated part in function of the Booth-coded value of the input, whereas the second one is an approximate version of the first one, allowing a dramatic reduction of the theoretical correction circuit size and delay.

In [51], Juang proposes a fixed-width Booth multiplier with a very low cost error correction system, based on the estimation of LP_{minor} bits of the summand grid in function of LP_{major} bits values. For more convenience, the proposed multiplier will be denoted as Fixed-width modified-Booth-encoded Multiplier version III (FBMIII). This section presents the methodology for an 8-bit FBMIII as well as accuracy and area comparisons for different sizes of FBMIII and previously described multipliers.

To design an 8-bit FBMIII, LP_{minor} levels need to be discriminated like showed on Figure 1.63. The four level weighted symbol strings of LP_{minor} are denoted as w, x, y and z, and can be expressed as

$$\begin{aligned} w &= \sum_{i=0}^{6} 2^{i-7} pp_{0,i}, \\ x &= \sum_{i=0}^{4} 2^{i-5} pp_{1,i}, \\ y &= \sum_{i=0}^{2} 2^{i-3} pp_{2,i}, \\ z &= 2^{-1} pp_{3,0}. \end{aligned} \tag{1.32}$$

As a reminder, each symbol $pp_{i,j}$ is in the set $\{-1, 0, 1\}$. The idea of FBMIII error correction is to find the relation between LP_{major} bits and these symbolic strings. For this, the best values

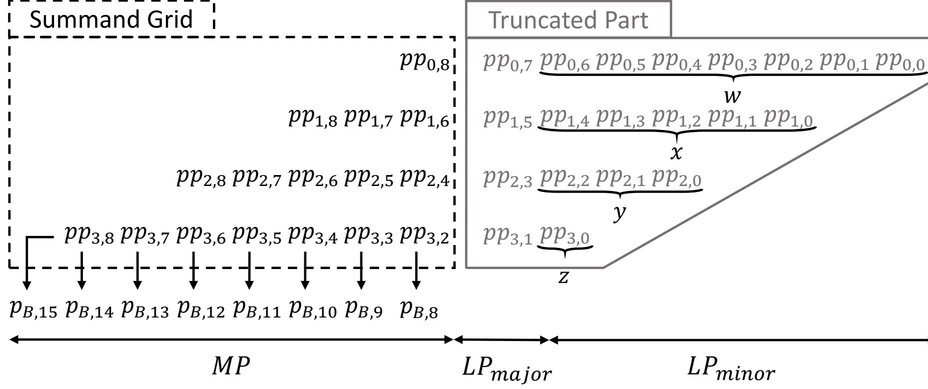


Figure 1.63 – LP_{minor} level discrimination for the design of FBMIII

of q_1, q_2 and q_3 must be found in order to get as close as possible to

$$\begin{aligned} w + x &\equiv q_1 \times (pp_{0,7} + p_{1,5}), \\ y &\equiv q_2 \times pp_{2,3}, \\ z &\equiv q_3 \times pp_{3,1}. \end{aligned} \tag{1.33}$$

Their best representative is their mathematical mean. q_2 mathematical mean can be decomposed as

$$E[q_2] = \sum_{k=-1}^{1} q_{2,k} \times P(pp_{2,3} = k). \tag{1.34}$$

where $q_{2,k}$ is the optimally correcting value of q_2 for a given k. $P(pp_{2,3} = k)$ can be easily computed for any value of k assuming that each multiplier binary input is equiprobable. Table 1.29 gives the probabilities for all $pp_{i,j}$ to be worth k in LP. Assuming these data, the calculation of $E[y|pp_{2,3} = 1]$ is

$$\begin{aligned} E[y|pp_{2,3} = 1] &= E\left[2^{-1} \times pp_{2,2} + 2^{-2} \times pp_{2,1} + 2^{-3} \times pp_{2,0}\right] \\ &= 2^{-1} \times E[pp_{2,2}] + 2^{-2} \times E[pp_{2,1}] + 2^{-3} \times E[pp_{2,0}] \\ &= 2^{-1} \times (1/2) + 2^{-2} \times (1/2) + 2^{-}3 \times (1/2) \\ &= 0.4375. \end{aligned} \tag{1.35}$$

Therefore, as $y = q_{2,k} \times k$, the best value for $q_{2,1}$ is 0.4375. The same computation and reasoning for $k = -1$ and $k = 0$ respectively gives $q_{2,-1} = -0.4375$ and $q_{2,0} \in \mathbb{R}$. As $q_{2,0}$ can take any value, 1 is chosen in an arbitrary manner. By injecting these three $q_{2,k}$ in Equation 1.34 as well as probability values of Table 1.29, then $E[q_2]$ is given by

$$\begin{aligned} E[q_2] &= 0.4375 \times (3/16) + 0.4375 \times (3/16) + 1 \times (5/8) \\ &= 0.7890625 \end{aligned} \tag{1.36}$$

Therefore, $q_2 = 1$ is chosen as the approximated coefficient for y. The same process is applied for q_1 and q_3, and finally we get

$$q_1 = 0, \quad q_2 = 1, \quad q_3 = 1. \tag{1.37}$$

Partial products	$\mathbf{p_{i,j} = -1}$	$\mathbf{p_{i,j} = 0}$	$\mathbf{p_{i,j} = +1}$
$\mathbf{p_{i,0}}$	1/8	3/4	1/8
$\mathbf{p_{0,j}}$	1/4	1/8	5/8
Others	3/16	3/16	5/8

Table 1.29 – Value of $P\left(pp_{i,j} = k\right)$ in LP

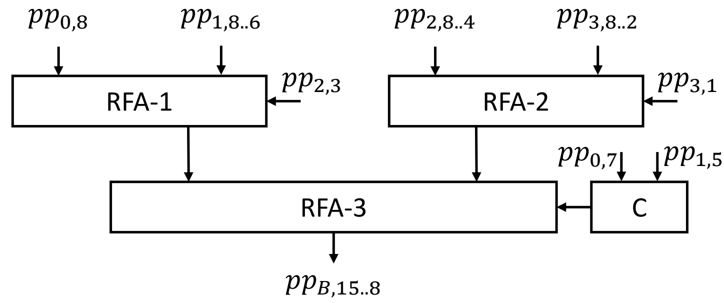


Figure 1.64 – 8-bit FBMIII schematized structure

Perfectly knowing w, x, y and z as a function of all inputs would lead to the following compensation value:

$$C^V_{ideal} = \left\lfloor 2^{-1}\left(pp_{0,7} + pp_{1,5} + pp_{2,3} + pp_{3,1}\right) + 2^{-1}\left(w + x + y + z\right)\right\rfloor. \tag{1.38}$$

Approximating w, x, y and z using q_1, q_2 and q_3 leads to

$$\begin{aligned} C^V_{app} &= \lfloor 2^{-1}\left(pp_{0,7} + pp_{1,5} + pp_{2,3} + pp_{3,1}\right) + 2^{-1}\left(q_1 \times \left(pp_{0,7} + pp_{1,5}\right)\right. \\ &\quad \left. + q_2 \times pp_{2,3} + q_3 \times pp_{3,1}\right)\rfloor, \\ &= \lfloor 2^{-1}\left(pp_{0,7} + pp_{1,5}\right)\rfloor + pp_{2,3} + pp_{3,1}. \end{aligned} \tag{1.39}$$

In spite of taking into consideration both LP_{major} and LP_{minor}, the computation of the compensation value C^V_{app} is not more complex than the compensation value of FBMI which only took LP_{major} into account for compensation, and so better compensation performance can be expected. Indeed, for C^V_{app} to be applied, adding $pp_{2,3}$ and $pp_{3,1}$ does not need any extra gates, and $\lfloor 2^{-1}\left(pp_{0,7} + pp_{1,5}\right)\rfloor$ can be performed with two AND gates. The global FBMIII can be mapped as in Figure 1.64, where RFA blocks are Redundant Full-Adder blocks and C block is the partial correction block for the appliance of the two partial compensations $\lfloor 2^{-1}\left(pp_{0,7} + pp_{1,5}\right)\rfloor$. $pp_{k,i..j}$ refers to the partial symbolic string from $pp_{k,i}$ to $pp_{k,j}$ and $pp_{B,i..j}$ refers to the Booth-encoded multiplier output from rank i to rank j.

The author gives comparisons of accuracy between the truncated output version of the fixed-width Booth multiplier and FBMI presented above, in terms of absolute error mean and

Multiplier	Error metric	$\mathbf{n=6}$	$\mathbf{n=8}$	$\mathbf{n=10}$	$\mathbf{n=12}$
Truncated	$\mu_{\lvert e \rvert}$	22	91	371	1,506
FBMI		23	107	477	2,052
FBMIII		24	104	449	1,925
Truncated	σ_e^2	236	4,083	67,576	1,104,876
FBMI		290	6,247	125,055	2,341,510
FBMIII		234	4,287	73,647	1,280,361

Table 1.30 – Accuracy comparison for FBMI and FBMIII

error variance. These metrics for a given n-bit operator denoted as *op* are defined as:

$$\begin{aligned} \mu_{|e|} &= \frac{1}{2^n}\left(\sum_{X=-2^{n-1}}^{2^{n-1}-1}\left(\sum_{Y=-2^{n-1}}^{2^{n-1}-1}|X\times Y - Z_{op}|\right)\right), \\ \sigma_e^2 &= \frac{1}{2^n}\left(\sum_{X=-2^{n-1}}^{2^{n-1}-1}\left(\sum_{Y=-2^{n-1}}^{2^{n-1}-1}|X\times Y - Z_{op}|^2\right)\right), \end{aligned} \tag{1.40}$$

where Z_{op} is the result of the operation $X \times Y$ performed by the operator *op*. Accuracy results are given in Table 1.30. In terms of absolute error mean, it can be noticed that FBMIII achieves slightly better performance than FBMI, but has an error variance which is twice as small. In terms of error power, an 8-bit FBMIII is 14.7% more efficient than its FBMI equivalent, and 23.9% for their 12-bit versions. FBMIII accuracy benefits towards FBMI seem to increase with the size of the operator. However, FBMIII is slightly beaten by truncation in term of accuracy. This is still good performance taking into account that FBMIII saves a 44% area compared to an 8-bit truncated-output fixed-width Booth multiplier and 49% compared to a 12-bit one. More results about area are detailed in [51] as well as more tests comparing FBMI and FBMIII on image processing tests, confirming FBMIII to be more accurate on several metrics such as root mean square error, SNR and PSNR. Therefore, FBMIII proposes better global performance than FBMI on many metrics, with a smaller error correction system.

In this section, three fixed-width Booth operators with error correction system were presented:

- FBMI [49] proposes a low-cost error-compensation system, considering only LP_{major}.
- FBMII [50] proposes a higher-cost error-compensation system, only based on the value of the input multiplier, but taking it entirely into account ($LP_{major} + LP_{minor}$). This higher cost allows to strongly beat FBMI in terms of accuracy, and even the basic fixed-width Booth multiplier obtained by truncation of the output. Compared to its summand grid, FBMII error compensation system is still small, and so it represents a very interesting fixed-width Booth multiplier.
- FBMIII [51] has the lowest-cost error-compensation system, which has even smaller cost than FBMI's LP_{major} consideration. Moreover, it has better performance than FBMI, but does not beat output truncation method as FBMII does.

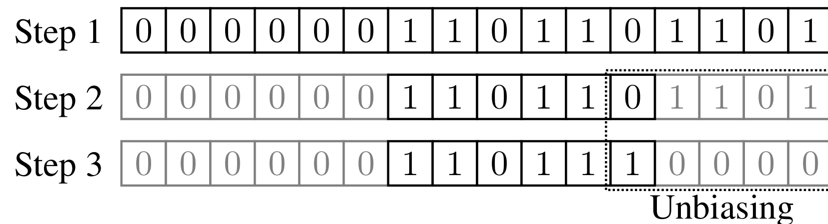


Figure 1.65 – DRUM input unbiasing process. Step 1: original input. Step 2: selecting the k non-zeros MSBs. Step 3: unbiasing. Greyed cells represent the virtual value of dropped bits.

To conclude, FBMII seems to be the most efficient fixed-width Booth operator in terms of accuracy, though FBMIII proposes a lower-cost error compensation. In terms of delay, the compensation system overhead for the three presented operators is nearly negligible compared to the cost of the carry-save partial product reduction. Therefore, for the best accuracy, FBMII should be given high priority, and FBMIII should be chosen only if area is the critical resource.

1.4.2.4 Dynamic Range Unbiased Multiplier

In [52], a novel approximate multiplier referred as Dynamic Range Unbiased Multiplier (DRUM) is inspired from floating-point multiplication. As a reminder, FlP multiplication process is described in Section 1.2.2. Indeed, the idea of DRUM with n-bit inputs is to use an accurate multiplier of size $k < n$, shifting the n-bit inputs in a way that the input MSBs of the multiplier are fed with the most significant one of each input. This way, no effort is wasted in the multiplication uselessly processing "high significance zero $\times$ zero".

To reduce the approximation, inputs are unbiased before multiplication. As only a subset of k bits is extracted from the inputs, all the less significant bits are virtually set to 0, causing an error which is always in the same direction. To prevent this, the LSB of the k bits extracted for multiplication is set to 1. The process applied on each input for unbiasing is depicted in Figure 1.65. Once the inputs extracted and unbiased, the accurate k-bit multiplication is performed. Finally, the result is shifted so the output of k-bit multiplication is expressed with its legitimate significance. The corresponding structure of DRUM is depicted in Figure 1.66 [52].

First, leading-zero detection is performed on each input to get the position of the first one in k-bit multiplication. Then, after input unbiasing, k-bit multiplication is performed. Finally, the result is shifted using the sum of the leading-zero detection values of the inputs. Designing DRUM is therefore about finding a good compromise for k. Decreasing k value by 1 diminishes the size of the effective multiplier, but increases the size of the multiplexers of Figure 1.66, and the $L0D$ needs to be able to count one potential more leading zero. Also, the final maximal shifting is increased by 1. Generally, decreasing k decreases area but may increase delay. To keep the benefit of using leading-zero detection, the inputs must be unsigned. Therefore, a signed version of DRUM must $unsign$ the inputs using two's complement transformation before being applied, and manage the sign of the output depending on the sign of the inputs, which can be achieved with a small overhead.

Using Synopsys Design Compiler and Mentor Graphics Modelsim with 65-nm standard

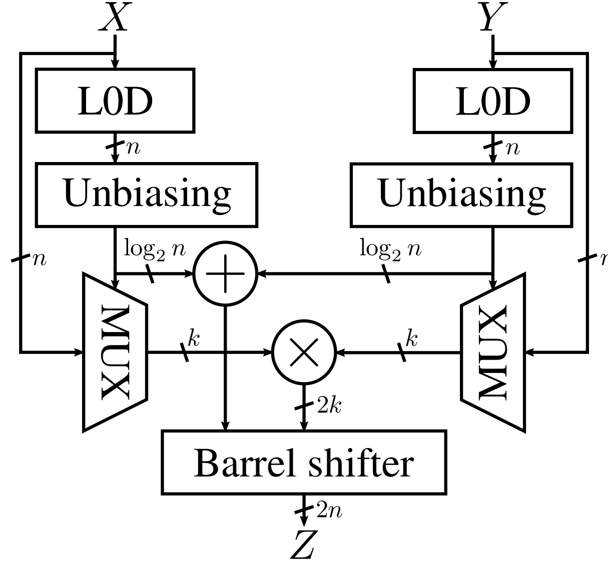


Figure 1.66 – Structure of DRUM. *L0D* stands for Leading-zero-Detector, used to select the bits to be effectively multiplied and to perform the final shift.

cell libraries, area and power for 16-bit DRUM as a function of k are depicted in Figure 1.67 [52]. These simulations show that substantial reductions in area and power are reached. For 16-bit DRUM with $k = 3$, more than 80% of area and more than 90% of power are saved with respect to a 16-bit accurate multiplier – the structure of the reference multiplier being unknown. For $k = 8$, nearly 50% area and power are saved. The intermediate k values seem to show near-linear savings.

These savings need to be put in relation with the errors being performed. In [52], four error metrics are explored, all relative to the accurate result. If the relative error referred as RE is defined by

$$\mathrm{RE} = \frac{Z - \hat{Z}}{Z}, \tag{1.41}$$

where Z is the exact result of 16-bit multiplication $X \times Y$ and $\hat{Z}$ the result of the same multiplication using DRUM, the four error metrics explored are defined by

$$\begin{aligned}
\mathrm{Max}_{\mathrm{RE}} &= \max_{i \in S} |\mathrm{RE}_i|, \\
\mathrm{MA}_{\mathrm{RE}} &= \frac{1}{N_S} \sum_{i \in S} |\mathrm{RE}_i|, \\
\mu_{\mathrm{RE}} &= \frac{1}{N_S} \sum_{i \in S} \mathrm{RE}_i, \\
\sigma_{\mathrm{RE}} &= \frac{1}{N_S} \sum_{i \in S} \mathrm{RE}_i^2,
\end{aligned} \tag{1.42}$$

where $\mathrm{Max}_{\mathrm{RE}}$ is the maximum relative error, $\mathrm{MA}_{\mathrm{RE}}$ the mean absolute relative error, μ_{RE} the

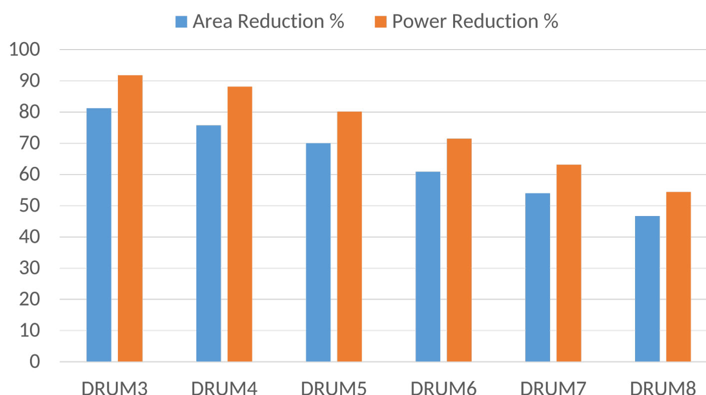


Figure 1.67 – Area and power benefits of 16-bit DRUM relatively to 16-bit accurate multiplier [52] – DRUMk refers to 16-bit DRUM using k-bit multiplier.

Metric \ k	3	4	5	6	7	8
$\mathrm{Max}_{\mathrm{RE}}$ (%)	56.25	26.56	12.86	6.31	3.1	1.54
$\mathrm{MA}_{\mathrm{RE}}$ (%)	11.90	5.89	2.94	1.47	0.73	0.37
μ_{RE} (%)	2.08	0.53	-0.14	-0.04	0.01	0.01
σ_{RE} (%)	14.75	7.26	3.61	1.80	0.90	0.45

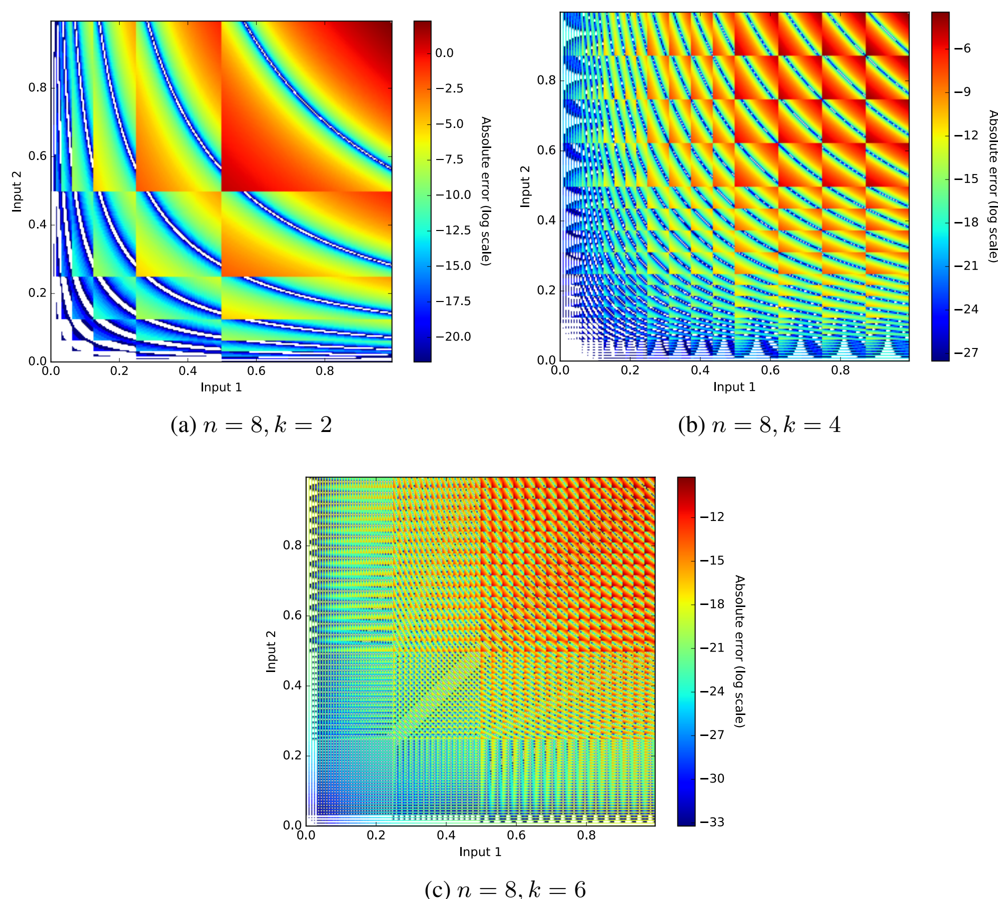
Table 1.31 – Error results for 16-bit DRUM for k between 3 and 8 – Error metrics are defined by Equations 1.42.

mean relative error (or relative error bias), and σ_{RE} the standard deviation of the relative error. S represents the set of all possible inputs and N_S the width of this set. It is important to notice that all these metrics are relative to the accurate output, as this is the family of metrics which is the most reliable when speaking about FlP-like error. Indeed, as the multiplier is "sliding" on the inputs, the error performed is always relative to the amplitude of the inputs. Table 1.31 gives the values of these metrics for the same 16-bit versions of DRUM as the ones of Figure 1.67.

As expected, the error is larger when k is smaller. For $k = 3$, maximum relative error reaches 56%, which is high but much smaller than most approximate operators which can have error propagated to their MSB in case of broken carry chain. Thanks to the unbiasing method, error bias is very low and lowers when k increases. The general amplitude of error, represented by $\mathrm{MA}_{\mathrm{RE}}$ and σ_{RE} is generally low.

As a conclusion, DRUM operator shows interesting area and power benefits, leveraging FlP multiplication style applied to fixed-significance data. As an example, for a 16-bit DRUM with $k = 6$, more than 60% area is saved and more than 70% of the power, while the error stays very tight. Indeed, the error bias is nearly zero, while the mean absolute relative error is 1.47% only, with a maximum at 6.31%. DRUM approximate operator is a scarce example of approximate operator with important savings and producing often-erroneous but low-amplitude

error outputs instead of producing scarce high-amplitude error. This is confirmed by error maps visible in Figure 1.68.



(a) $n = 8, k = 2$

(b) $n = 8, k = 4$

(c) $n = 8, k = 6$

Figure 1.68 – Error maps of 16-bit DRUM with different k

1.4.3 Final Discussion on Approximate Operators in Literature

In this chapter, besides classical floating-point and fixed-point paradigms, a subset of approximate operators were described, all chosen for being different the ones from the others and basing their designs on different techniques of integer addition and multiplication. The list dressed in this chapter is far from exhaustive but tried to cover the main stakes of approximate operators. The study of the existing literature also concludes in a more bitter observation: most presented operators do not come with enough results about their impact on real-life application. Some of them only come with stand-alone results using convenient metrics hiding high error spikes, and others are tested on applications too simple to make definitive conclusions. In this document, we intend to provide methods, tools and conclusions about the general advantages

and drawbacks of these operators which are all different the ones from the others.

It is also important to note that all the approximation techniques presented in this chapter are not exhaustive, though they are the main ones and lay the foundation for the remainder of this thesis. A more complete survey of existing approximation techniques at many levels can be found in [53]. The high number of existing techniques, which are enabled at different levels, from algorithm design to the physical layer, will need by the future to be unified in a single general technique allowing to take advantage of the best of each through cross-level design.

Chapter 2

Leveraging Power Spectral Density in Fixed-Point System Refinement

The first contribution of this thesis, after literature study and comparison of approximate operators in the previous chapter, is a novel method for system-level optimization in FxP arithmetic. This work led to a paper in the DATE'16 conference [1].

2.1 Motivation for Using Fixed-Point Arithmetic in Low-Power Computing

Signal processing applications popularly use fixed-point data types for implementation. The choice of fixed-point data types is driven usually by cost constraints such as power, area and timing. The objective of fixed-point refinement during the design process is to make sure that chosen data types are precise enough to achieve the expected quality of computation while minimizing the cost constraint. The acceptable lower quality of computation is because either there are error correction mechanisms explicitly defined as a part of the system or that the user perception defines the lower bound on the quality of output or both. For instance, video CODECs such as H.264 popularly used for wireless transmission allow a certain amount of errors on the channel, which can be corrected by error-resilience [54] or because the human eye is insensitive to some errors [55]. All these layers of error resilience mean that using approximations for concerned computations could represent significant gain of area, time and/or energy. A classical way to approximate a computation process is to use fixed-point arithmetic. Indeed, the representation of fractional numbers by integers insures faster and more energy-efficient arithmetical computations as discussed in previous chapter, and the design of the operators requires meaningfully less area. The most important drawback of using an approximated arithmetic is the need for managing the induced computation errors. The errors with fixed-point data types are classified into two types arising from finite precision on one hand and finite dynamic range on the other. Although the impact of errors due to violation of finite dynamic range is more pronounced, these errors can be mitigated by techniques such as range analysis using affine arithmetic, interval arithmetic or more complex statistical techniques such as [56]. In spite of allowing for good dynamic range, the lack of precision causes errors that are perceived as bad

quality of computation. In case of wireless applications, this can be measured as Bit-Error-Rate (BER), in image and signal processing as Signal-to-Quantization-Noise Ratio (SQNR), and, in general, as quantization noise power. Measuring the impact of finite precision on the output quality of computation is discussed in this Chapter.

Commercial tools for performing fixed-point accuracy include C++ fixed-point libraries (e.g. `ac_fixed` from Mentor Graphics used with Catapult-C, or `sc_fixed` from SystemC) or the Matlab fixed-point design toolbox. These tools are primarily based on facilitating FxP simulation with user-defined word-lengths using software FxP constructs and libraries. Although very useful, evaluation by simulation can be very time consuming. The time required for FxP evaluation grows in proportion with the number of FxP variables and also the number of input sample size.

Using the analytical approach for accuracy evaluation, the noise power is obtained by evaluating a closed-form expression as a function of the number of bits assigned to various signals in the system. This approach requires a one-time effort for arriving at the closed-form expression for a given system. These analytical techniques can be handy but are generally limited in applicability to linear and some types of non-linear systems (referred to as smooth operations). The analytical technique evaluates the first two moments of the quantization noise sources and propagates it through the Signal Flow Graph (SFG) from all noise sources to the system output. On relatively small systems, the evaluation of path functions can be accomplished manually. As the system complexity grows, it would require automation support. And possibly for very large systems, the automation could also prove painstakingly slow. Therefore, several divide and conquer approaches have been proposed [57, 58] to overcome the apparent complexity of large systems which respectively suffer from loss of information or enumerating all paths in the graph.

With the method described in this chapter, we provide an alternative analytical accuracy evaluation approach for use with hierarchical techniques to be applied on LTI systems. This technique captures the information associated with the frequency spread of quantization noise power by sampling its PSD. We show how such information can be used for breaking the complexity of evaluating quantization noise at the output of large signal processing systems. Contributions brought by our work are as follows:

- quantifying the accuracy of the proposed technique based on PSD propagation and
- demonstrating its high scalability at system level resulting from linear time complexity.

The rest of this chapter is organized as follows. Section 2.2 reviews analytical methods for accuracy analysis at the algorithm level of errors due to finite arithmetic effects in systems using fixed-point arithmetic. In Section 2.3, the proposed estimation method based on PSD is introduced and developed for general systems. Finally, in Section 2.4, two representative signal-processing benchmarks are chosen to showcase the efficiency of the proposed method.

2.2 Related work on accuracy analysis

The loss in accuracy due to finite precision imposed by fixed-point numbering format has been evaluated using several metrics. The most common among them are the error bounds and the

Mean Square Error (MSE). While the first metric is used to determine the worst case impact, the MSE is an average case metric very useful in tuning the average performance of the system under consideration in terms of its energy and timing. Although the finite precision accuracy must be compared with infinite precision (or arbitrary precision) numbers, it is impossible to do so while simulating using a computer. So, the IEEE double-precision floating-point format, whose dynamic range and precision are several orders of magnitude higher compared to typical fixed-point word lengths, is considered as the reference for all comparison purposes and may be referred as *infinite precision* in what follows.

In the literature, the MSE is the mean square value of the differences between computations by fixed-point system and the reference system implementation extracted as showed on Figure 2.1 and is also referred to as quantization noise power. This is a scalar quantity and it changes as a function of the FxP word-length. Evaluation of quantization noise power at the output of a fixed-point system is either performed by simulation-based technique or using analytical techniques. Simulation-based techniques are universal and can be made use of as long as there are enough computational resources. By the nature of it, simulation-based techniques take longer time and are subjected to the input stimulus bias.

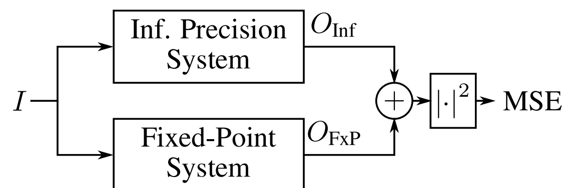


Figure 2.1 – Extraction of the mean square error of a fixed-point system

Analytical techniques, on the other hand, provide a closed-form expression for calculating the quantization noise power as a function of FxP word-lengths. However, they are limited due to their dependence upon the following properties [59]:

1. Quantization noise and the signal are uncorrelated.
2. Quantization noise at its source is spectrally white.
3. Effect of a small perturbation at the input of the operation generates a linearly proportional perturbation at the output of the operation.

The first two properties pertain to the quantization noise source under conditions defined in the Pseudo-Quantization Noise (PQN) model, the statistics of the noise and signal are uncorrelated and even though the signal itself may be correlated in time, the noise signal is uncorrelated in time [59]. The independent and uniform nature of this noise was already discussed in Section 1.3.2. The representation of quantization error as an additive uniformly distributed white noise is depicted by Figure 1.6.

The third property relates to the application of "perturbation theory" [60]. It is possible to propagate quantization noise through as long as the function defined by the operation can be linearized. Consider a binary operator whose inputs are x and y and the output is z. If the input

signals be perturbed by b_x and b_y to obtain $\overline{x}$ and $\overline{y}$ respectively, the output is perturbed by the quantity b_z to obtain $\overline{z}$. In other words, as long as the fixed-point operator is smooth, the impact of small perturbations at the input translates to perturbation at the output of the operator without any change in its macroscopic behavior. In the realm of perturbation theory, the output noise b_z is a linear combination of the two input noises b_x and b_y such as

$$b_z = \nu_1 b_x + \nu_2 b_y \tag{2.1}$$

where ν_1, ν_2 are obtained from a first-order Taylor approximation [60] of the continuous and differentiable function f:

$$\begin{aligned} \overline{z} &= f(\overline{x}, \overline{y}) \\ &\simeq f(x,y) + \frac{\partial f}{\partial x}(x,y).(\overline{x} - x) + \frac{\partial f}{\partial y}(x,y).(\overline{y} - y). \end{aligned} \tag{2.2}$$

Therefore, the expression of the terms ν_1 and ν_2 are given as

$$\nu_1 = \frac{\partial f}{\partial x}(x,y) \qquad \nu_2 = \frac{\partial f}{\partial y}(x,y). \tag{2.3}$$

Following the third property of quantization noise enumerated above, a further assumption for Equation 2.1 to hold true is that the noise terms b_x and b_y are uncorrelated with one another. It has to be noted here that the terms ν_1 and ν_2 can be time varying. This method is not limited to binary operations only. In fact, this method can be applied at the functional level with any number of inputs and outputs and to all operators on a given data path in order to propagate the quantization noise from all error sources to the output.

When above conditions hold true, the output quantization noise power of the system is obtained by linear propagation of all quantization noise sources [61] as

$$E\left[b_y^2\right] = \sum_{i=1}^{N_e} K_i \sigma_i^2 + \sum_{i=1}^{N_e} \sum_{j=1}^{N_e} L_{ij} \mu_i \mu_j \tag{2.4}$$

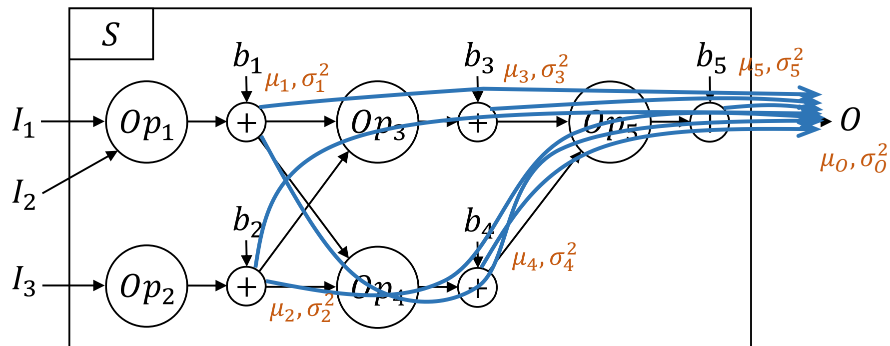
where $E\left[\cdot\right]$ is the expectation function, b_y is the error signal associated with its corresponding system output signal y. The system under consideration consists of N_e fixed-point operations and the ith operation is generating quantization noise b_i with mean and standard deviation μ_i and σ_i. Figure 2.2a illustrates this noise propagation. The terms K_i and L_{ij} are constants and depend on the path function h_i from the ith source to the output y and are calculated as

$$K_i = \sum_{k=-\infty}^{\infty} E\left[h_i^2(k)\right], \tag{2.5}$$

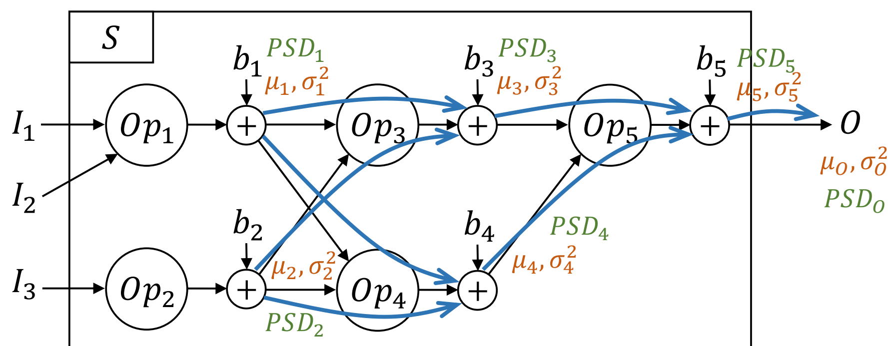
$$L_{ij} = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} E\left[h_i(k) h_j(l)\right]. \tag{2.6}$$

Hierarchical techniques for evaluation of quantization noise power have been proposed [62, 57] to overcome the scalability concerns associated with fixed-point systems. In this approach, the

system components are evaluated one at a time and then combined by superposition at the output (Figure 2.2b, blind propagation of μ_i, σ_i^2). If simulation-based technique is used for evaluation of quantization noise power at the output, the hierarchical evaluation process helps parallelize simulation of each of the components. When employing analytical technique such as the one in Eq. 2.4, the number of paths required to be evaluated is reduced dramatically. This reduction is very interesting from the design automation perspective. The paths are broken around the system component boundaries and each component can be evaluated separately thereby reducing the burden of semantic analysis. However, it has to be borne in mind that the application of the technique in Equation 2.4 requires that the quantization noise satisfies the three properties enumerated above and also that the noise signals are always uncorrelated, which is often false and can cause severe errors. The method in this chapter addresses this problem and suggests a technique that exploit the information hidden in the PSD of the quantization noise [59, 63] signal to achieve very accurate estimates.



(a) Traditional flat method: propagation of μ_i, σ_i^2 from each noise to output



(b) PSD agnostic: blind propagation of μ_i, σ_i^2. Proposed PSD method: propagation of μ_i, σ_i^2, and PSD_i

Figure 2.2 – Comparison of noise parameters propagation using traditional flat, PSD agnostic and proposed PSD methods

2.3 PSD-based accuracy evaluation

It is clear from the state of the art that there exists two types of limitations to the existing accuracy evaluation techniques. While the analytical technique reduces the simulation time greatly, its preprocessing time can grow exponentially requiring to employ hierarchical techniques such as [57]. However, these techniques introduce the problem of inaccuracy by approximating error quantities with just mean and variance. This is especially true in cases when large systems are broken down to smaller sub-systems for analysis. For illustration, consider the system shown in Figure 2.2b. The system S consists of several sub-systems marked as $Op_{1...5}$. The noise generated at the output of each system, correspondingly marked as $b_{1...5}$, is propagated (blue arrows) through several parts of the system for calculation of the moments of error at the system output. Suppose there are memory elements in Op_1 and Op_2, propagation of noise b_1 and b_2 (say) through Op_3 by just using the first two moments of the quantization noise (as described in the previous section) can lead to errors in estimates at the output of Op_3 which can further be amplified by Op_5 all the way to output O. Similarly, the path through Op_4 also influences the error of the estimate through Op_5 leading to very large error margins for O. In order to analytically arrive at the moments of the system output, additional information pertaining to quantization noise at points of convergence of two or more noise paths is required. We refer to the methods that do not consider PSD information (such as [62]) as PSD-agnostic methods. In this section, we propose a technique which efficiently makes use of the PSD of the quantization noise for evaluating the error at the output of a system and which is scalable both in terms of accuracy and system size.

2.3.1 PSD of a quantization noise

A large signal processing system can be divided into a number of sub-systems, each characterized by its transfer function. The transfer function defines the magnitude and phase relationship of the path for input signals of different frequencies. Since our interest is only the noise power, we ignore the phase spectrum and consider only its magnitude spectrum or the PSD. With the knowledge of the PSD distribution of the input and the system PSD profile, it is possible to calculate the PSD of the output. The PSD $S_{xx}(F)$ of a signal x at any normalized frequency F is defined as the Fourier transform ($\mathcal{F}\{\cdot\}$) of the autocorrelation function of x as

$$\begin{aligned} S_{xx}(F) &= \mathcal{F}\{x(n) \cdot x^{*}(n+m)\}, & (2.7)\\ S_{xx}(F) &= \mathcal{F}\{x\} \cdot \mathcal{F}\{x\}^{\star} = |\mathcal{F}\{x\}|^{2}. & (2.8)\end{aligned}$$

With the knowledge of the PSD of x, the MSE and the mean of x is obtained by summing up the power in each frequency component as

$$\begin{aligned} E\left[x^2\right] = \int_{-1}^{1} S_{xx}(F)\, dF &= \mu^2 + \sigma^2 \\ S_{xx}(0) &= \mu^2. \qquad (2.9)\end{aligned}$$

The PSD of the quantization noise generated by a fixed-point data type with d fractional bits is (as discussed in Section 2.2) white, except for $F = 0$, which depends on mean. By discretizing the PSD into N_{PSD} regular bins including the DC component, the PSD of a generated

quantization noise b_x is given by:

$$S_{b_x}(F) = \begin{cases} \frac{1}{N_{\text{PSD}}}\sigma^2 & \text{if } F \neq 0, \\ \mu^2 & \text{if } F = 0. \end{cases} \tag{2.10}$$

where mean and variance μ and σ^2 for both truncation and rounding modes with d bits is as given in [59].

2.3.2 PSD propagation across a fixed-point LTI system

In the method developed in this chapter, we will focus on linear and time-invariant (LTI) systems, which constitute the major part of signal processing systems. An LTI system can be represented by a signal flow graph (SFG) composed of boxes corresponding to sub-systems defined by their impulse response and delimited by additive quantization noise sources such as the one described in Section 2.2. The proposed PSD evaluation method then consists of three steps:

1. Detect cycles in SFG and break them to obtain an equivalent acyclic SFG that can be used for noise propagation using classical SFG transformations [64]. An example of SFG breaking is issued by Figure 2.3. Given the original cyclic SFG of Figure 2.3a, the loop generated by H_3 loopback is flattened as showed on Figure 2.3b.

2. The discrete PSD of each signal processing block and of the additive noise associated with the input signal is calculated on N_{PSD} points.

3. The noise PSD parameters are propagated from inputs to outputs, using Equations 2.11 and 2.14.

Let x be the input of a system of impulse response h. Then the output y is obtained by the convolution operation ($*$) of x and h as $y = x * h$. In the Fourier transform domain it can be written as $Y = X \cdot H$ where $Y = \mathcal{F}\{y\}$. Following this, the output PSD $S_{yy}(F)$ is obtained as [63]

$$S_{yy}(F) = S_{xx}(F) \cdot \|H(F)\|^2. \tag{2.11}$$

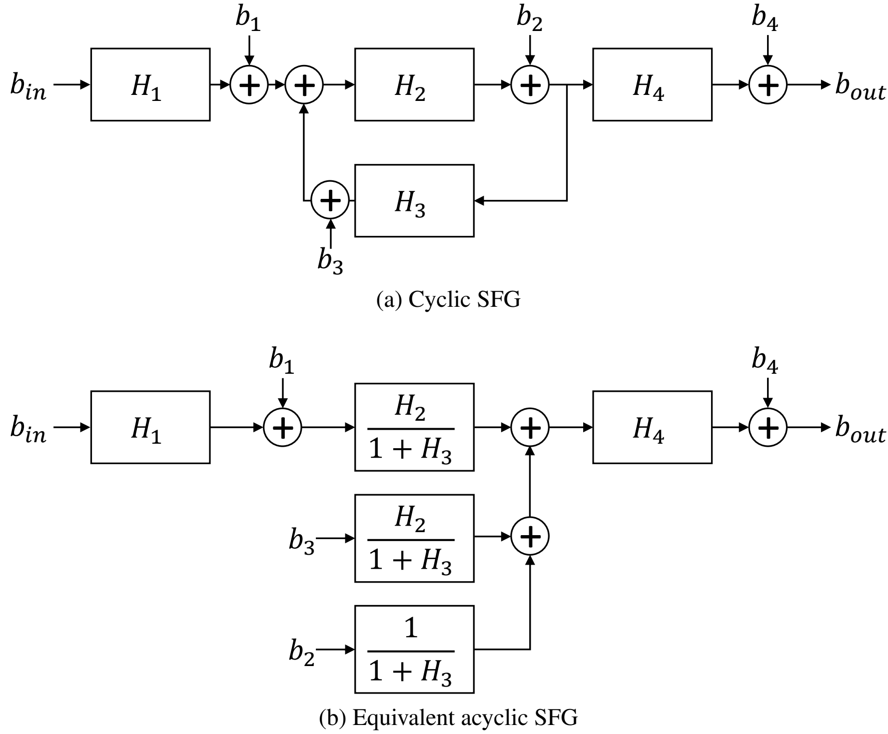
where $\|H(F)\|$ is the magnitude response of the system h.

In any signal processing system, the quantization noise sources from various inputs converge in at either an adder or a multiplier. Considering the LTI subset, multiplications are nothing but multiplication with constants and hence correspond to linear scaling factors for noise powers. In the case of adders, if the sum of two quantities x and y is obtained as $z = x + y$, then $S_{zz}(F)$ is given by

$$S_{zz}(F) = S_{xx}(F) + S_{yy}(F) + S_{xy}(F) + S_{yx}(F), \tag{2.12}$$

where $S_{xy}(F)$ is obtained using the cross-correlation spectrum of x and y and is obtained as

$$S_{xy}(F) \quad = \quad \mathcal{F}\{x(n) \cdot y^*(n+m)\}. \tag{2.13}$$



(a) Cyclic SFG

(b) Equivalent acyclic SFG

Figure 2.3 – SFG cycle breaking process example

Also, $S_{yx}(F)$ is obtained as the complex conjugate of $S_{xy}(F)$. Indeed, if x and y are uncorrelated, the cross-correlation is rendered zero and $S_{zz}(F)$ is simply the sum of $S_{xx}(F)$ and $S_{yy}(F)$.

$$S_{zz}(F) = S_{xx}(F) + S_{yy}(F). \tag{2.14}$$

The complexity of propagating PSD parameters through the system essentially depends on the number of discrete points N_{PSD}. The total time for evaluation of the PSD parameters can be split into two parts: first, τ_{pp} corresponding to the preprocessing stage which involves evaluating the N_{PSD}-point Fourier transform of transfer function of the sub-systems with complexity $\mathcal{O}\{Nlog(N)\}$; second, the actual time required for evaluation τ_{eval} which is $\mathcal{O}\{N\}$ from Equations 2.11 and 2.14. τ_{eval} is required for evaluating the accuracy for various inputs and can be repeatedly performed without any preprocessing say N_{eval} times. Since the time spent on preprocessing is a one-time effort, the actual evaluation time is dominated by the τ_{eval} which is linear with N_{PSD}.

2.4 Experimental Results of Proposed PSD Propagation Method

In this section, the proposed method is evaluated using a three-step approach. First, we show experimentally that the estimates obtained by proposed PSD technique are close to simulation.

Then, we present the impact of choosing the number N_{PSD} to capture PSD information on the accuracy as well as the execution times of the proposed approach. Finally, we also discuss the improved accuracy in estimation and compare it with the result obtained by PSD agnostic method.

All experiments are performed using Matlab R2014b. The MSE deviation E_d is chosen as the metric for comparison in all these experiments. It is calculated as

$$E_d = \frac{E\left[err_{sim}^2\right] - E\left[err_{est}^2\right]}{E\left[err_{sim}^2\right]}, \tag{2.15}$$

where $E\left[err_{sim}^2\right]$ is the output error power obtained by simulation and $E\left[err_{est}^2\right]$ is obtained by proposed analytical estimation. From this metric, an accuracy equivalent to less than one bit corresponds to the range $E_d \in (-75\%, 300\%)$, which can be trivially proven considering the error power relative to two successive word-lengths. Beyond these limits, the estimation is unmistakably not suitable for the fixed-point refinement process as the finally selected word-length would not meet the maximum error requirements. In the following sections, we first present the experiments and provide a discussion of the results obtained.

2.4.1 Experimental Setup

2.4.1.1 Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) Filters

The first experiment consists in evaluating the PSD of a single Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filter blocks as described in Section 2.3. The quantized input signal is propagated through the chosen filter and the output quantization noise power is measured by simulation and by the proposed PSD method. The error in estimates of the noise power E_d is obtained on a total of 147 FIR and 147 IIR filters obtained by attributing different functionalities (bandpass, low-pass and hi-pass), various taps involving memory elements between 16 and 128 taps for FIR filters and from 2 to 10 taps for IIR filter. Simulation is run on 10^6 inputs and PSD estimation is performed on 1024 samples.

2.4.1.2 Frequency Domain Filtering

The system described in Figure 2.4 is a frequency domain band-pass filter. It consists of a 16-tap low-pass FIR filter H_{hp} followed by a frequency-domain filter, composed of a 16-point FFT block, a multiplication by the 16 coefficients of a high-pass FIR filter H_{lp} and an inverse FFT. The frequency domain filter applies the filter using the popular overlap save method. Simulations are carried out on a set of 10^7 input samples.

2.4.1.3 *Daubechies* 9/7 Discrete Wavelet Transform

A 2-level *Daubechies 9/7 Discrete Wavelet Transform (DWT)* pair which forms the basis of many modern image and video codecs such as JPEG-2000, H.264 is shown in Figure2.5. For this experiment, 196 grayscale images extracted from USC-SIPI and RPI-CIPR image

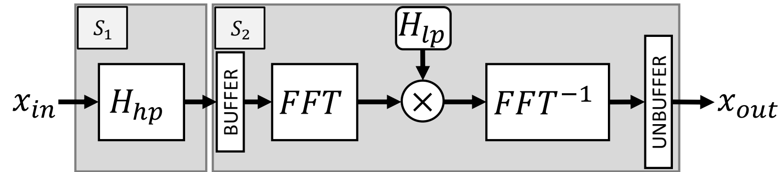


Figure 2.4 – Band-pass frequency filtering scheme

databases and from Brodatz texture images [65] used generally for evaluating JPEG2000 compression algorithms. Two levels of sub-band decomposition are performed on the sample images using the hierarchical signal flow graph. For the encoder, the first level of filtering and downsampling is applied on rows and the second one on columns. The second level coding is applied on the low-pass components (x_{ll}). Symmetrically, the decoder first performs upsampling and filtering is applied on columns followed by the second upsampling and filtering on the rows. For this experiment, fractional word-lengths d of all variables are set to the same value and are varied across $8 - 32$ bits in steps of 4 and N_{PSD} is set to 1024.

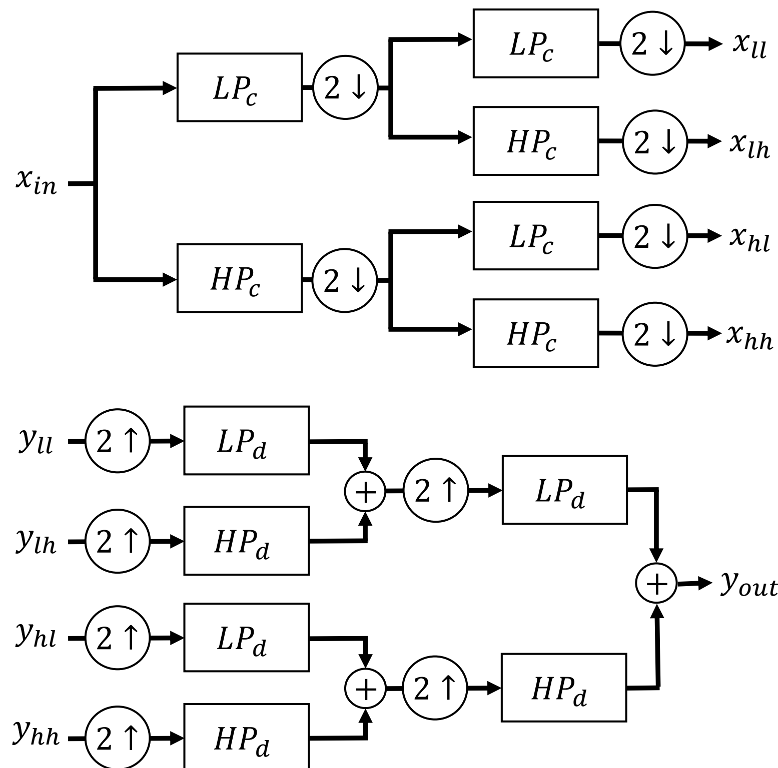


Figure 2.5 – 1-level DWT coder and decoder

2.4.2 Validation of the Approach for LTI Systems

The min, max and $absolute\ mean$ of E_d for FIR and IIR filters are given in Table 2.1.

	FIR filters	IIR filters
$min(E_d)$	-0.37%	-19.4%
$max(E_d)$	0.37%	31.2%
$mean(\|E_d\|)$	0.11%	9.44%

Table 2.1 – Relative error power estimation statistics E_d

In the case of FIR filters, E_d is contained within an absolute value of 0.37% in comparison with simulation. In the case of IIR filters, Ed bounds are higher because of their recursive nature and the high filter orders tested. FIR and IIR filters result in an average absolute E_d of respectively 0.11% and 9.44%, showing a generally very accurate estimation. For both, the accuracy is anyway largely less than one-bit equivalent. Moreover, classical flat estimation [61] applied to the same filters gives $the\ exact\ same\ results$ in terms of E_d, showing their strict equivalence on an elementary filtering block.

Figure 2.6 presents the results for the two other experiments when the number of fractional bits are changed between 8 and 32 bits with a maximum deviation in error of only about 10%. The maximum error in estimate is by far too small to have an impact on the final optimization.

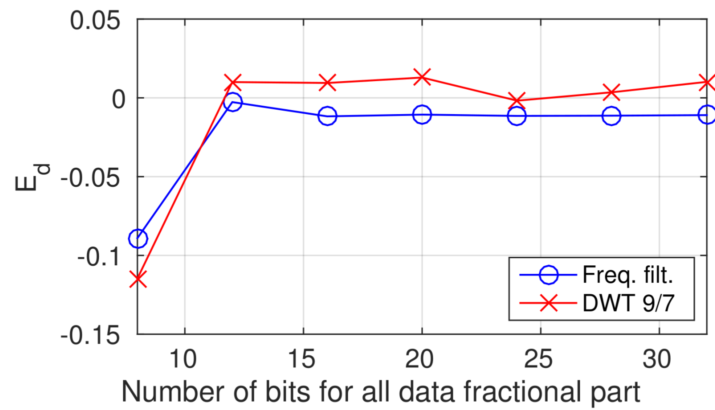


Figure 2.6 – E_d versus fractional bit-width d

2.4.3 Influence of the Number of PSD Samples

The proposed PSD estimation method achieves very good accuracy with a large number of sampling PSD samples. However, as discussed in Section 2.3.2, a larger number of N_{PSD} samples increases the evaluation time. Therefore, it would be of interest to know the impact of finding out how this choice affects the estimation accuracy. To observe this, in both examples chosen in this section, the fixed-point error is obtained by both simulation and the proposed

PSD method with different values of N_{PSD} in powers of 2 ranging from 16 to 1024. In this example, fractional bit-width d is uniformly set to 32 for all signals. Output error power deviation E_d value for this experiment is plotted in Figure 2.7 versus N_{PSD}. As expected, increasing the number of PSD samples leads to an improvement of E_d. For $N_{\text{PSD}} = 16$, E_d is slightly inferior to -8% for the frequency filtering system, and slightly superior to 1% for the DWT system. Then, both curves tend to a value inside $\pm 1\%$. The accuracy obtained is better than the sub-one-bit objective. The accuracy of estimates obtained using the proposed method is a function of the system complexity.

2.4.4 Comparison with PSD-Agnostic Methods

The deviation of the error estimates between the proposed and the PSD agnostic method is presented in Table 2.2. The $\max$ error is obtained with $N_{\text{PSD}} = 16$ and $\min$ error is obtained with $N_{\text{PSD}} = 1024$. In all cases, it can be observed that the PSD agnostic method is much more erroneous than even the maximum error obtained using the proposed technique. It has to be noted that for the DWT example, the PSD agnostic method renders an error of 610%. The PSD agnostic method is $4.5\times$ worse off in its estimate for frequency filtering, and $554\times$ for DWT. For the best case, these values raise respectively to 3.5 $10^3\times$ and 6.7 $10^4\times$.

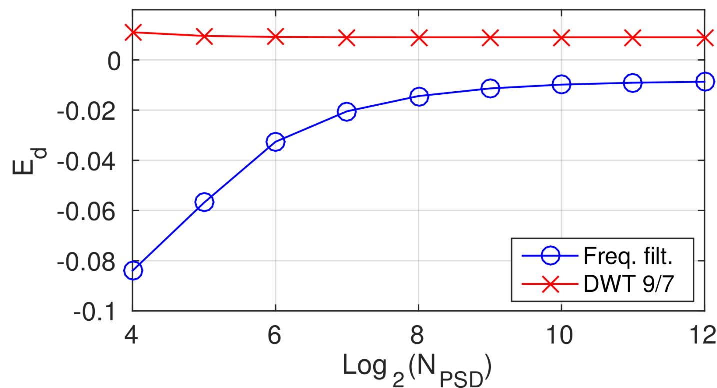


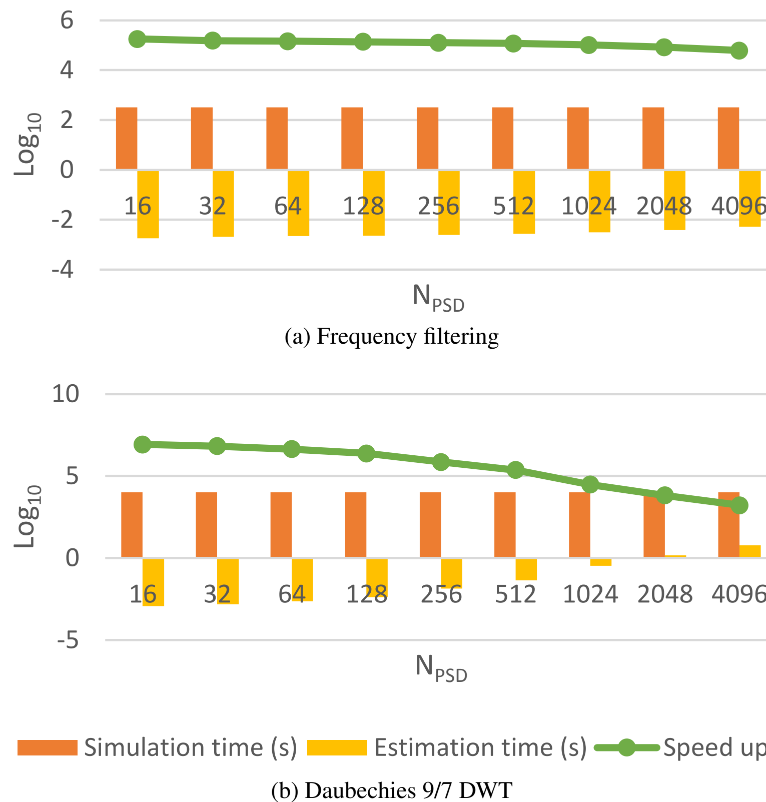
Figure 2.7 – E_d versus number of PSD samples N_{PSD}

	Proposed PSD method (max accuracy)	Proposed PSD method (min accuracy)	PSD agnostic method
Freq. Filt.	-8.40%	-0.87%	29.5%
DWT 9/7	1.10%	0.90%	610%

Table 2.2 – Comparison of E_d between PSD agnostic method and proposed PSD method

Time spent on this estimation is usually another critical resource. Figure 2.8 gives the time of output error estimation using the proposed PSD method versus N_{PSD}. With $N_{\text{PSD}} = 16$ the

proposed method requires about one millisecond in case of both experiments. With more PSD samples, the time taken by frequency filtering example grows slower than Daubechies DWT example owing to its small size. A speed-up factor of $3-5$ orders of magnitude compared to simulation is obtained in both cases even for the highest value of N_{PSD}.



(a) Frequency filtering

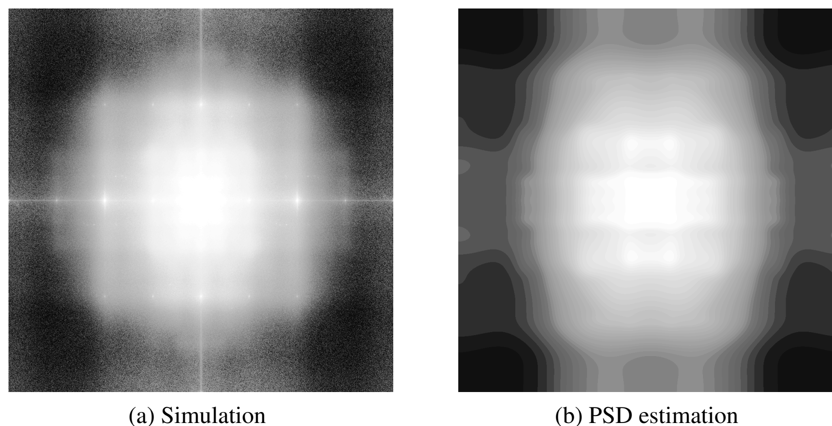
(b) Daubechies 9/7 DWT

Figure 2.8 – Execution time in seconds and speed up for frequency filtering and DWT systems versus the number of PSD samples

2.4.5 Frequency Repartition of Output Error

Another interesting feature inherent to the proposed estimation method is to know the frequency repartition of errors, which is relevant for refinement of fixed-point signal processing systems, and which is not estimated with conventional methods. Indeed, the classical flat method is not able to give any clue about the frequency repartition of the error, which is a capital information in signal processing. E.g, for image compression for instance, accuracy is more likely to be relaxed in low frequencies than in high ones, human vision being less sensible to slight variations of colors than in tiny details. The proposed PSD estimation is able to give the frequency repartition of this error in a very precise and fast way. Figure 2.9 gives a visual comparison between the PSDs of output error obtained by intensive simulation and PSD method on 1024 samples for a 2-level Daubechies DWT encoding and decoding, with all

data fractional parts set to 12 bits. Black to white values represent log-normalized low to high errors. The center of the image represents low frequencies, while the borders represent the high ones. These visual representations show that proposed method achieves a very good estimation of frequency repartition of the output error, taking only a few milliseconds with PSD method whereas simulation on 72 grayscale images has taken several hours of computation using Matlab. Such a fast and accurate information can be used for refining the system word-lengths to reach a better output quality, basing the refinement not only on output error intensity but also on what frequency repartition is best for the application. Using PSD method, different versions of the application can be evaluated in terms of error frequency repartition to allow for code transformations leading to less impact in the relevant frequency bands. Frequency repartition information can then be modified by allowing the introduction of errors in one or several given parts of the system to identify.



(a) Simulation (b) PSD estimation

Figure 2.9 – Output frequency repartition of the fixed-point error after DWT encoding and decoding

2.5 Conclusions about PSD Estimation Method

This chapter described the characterization and propagation of quantization noises in a fixed-point signal processing system using its power spectral density. This method is applied at block level, which dramatically reduces the complexity of fixed-point system evaluation when compared to classical flat estimation method. It therefore leads to a significant speed up for accuracy evaluation, going from 3 to 5 orders of magnitude when compared to Monte-Carlo simulation in tested examples. Results demonstrate that the proposed estimation method leveraging spectral information achieves a less than one-bit accuracy with a large margin. They also show that complexity-equivalent PSD-agnostic techniques evaluate the accuracy with large errors. The proposed PSD technique also allows the observation of useful frequential properties of the output error that could not be achieved with conventional scalar methods. This work was published at DATE'16 conference [1].

Chapter 3

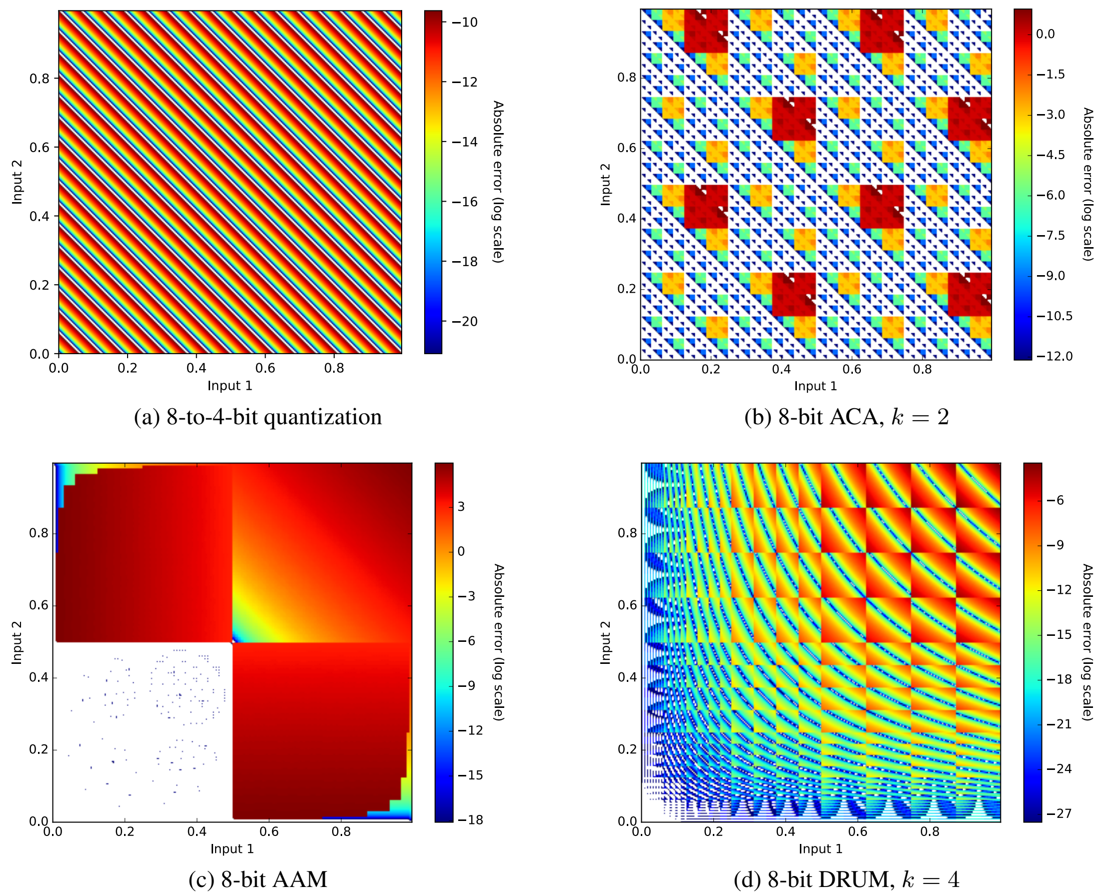
Fast Approximate Arithmetic Operator Error Modeling

In this chapter, techniques based on propagation of Bit-Error Rate (BER) are presented. First, the bitwise-error rate propagation method is proposed and applied to approximate operators. This method allows for fast analytical propagation of approximate operators error, with low memory cost. The model is trained by simulation and converges fast. Then, attempts to use approximate operators for the simulation of Voltage Over-Scaling (VOS) effects are discussed.

3.1 The Problem of Analytical Methods for Approximate Arithmetic

As pointed out in Chapter 1, Section 1.4, many different approximate operators do exist. Most adders rely on different ways to break the carry chain, like LPA, ACA, ETA version II to IV, and most multipliers by pruning the partial products to simplify the summand grid reduction, such as AAM version I to III and Fixed-width modified-Booth-encoded Multiplier (FBM) version I to III. Only a few examples such as ETAI or DRUM use strongly different techniques. Amongst these operators, some are configurable at run time like AC2A and GDA.

We discussed in Chapter 2 the importance of optimizing a computing system so the operators with the lowest cost meeting accuracy requirements are used so no time, area and/or energy is uselessly spent. For FxP error, modeling the error as an additive uniform white noise allows an efficient propagation of the mean and variance of the noise [24, 25]. However, the nature of approximate operators error is very different from FxP noise. To account for this specifity, Figure 3.1 shows the error maps of several different 8-bit approximate operators previously mentioned. All error maps take as a reference an 8-bit exact adder. The uniform nature of 4-bit reduction FxP noise illustrates with a very regular striped-pattern on the error map 3.1a, whereas all the others are very different. ACA error map 3.1b shows a fractal behavior, with nested error triangles. AAM error map 3.1c has four areas with very different error amplitude and patterns. Finally, DRUM error map 3.1d, with its floating-point-like behavior, has an error pattern which is transformed depending on the amplitude of the inputs.



(a) 8-to-4-bit quantization

(b) 8-bit ACA, $k = 2$

(c) 8-bit AAM

(d) 8-bit DRUM, $k = 4$

Figure 3.1 – Error maps of 8-bit FxP quantization process and approximate operators. 8-to-4-bit quantization illustrates the regularity of the uniform error repartition of FxP arithmetic. The error maps of the three other approximate operators illustrate that the nature of their error is far more complex.

All these differences between Approximate (Apx) operators and FxP, and between different Apx operators themselves, make it hard to find pure analytical models to estimate their impact on the output error of an application. The only efficient way to estimate their impact is therefore simulation. Hybrids between analytical models and simulation, referred as pseudo-simulation, have been developed, but they are inefficient because often heavier and less accurate than simulation. In [66], an analytical propagation of the error Probability Density Function (PDF) of approximate operators is proposed. It leverages modified interval arithmetic, representing and propagating the PDF of signal, signal error and operators error by sets of intervals. This method allows fast simulation compared to Monte Carlo simulation. However, the method has a major limitation. Indeed, the model for the propagation of error PDF, which is specific to each operator, potentially costs a lot of memory. Indeed, given 2 input error PDFs with k intervals each, k^2 resulting values must be kept in memory. However, to be accurate, k must be large enough to be representative of the real error PDF. For an n-bit value, a perfect accuracy for the representation of a corresponding error PDF must have $k = 2^n$. Therefore, for a 32-bit operator, a perfect accuracy would require $2^{2*32} = 2^{64}$ values to be stored, that is 10^{19}. Of course, a much lower value must be chosen for k, which implies important approximations in the PDFs, and also in the model of propagation. Therefore, this model is likely to diverge quite fast along propagations, or to be memory-hungry.

In the next section a proposition of a lighter model to propagate the Bitwise-Error Rate (BWER) caused by approximate operators is presented.

3.2 Bitwise-Error Rate Propagation Method

This section presents the Bitwise-Error Rate (BWER) propagation method. First, the main principle of BWER propagation method is described. Then, the data structure used for propagation and the training algorithm are discussed. Finally, the propagation algorithm is described.

3.2.1 Main Principle of BWER Propagation Method

BWER propagation method is an analytical method which consists in estimating the output BWER of a system composed of approximate integer operators. BWER is defined as the BER associated to each bit position of a binary word. Given an n-bit binary word $x = \{x_i\}_{i \in [\![0,n-1]\!]}$, BWER is the vector $\mathrm{BWER}(x) = \{p_i\}_{i \in [\![0,n-1]\!]}$ composed of n real numbers in range $[0, 1]$ corresponding to the probability p_i for x_i to be erroneous. Given an approximate operator Op whose inputs are x and y of width n and whose output is z of width m, BWER propagation method aims at determining analytically the output BWER vector, knowing both input BWER vectors such as depicted on Figure 3.2. Then, considering a network of approximate operators,

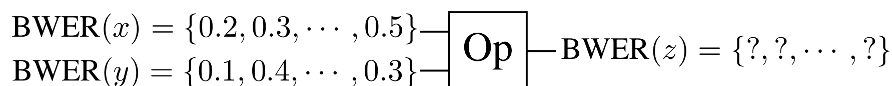


Figure 3.2 – Propagation of BWER across an operator

the BWER vectors can be propagated operator by operator from the inputs, considered as accurate, to the outputs. To be time-efficient, the propagation must be simulation-free and so the model must be completely analytic.

3.2.2 Storage Optimization and Training of the BWER Propagation Data Structure

To propagate BWER across an operator, a BWER transfer function must be built. For this, the impact of an error at any bit position of both inputs on any bit of the output must be determined. Let $e_{x,i}$ be the event "The i-th position bit of x is erroneous". Considering n-bit inputs x and y, let the vector

$$E_{x,y,\mathrm{err_id}} = \{e_{y,n-1}, e_{x,n-1}, e_{y,n-2}, e_{x,n-2}, \ldots, e_{y,1}, e_{x,1}, e_{y,0}, e_{x,0}\} \tag{3.1}$$

be the event "x and y to have all their bits erroneous". In this notation, err_id is the integer value represented by the binary word $E_{x,y,\mathrm{err_id}}$ where the event $e_{x,i}$ is represented by 1 and the opposite event $\overline{e_{x,i}}$ is represented by 0, read left to right from MSB to LSB. E.g, if x and y are 2-bit inputs, $E_{x,y,3}$ represents the event vector $\{e_{y,1}, e_{x,1}, \overline{e_{y,0}}, \overline{e_{x,0}}\}$, and $E_{x,y,6}$ represents $\{e_{y,1}, \overline{e_{x,1}}, \overline{e_{y,0}}, e_{x,0}\}$. For inputs of width n, there are 4^n possible event vectors, $E_{x,y,0}$ being the vector for which all input bits are correct, and $E_{x,y,4^n-1}$ being the one for which all input bits are erroneous, which is Equation 3.1. From this point, these vectors will be referred as Error Event Vectors (EEVs).

Therefore, to know the impact of an error on any input bit for n-bit inputs on m-bit output, the set of probabilities must be determined and stored:

$$P\left(e_{z,j} | E_{x,y,i}\right)_{i \in [\![0,4^n-1]\!], j \in [\![0,m-1]\!]} \tag{3.2}$$

This set has a size of $m \times 4^n$ real numbers. The cost for storing this data in memory considering single-precision floating-point representation is given by Table 3.1 for different operations. It

Operation	n	m	Storage
8-bit addition	8	9	2.4 MB
8-bit multiplication	8	16	4.2 MB
16-bit addition	16	17	292 GB
16-bit multiplication	16	32	550 GB

Table 3.1 – Storage Cost of BWER Propagation Full Data Structure

is clear that storing such an amount of data is not scalable, even for small bit-width such as 16-bit. Moreover, the time that would be needed to train such a volume of data would also be huge. Therefore, reductions of this data structure are necessary.

First, for arithmetic operators, the output bits of significance j only depend of input bits of significance $i \leq j$. This already allows for an important reduction in the required storage. Indeed, the number of data needed to be stored is now $(m - n + \frac{1}{3})4^n - \frac{1}{3}$ instead of $m \times 4^n$, as the set of conditional probabilities to be stored of Equation 3.2 is now:

$$P\left(e_{z,j} | E_{x,y,i}\right)_{i \in [\![0,4^{\min(j,n)}-1]\!], j \in [\![0,m-1]\!]} \tag{3.3}$$

As an example, for 16-bit addition, 22 GB are necessary instead of the previous 292 GB. In spite of a 92% memory reduction, this is still too high to be decently implemented.

The previous reduction implies no approximation. However it is possible to reduce dramatically the size of the data structure, allowing a small amount of inaccuracy. For this, the hypothesis that any output bit of significance j only depends at most on the input bits of significance $i \in [\![j-k+1, j]\!]$, where k is arbitrary. With this method, the set of conditional probabilities of Equation 3.3 becomes:

$$P\left(e_{z,j}|E_{x,y,i}\right)_{i\in[\![0,4^{\min(j,k)}-1]\!],j\in[\![0,m-1]\!]} \tag{3.4}$$

This approximation is legitimated by two facts:

1. As already stated, the probability for a carry chain to be long is very small [32]. Therefore, a low significance input bit only has an impact on a much higher significance output bit in a very small minority of cases.

2. A vast majority of approximate arithmetic operators are based on cutting carry propagations to a certain limitation l. Therefore, choosing $k > l$ even induces no approximation at all.

Choosing an arbitrary k reduces the storage cost to

$$(m-k+\frac{1}{3})4^k - \frac{1}{3}$$

real numbers. Table 3.2 gives the corresponding memory cost as a function of k for 16-bit addition and multiplication. For 16-bit addition, the data structure size is reduced from 292 GB

Operator	Original	1st Reduction	2nd Reduction				
			$k=10$	$k=8$	$k=6$	$k=4$	$k=2$
16-Add	292 GB	22 GB	31 MB	2.4 MB	186 kB	14 kB	980 B
16-Mul	550 GB	280 GB	93 MB	6.3 MB	431 kB	29 kB	1.9 kB

Table 3.2 – Storage Cost of BWER Propagation Data Structure for 16-bit Addition and Multiplication Depending on Reduction Method

to 2.4 MB for instance with $k = 8$, limiting the consideration of input LSBs to a horizon of 8, turning the storage of the data structure to a decent value. Moreover, knowing the parameters of each considered operator, k can be minimized so there is no approximation in the model. E.g, for $\text{ACA}_{16}(5)$, taking $k = 6$ can be chosen with no compromise, requiring only 186 kB of storage.

Once k is selected, the model needs to be trained offline. Monte Carlo simulation with fault injection is used for this, using functional model of the operator in C++, from the approximate operator library of *ApxPerf 2.0* [2]. The extraction process of an observation of the EEV E_z corresponding to an observation of an input EEV $E_{x,y}$ is depicted in Figure 3.3. x and y inputs are randomly picked using Monte Carlo simulation, and the accurate operation Op_{Acc} is

performed to obtain the corresponding exact output $z = z_{j\,j \in [\![0,m-1]\!]}$. In parallel, random faults are injected in x and y performing an XOR with fault injection vectors. These fault injection vectors produce the $2n$-bit observation $F_{x,y}$ of an EEV $E_{x,y}$. The approximate operation $\mathrm{Op}_{\mathrm{Apx}}$ is then fed with the generated faulty inputs $\hat{x}$ and $\hat{y}$, returning $\hat{z}$. Finally, the m-bit bitwise-error observation vector $F_z = \{f_{z,j}\}_{j \in [\![0,m-1]\!]}$ of an EEV $E_z = \{e_{z,j}\}_{j \in [\![0,m-1]\!]}$ is extracted from $\hat{z}$ and z by an m-bit XOR.

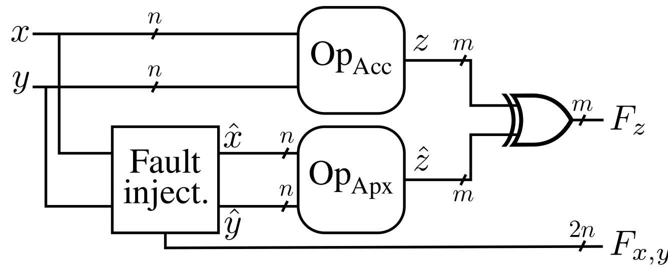


Figure 3.3 – Extraction of Binary Error Event Vectors for BWER Model Training

The conditional probability of Equation 3.4 can be estimated by the ratio between the number of observations of the event $e_{z,j}$ when the corresponding EEV $E_{x,y,i}$ is simultaneously observed, referred as $(f_{z,j} = 1|F_{x,y,i})$, and $F_{x,y,i}$ the total number of observations of $E_{x,y,i}$:

$$P\left(e_{z,j}|E_{x,y,i}\right) \equiv \frac{\sum_l \left(f_{z,j} = 1|F_{x,y,i}\right)_l}{\sum_l (F_{x,y,i})_l}. \tag{3.5}$$

Figure 3.4 presents an example of how are trained the corresponding conditional probabilities after one iteration of training. The inputs and output are both 4-bit and $k = 2$. First, looking at the blue part, after approximate operation, the output LSB is not faulty. Therefore, $f_{z,0} = 0$. As $\hat{x}_0$ was not faulty and $\hat{y}_0$ was faulty, the corresponding observation of input EEV is $F_{x,y,2}$. Thus, following Equation 3.5, the estimation of $P\left(e_{z,0}|E_{x,y,2}\right)$ is modified by increasing the denominator by 1. Then, looking at the yellow part of Figure 3.4, the output observation is $f_{z,1} = 1$, meaning the output is erroneous. As $k = 2$, input ranks 1 and 0 are observed together. The corresponding observation $(0, 1, 1, 0)$ is $F_{x,y,6}$. $P\left(e_{z,1}|E_{x,y,6}\right)$ is modified increasing the numerator by 1 (faulty output) and increasing the denominator by 1. The operation is repeated at significance position 2 (green part) and 3 (red part), each time observing $2k = 2$ input error vector bits.

Following this method, the conditional probability data structure has m elements updated at each new training cycle. Finally, after a sufficient number of training cycles (discussed in Section 3.3.1), the model is trained and ready for propagation, which is discussed in the following section.

3.2.3 BWER Propagation Algorithm

The propagation of BWER is performed the following way. Given two BWER inputs $B_x = \{b_{x,i}\}_{i \in [\![0,n-1]\!]}$ and $B_y = \{b_{y,i}\}_{i \in [\![0,n-1]\!]}$ and the equivalent conglomerate vector

$$B_{x,y} = \{b_{y,n-1}, b_{x,n-1}, b_{y,n-2}, b_{x,n-2}, \ldots, b_{y,1}, b_{x,1}, b_{y,0}, b_{x,0}\}\,. \tag{3.6}$$

Figure 3.4 – Example of Conditional Probabilities Training for $n = 4$, $m = 4$ and $k = 2$

The output BWER $B_z = \{b_{z,j}\}_{j \in [\![0,m-1]\!]}$ is then determined by

$$b_{z,j} = \sum_{i=0}^{4^{\min(j,k)}-1} \beta_{i,j} P\left(e_{z,j} | E_{x,y,i}\right), \tag{3.7}$$

where $\beta_{i,j}$ is the probability of the event vector $E_{x,y,i}$ to be true at significance j knowing $B_{x,y}$, referred as $P_j\left(E_{x,y,0} | \, B_{x,y}\right)$. Let the partial input BWER $b_{x,3}$, $b_{y,3}$, $b_{x,4}$, $b_{y,4}$ given by Table 3.3. Let an approximate adder trained with $k = 2$. To determine $b_{z,4}$ (similarly for other

$b_{y,4}$	$b_{x,4}$	$b_{y,3}$	$b_{x,3}$
0.21	0.14	0.17	0.05

Table 3.3 – Partial Input BWER for the Example

$b_{z,i}$), $\beta_{i,4}$ must be known for $i \in [\![0, 15]\!]$. $\beta_{6,4}$ is calculated from Table 3.3 the following way:

$$\begin{aligned} \beta_{6,4} &= P_4\left(E_{x,y,6} | \, B_{x,y}\right) \\ &= P\left(e_{y,4}, \overline{e}_{x,4}, \overline{e}_{y,3}, e_{x,3}\right) \\ &= 0.21 \times (1 - 0.14) \times (1 - 0.17) \times 0.05 \\ &= 7.49\text{E}{-3} \end{aligned}$$

This operation has then to be iterated and summed for all other 15 values i to determine $b_{z,4}$, and again for all j in $[\![0, m-1]\!]$.

3.3 Results of the BWER Method on Approximate Adders and Multipliers

This section presents results about the BWER propagation method. These results were produced a few days before the redaction of this part of the document and are consequently not yet published. First, the convergence speed of the trained data structure is studied in Section 3.3.1. Then, results concerning stand-alone approximate adders and multipliers and tree structures of adders are given considering inputs with a maximal activity.

3.3.1 BWER Training Convergence Speed

As developed previously in Section 3.2.2, one of the main interests of BWER propagation method is the reasonable memory cost of the trained structure. However, the method can only be suitable if the training time is not too important. In this Section, we evaluate the convergence speed of BWER training.

To evaluate the convergence speed, three approximate adders and two approximate multipliers were used. The adders are ACA, ETAIV and IMPACT, described in Section 1.4.1. The multipliers are AAMIII, denoted as AAM in the rest of this section, and DRUM described in Section 1.4.2. They were all tested with input/output widths between 8 and 16 using multiple configurations, with values of k in BWER method in $\{2, 4, 6, 8\}$, leading to 374 different experiment parameters. In this experiment, the reference of the final BWER trained structure are the values obtained after 10^8 random value draws. The experiment was performed using 374 cores of IRISA's computing grid IGRIDA, each instance of the experiment being computed on a single-core for correct time analysis. IGRIDA computing grid embeds 1700 cores leveraging Intel Xeon CPUs. All implementations are done in C++, using approximate library APX__FIXED described in the next chapter.

The results of training convergence speed are depicted in Figure 3.5. The experiments show that the training time relative to the number of random training samples is independent of k on a same processor. Therefore, a single curve for elapsed time is depicted on the figure in dotted red, which is the mean of the curves for all 374 experiments. The elapsed time is linear with the number of training samples. On this experiment, the 10^8 training samples took about 35 minutes for each experiment in average. The blue curves represent the mean distance of the training values from the reference for each k. As a reminder, the values in the data structure are probabilities and thus they are in $[0, 1]$.

A first observation of the curves shows that the convergence speed of the training clearly depends on k. The smaller k, the faster the convergence. For $k = 2$, the mean distance of the estimation from the reference gets under 10^{-2} between $20,000$ and $50,000$ training samples, which represents only $0.5 - 1.2$ seconds of training. For $k = 8$, getting as near from the reference as 10^{-1} takes about $1,000,000$ training samples, which represents a training time of 22 seconds. Indeed, when k is small, the training structure is very small, and the elements of the structure are likely to be activated by a random input with a high probability. When k is large, however, each element of the structure has very few chances to be activated by the drawn number. This is why the number of random inputs necessary to have a sufficient number of activations for all elements of the data structure to estimate the BWER probabilities grows very fast with the value of k. From now on, all BWER estimation structures used are trained on 10^8 input samples.

3.3.2 Evaluation of the Accuracy of BWER Propagation Method

As mentioned in Section 3.2.3, the trained structure of BWER propagation method is built to work for inputs with maximum activity at each bit position. In this section, all presented results

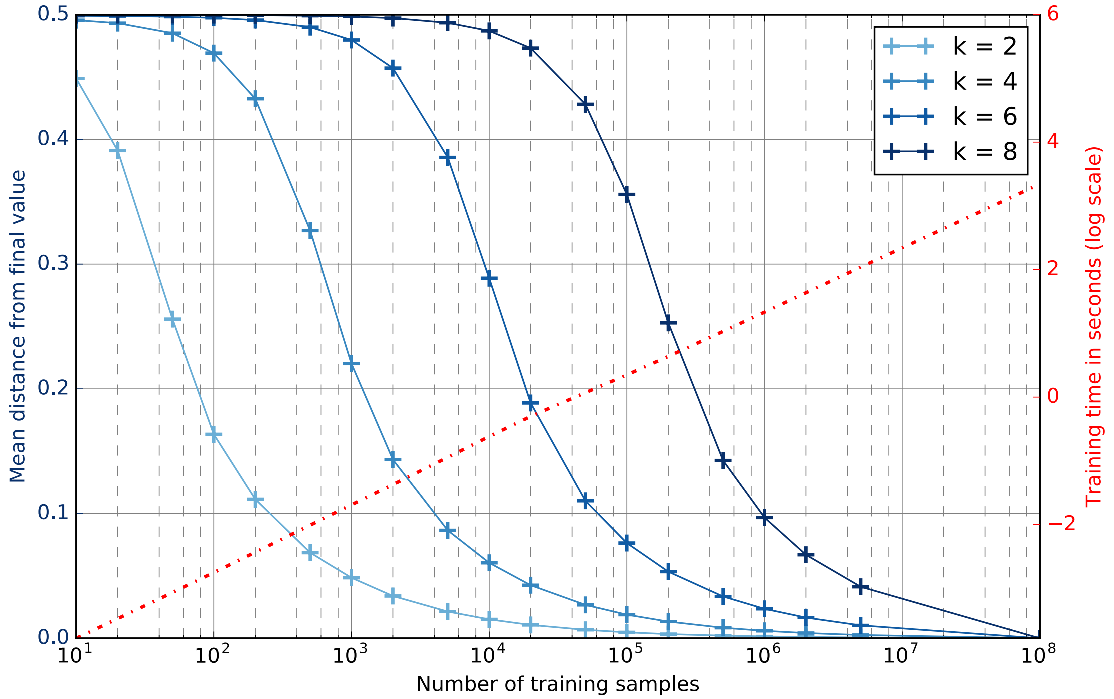


Figure 3.5 – Convergence of BWER Training in Function of k

are produced in these conditions, meaning α_j in Equation 3.9 is always worth 0.5. Two experiments are presented in this section. First, results on single stand-alone operators are given, using the same approximate operators as the ones described in the previous section. Finally, results on tree of operations for the same approximate operator instance are given. Figure 3.6 shows the example of this structure, made of three stages. This tree structure is considered to represent typical data-flow graphs used in signal processing applications and is used to see how the model propagates in this structure. Even if the model can be accurate for one operator alone, the interest of analytical models is in their use for application-level error estimation, which is not considered in the models currently published in the literature for approximate operators.

To evaluate the efficiency of the BWER method, the mean distance

$$D_B = \frac{1}{n}\sum_{i=0}^{n-1}\left|\hat{B}(i) - B(i)\right|, \tag{3.8}$$

is used, where $B(i), i \in [\![0, n-1]\!]$ is the reference output BWER of a given operator of output width n and $\hat{B}(i), i \in [\![0, n-1]\!]$ is the output estimated by the model. The first experiment without considering the activity of the inputs is the verification of the model on stand-alone operators. For this experiment and all the other experiments to come in this section, all simulations for the reference are run on 10^7 input samples. For the estimation time, the analytical propagation is averaged on at least 5 seconds of repeated experiments.

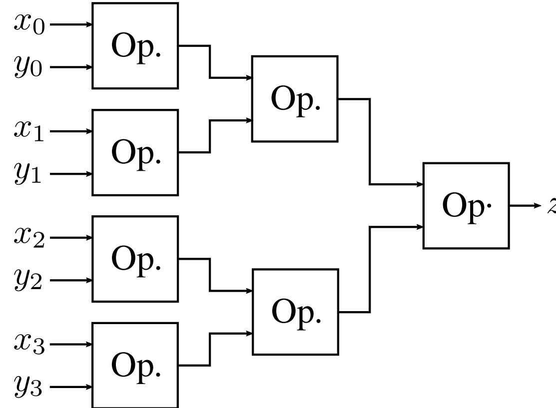


Figure 3.6 – Tree Operation Structure with Three Stages

An example of estimation and simulation output BWER vector is shown in Figure 3.7 for a 12-bit approximate adder ACA with carry chains limited to $x = 2$, and a parameter $k = 4$ for the BWER propagation method. In this specific case, the estimation is very near from the reference simulation. There are no errors occurring on the first 3 bits, which is normal for an ACA with $x = 2$. Then, the error probability increases with the bit significance, as expected. Other cases may show different results, depending on the value of k. E.g, if $k \geq x$ in ACA, then at least one LSB causing error at a higher significance is not taken into account. In this case, the estimation is not accurate. Therefore, it is important to find the optimal value of k to minimize the memory needed and the training time while keeping the best possible accuracy in the estimation.

Figure 3.8a shows the evolution of D_B with k for 16-bit ACA as a function of x. As expected, when $k < x$, the estimation is bad. Then, the quality of estimation improves when k approaches x. However, when k gets larger than x, the estimation gets worse again. This is due to the number of training samples, which is the same for all k in our experiment. As showed in Section 3.3.1, the training process converges slower when k is larger. Therefore, when $k > x$, as there is no improvement in the accuracy of the model compared to $k = x$, the slowest convergence is source of inaccuracy. Thus, the optimal value of k in this case is $k = x$, which is the best balance between accuracy and required training samples.

Figure 3.8b shows the evolution of D_B with k for 16-bit IMPACT as a function of the number of MSBs computed with an exact adder $N_{\mathtt{acc}}$. For IMPACT, the results are different than for ACA. Indeed, in IMPACT, every output bit depends on every input bit of lower significance. Therefore, for all $k < n$, there is a lack of information in the model. It is interesting to see that for IMPACT, when the number of accurate MSB computations gets larger, the accuracy of the estimation gets worse off, until a certain value of k for which the lack of samples used for training the model compensates for the increase of the information taken into account. This tendency is quite opposed to what could be expected and needs to be investigated in the future for better comprehension.

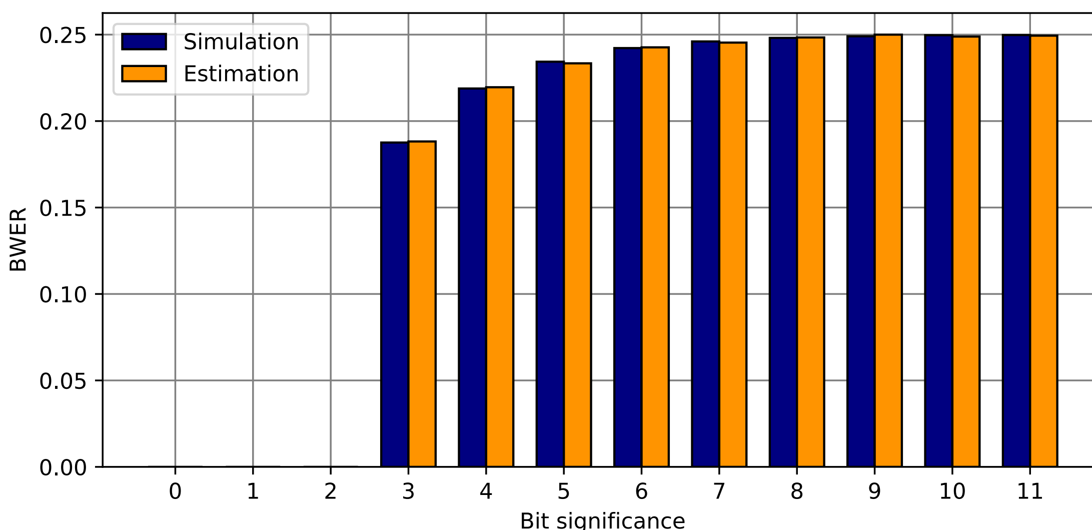
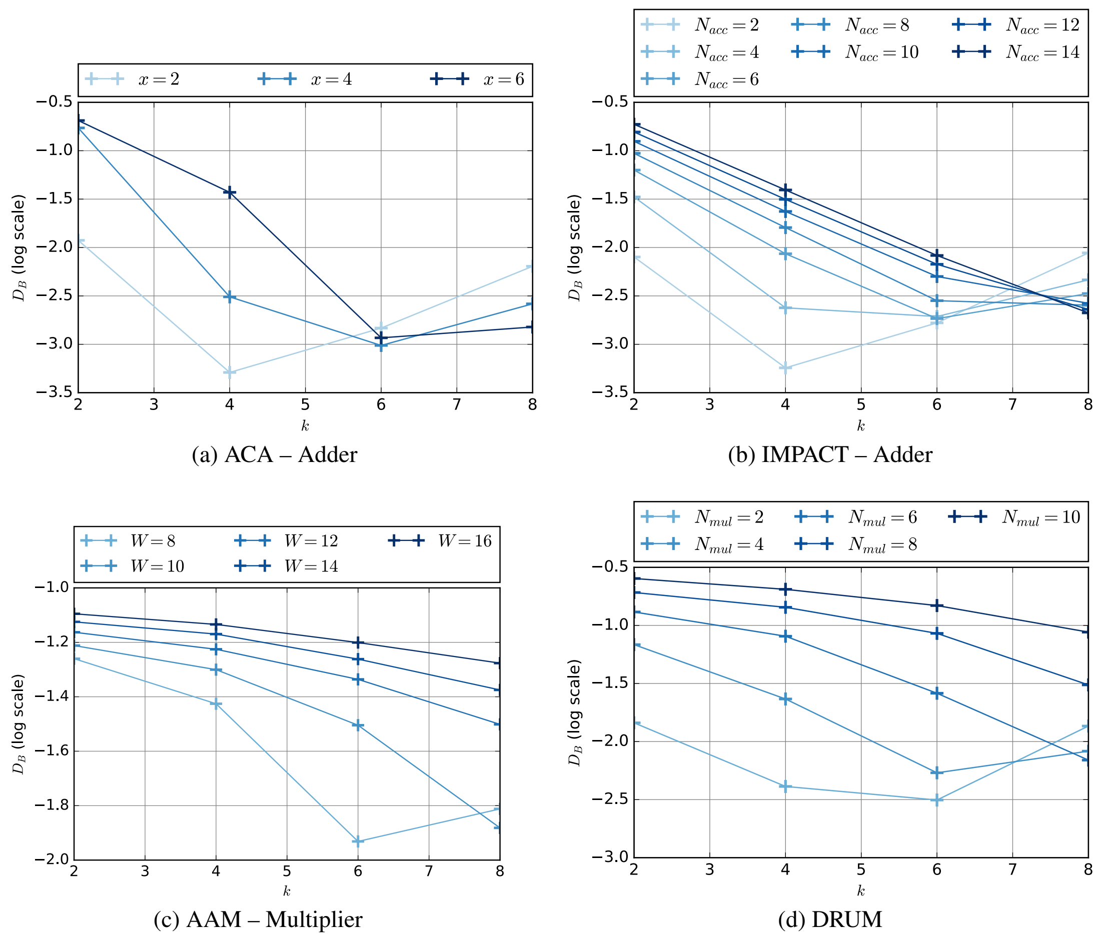


Figure 3.7 – BWER Estimation and Simulation Results for Stand-Alone ACA with $x = 2$ and $k = 4$

AAM and DRUM multipliers results are respectively given in Figures 3.8c and 3.8d. A first observation on the error shows that globally, the accuracy of the estimation is worse off than for adders. Indeed, there is a main difference between adders and multipliers. In adders, the output bit of significance i is strongly dependent on the input bits of significance i, and then less and less dependent on the input bits of significance $i-1$, $i-2$, $i-3$, $\ldots$, 0. Therefore, when k is high enough, the absence of information on the inputs of rank $i-k-1$, $\ldots$, 0 represents a small accuracy penalty. However, for an n-bit multiplier with n even, the output bit of significance $n-1$ is as dependent on the inputs of significance $n-1$ as on the input bits of significance 0. However, when $k < n$, the input bit of significance 0 is not reached by the estimation, leading to potentially bad accuracy. In Figure 3.8c for AAM results, as AAM has no parameter, each curve represents the bit-width of the multiplier. The smaller the multiplier, the better the estimation. For all AAM widths, increasing k always leads to better accuracy as expected, except for 8-bit AAM with $k = 8$, which is worse than $k = 6$. Indeed, as only the most significant half of multiplication is taken into account in AAM, this means that for each bit j at the output no partial product implying input significance 0 is operating at weight j. Therefore, when switching from $k = 6$ to $k = 8$, only one more significant bit instead of two is taken into account, which is not enough to counterbalance the fact that the training was less efficient for $k = 8$, thus leading to a lower accuracy. The same phenomenon can be observed for DRUM on Figure 3.8d when the floating multiplier is only 2-bit or 4-bit large.

Figure 3.9 presents the value of D_B metric of adders for different number of stages in the tree configurations described in Figure 3.6. For each figure, the number of stages varies between two and four, and the output error refers to the output of the last stage. For each configuration of approximate operators, the value of k giving the best results in the previous stand-alone estima-



(a) ACA – Adder

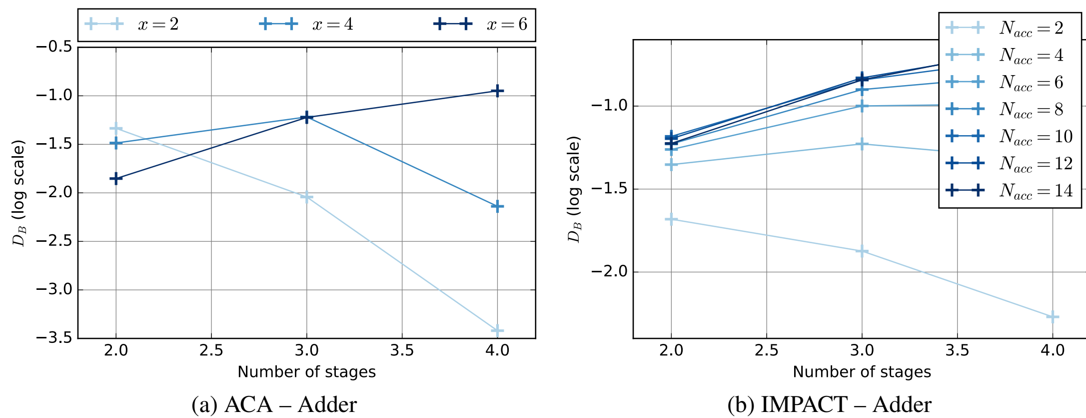
(b) IMPACT – Adder

(c) AAM – Multiplier

(d) DRUM

Figure 3.8 – Evolution of estimation error D_B with k for different configurations of 16-bit approximate stand-alone adders and multipliers

tion is used. For ACA on Figure 3.9a, for $x = 6$, the estimation becomes less accurate with the number of stages, which is what could be expected. However, for the other configurations, the opposite is observed, i.e., more stages lead to better estimation accuracy. This is actually due to the high BWER of the approximate adder configuration, which leads to a maximal BWER at nearly all positions after a high number of stages. As the BWER propagation model also estimates a maximal BWER, not because of a high accuracy but because of a *saturation* effect, the estimation becomes very good. However, this is only a side effect of the bad performance induced by using several layers of ACA.



(a) ACA – Adder

(b) IMPACT – Adder

Figure 3.9 – Evolution of estimation error D_B with k for different configurations of 16-bit approximate adders with different number of stages

3.3.3 Estimation and Simulation Time

As the BWER propagation method is analytical, it is made for fast accuracy evaluation. In this section, the execution time of the method is evaluated. Table 3.4 gives the time spent for the BWER propagation method to evaluate ACA, IMPACT, AAM and DRUM in their 8-, 12- and 16-bit versions. For each bitwidth, the training is obtained taking the average of several different parameters of all approximate adders and multipliers, except for AAM which takes no parameter. All simulations are run on 10^7 points. All estimations are repeated during at least 5 seconds and the total time is divided by the number of repetitions. All computations were run on IGRIDA computing grid composed of different models of Intel Xeon processors. As the computing grid is heterogeneous, the evaluation may vary if one processor or another is used, and this must be considered in the analysis of results of Table 3.4.

BWER propagation time roughly oscillates between $500\mu s$ and 18ms for addition, and between $800\mu s$ and 42ms for multiplication for stand-alone operators. In a larger system, the propagation time has to be multiplied by the number of operators. In comparison, 10^7 simulations take between 26s and 129s. This makes a huge difference when the operation has to be performed on complete systems and repeated many times, which is the case in an incremental system optimization process where many configurations of approximate operators must be

		Estimation time	Simulation time
ACA	8	0.447 ms	80.3 s
	12	10.8 ms	80.2 s
	16	15.7 ms	80.7 s
IMPACT	8	0.564 ms	129 s
	12	12.6 ms	97.7 s
	16	17.5 ms	85.2 s
AAM	8	18.8 ms	25.9 s
	12	25.1 ms	35.8 s
	16	42.7 ms	81.8 s
DRUM	8	0.766 ms	73.1 s
	12	25.3 ms	70.6 s
	16	42.4 ms	121 s

Table 3.4 – BWER Propagation and Simulation Time of Stand-Alone Approximate Operators – Simulation is run on 10^7 input samples

considered.

3.3.4 Conclusion and Perspectives

In the previous Sections, BWER propagation model principle, its convergence speed and some results on operators or tree of operators were presented. The existing literature and the results show how difficult it is to build strong general error propagation models for approximate operators. First, their many different natures make the accuracy of the models very dependent on each of the structures, and a good estimation accuracy on ACA for instance could give bad results on an adder like IMPACT (and vice versa). Then, their errors often containing scarce and very high amplitude peaks, it is very hard to evaluate this with analytical models smoothing these phenomena.

Another limitation of the BWER propagation method is that it is only suitable if all inputs are uniformly distributed on their whole dynamic, which means they have a maximal activity. Indeed, BWER does not carry any information about the activity at any position. If this hypothesis is not true, then the results must be weighted by information about activity. Let α_j the probability for the output bit z_j of an operator to be worth 1. Then, the output BWER in these conditions can be approximated by

$$b_{z,j} = 2\alpha_j \sum_{i=0}^{4^{\min(j,k)}-1} \beta_{i,j} P\left(e_{z,j} | E_{x,y,i}\right). \tag{3.9}$$

Indeed, if the input MSB are mostly worth 0 instead of equally 0 or 1, the output BWER on the MSBs will be proportionally lower. Thus, knowing the probability of input bits to be 1 at each position, these probabilities need to be propagated analytically across the operators along

with BWER propagation. In practice, the simplest way is to use the analytical propagation of these probabilities across exact operators (adders and multipliers). The propagation of the probability of the output bits to be worth 1 across an adder can be trivially calculated composing their propagation across a full adder. Their propagation across a multiplier is calculated on the composition of additions in the partial product reduction. The output probability of a bit to be worth 1 analytically obtained this way is then weighted by the initially computed BWER at the output of the operator. Indeed, as approximate operators are erroneous by nature, they can generate bit flips that can totally modify the value of activity when compared to an exact operator. If $\alpha_j^{'}$ is the probability for the output of an exact operator to have its j-th bit worth 1, then we approximate α_j (see Equation 3.9) as

$$\alpha_j = \alpha_j^{'} \times (1 - b_{z,j}) + b_{z,j} \times \left(1 - \alpha_j^{'}\right). \tag{3.10}$$

The question of using BWER for propagation is also contestable. Indeed, few significant signal processing metrics are possibly deducible from it. However, the objective of this section is to point out the interest of basing the analytical error estimation on models trained using simulated values of approximate adders, as it is the only way to catch their many different natures. Also, the interest of finding ways to reduce the storage cost of the models while sacrificing a minimum of estimation accuracy has been highlighted by the use of k in the training and the propagation method. In the future, methods derived from BWER training and propagation could be developed leveraging other metrics for the propagation and more efficient storage compression methods. Indeed, as discussed in Section 3.3.2, important data are likely to be lost, especially in multipliers, when choosing bad parameters for the model.

After many unsuccessful attempts and a deep study of the literature, a more general conclusion about modeling approximate operators error is that there seem not to be better method than Monte Carlo simulation. Indeed, simple models always suffer from high imprecision which does not allow them to be used in real system design process, while complex models giving reasonably good results always come with a high storage or computational cost approaching the cost of Monte Carlo simulation. Moreover, simulating approximate operators can be done with a potentially good computational efficiency. Indeed, most of them are essentially a composition of small exact adders or multipliers. When computing using a CPU, it is therefore possible to use the integer arithmetic units to accelerate the computations using high-level code instead of heavier gate-level descriptions. There also are good opportunities to use HLS on the approximate operators code to simulate them on FPGA, accelerating computation using DSPs. In the next section, pseudo-simulation leveraging approximate operators is used for the reproduction of VOS effects.

3.4 Modeling the Effects of Voltage Over-Scaling (VOS) in Arithmetic Operators

In Section 1.1.1, functional approximation leveraging VOS is discussed. In this section, a method to reproduce the effects of VOS on arithmetic operators using models based on approximate operators is discussed. As for BWER method developed above, it is based on model training applied to BER at each significance position of an operator.

Voltage scaling has been used as a prominent technique to improve energy efficiency in digital systems, scaling down supply voltage effects in quadratic reduction in energy consumption of the system. Reducing supply voltage induces timing errors in the system that are corrected through additional error detection and correction circuits. A class of circuit-level approximation is achieved by applying dynamic voltage and frequency scaling aggressively to an accurate operator. Due to the dynamic control of voltage and frequency, timing errors due to scaling can be controlled flexibly in terms of trade-off between accuracy and energy. This method is referred as Voltage Over-Scaling (VOS). It has the potential to unlock the opportunities of higher energy efficiency by operating the transistors near or below the threshold. VOS-based approximate operators can be used when error-resilient applications are considered.

Despite its high efficiency in terms of energy savings, sub-threshold VOS has important drawbacks. First, the process variability makes its effects hard to predict in a general way since two instances of a same chip are likely to behave differently. Second, besides on-chip measurements or transistor-level simulation simulation, there is no suitable method able to give a prediction of these effects. On-chip measurements is costly in terms of human resource, and observing at wire level the effects of VOS is impossible – adding hardware at that level would modify the nature of what is intended to observe. Transistor-level simulation (such as SPICE), on the other hand, is accurate. However, the computational resources and simulation time necessary for large system observation are prohibitive.

In this section, we intend to reproduce the effects of VOS at arithmetic operator scale, leveraging approximate operators. This allows for much faster and low-resource simulation, while keeping satisfying accuracy compared to the reality. For this, we propose a new modeling technique that is scalable for large-size operators and compliant with different arithmetic configurations. The proposed model is accurate and allows for fast simulations at the algorithm level by imitating the faulty operator with statistical parameters. We also characterize the basic arithmetic operators using different operating triads (combination of supply voltage, body-biasing scheme and clock frequency) to generate models for approximate operators. Error-resilient applications can be mapped with the generated approximate operator models to achieve better trade-off between energy efficiency and error margin. In our experiments using 28nm FDSOI technology, we achieve maximum energy efficiency of 89% for basic operators like 8-bit and 16-bit adders at the cost of 20% Bit Error Rate (ratio of faulty bits over total bits) by operating them in near-threshold regime.

3.4.1 Characterization of Arithmetic Operators

In this section, characterization of arithmetic operators is discussed for voltage over-scaling based approximation. Characterization of arithmetic operators helps to understand the behaviour of the operators with respect varying operating triads. Adders and Multipliers are the most common arithmetic operators used in datapaths. In this work different adder configurations are explored in the context of near-threshold regime.

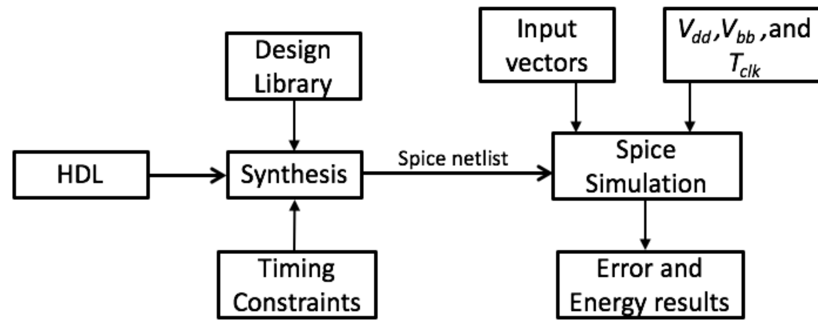


Figure 3.10 – Proposed Design Flow for Arithmetic Operator Characterization

Fig. 3.10 shows the characterization flow of the arithmetic operators. Structured gate-level HDL is synthesized with user-defined constraints. The output netlist is then simulated at transistor level using SPICE (Simulation Program with Integrated Circuit Emphasis) platform by varying operating triads (V_{dd}, V_{bb}, T_{clk}), where V_{dd} is supply voltage, V_{bb} is body-biasing voltage, and T_{clk} is clock period. In ideal condition, the arithmetic operator functions without any errors. Also, EDA tools introduce additional timing margin in the datapaths during Static Timing Analysis (STA) due to clock path pessimism. This additional timing prevents timing errors due to variability effects. Due to the limitation in availability of design libraries for near/sub-threshold computing, it is necessary to use SPICE simulation to understand the behaviour of arithmetic operators in different voltage regimes. By tweaking the operating triads, timing errors e are invoked in the operator and can be represented as

$$e = f(V_{dd}, V_{bb}, T_{clk}) \tag{3.11}$$

Characterization of arithmetic operator helps to understand the point of generation and propagation of timing errors in arithmetic operators. Among the three parameters in the triad, scaling V_{dd} causes timing errors due to the dependence of operator's propagation delay t_p on V_{dd}, such as

$$t_p = \frac{V_{dd}.C_{load}}{k(V_{dd} - V_t)^2} \tag{3.12}$$

Body-biasing potential V_{bb} is used to vary the threshold voltage (V_t), thereby increasing the performance (decreasing t_p) or reducing leakage of the circuit. Due to the dependence of t_p on V_t, V_{bb} is used solely or in tandem with V_{dd} to control the timing errors. Scaling down V_{dd} improves the energy efficiency of the operator due to its quadratic dependence to total energy.

$E_{total} = V_{dd}^2.C_{load}$. Mere increase in T_{clk} does not reduce the energy consumption, though it will reduce the total power consumption of the circuit

$$P_{total} = \alpha.V_{dd}^2.\frac{1}{T_{clk}}.C_{load} \tag{3.13}$$

Therefore, T_{clk} is scaled along with V_{dd} and V_{bb} to achieve high energy efficiency.

Characterization of Adders Adder is an integral part of any digital system. In this section, two adder configurations Ripple carry adder (RCA) and Brent-Kung adder (BKA) are characterized based on circuit level approximations. Ripple carry adder is a sequence of full adders with serial prefix based addition. RCA takes n stages to compute n-bit addition. In worst case, carry propagates through all the full adders and makes it longest carry chain adder configuration. Longest carry chain corresponds to the critical path of the adder, based on which the frequency of operation is determined. In contrast, Brent-Kung adder is a parallel prefix adder. BKA takes $2\log_2(n-1)$ stages to compute n-bit addition. In BKA, carry generation and propagation are segmented into smaller paths and executed in parallel.

Behaviour of arithmetic operator in near/sub-threshold region is different from the super-threshold region. In case of an RCA, when the supply voltage is scaled down, the expected behaviour is failure of critical path(s) from longest to the shortest with respect to the reduction in the supply voltage. Fig. 3.11 shows the effect of voltage over-scaling in 8-bit RCA. When the supply voltage is reduced from 1V to 0.8V, MSBs starts to fail. As the voltage is further reduced to 0.7V and 0.6V more BER is recorded in middle order bits rather than most significant bits. For 0.5V V_{dd}, all the middle order bits reaches BER of 50% and above. Similar behaviour is observed in 8-bit BKA shown in Fig. 3.12 for v_{dd} values of 0.6V and 0.5V. This behaviour imposes limitations in modelling approximate arithmetic operators in near/sub-threshold using standard models. Behaviour of arithmetic operators during voltage over-scaling in near/sub-threshold region can be characterized by SPICE simulations. But SPICE simulators take long time (4 days with 8 cores CPU) to simulate exhaustive set of input patterns needed to characterize arithmetic operators.

3.4.2 Modelling of VOS Arithmetic Operators

As stated previously, there is a need to develop models that can simulate the behavior of faulty arithmetic operators at functional level. In this section, we propose a new modelling technique that is scalable for large-size operators and compliant with different arithmetic configurations. The proposed model is accurate and allows for fast simulations at the algorithm level by imitating the faulty operator with statistical parameters.

As VOS provokes failures on the longest combinatory datapaths in priority, there is clearly a link between the impact of the carry propagation path on a given addition and the error issued from this addition. Figure 3.13 illustrates the needed relationship between hardware operator controlled by operating triads and statistical model controlled by statistical parameters P_i. As the knowledge of the inputs gives necessary information about the longest carry propagation chain, the values of the inputs are used to generate the statistical parameters that control the equivalent model. These statistical parameters are obtained through an off-line optimization

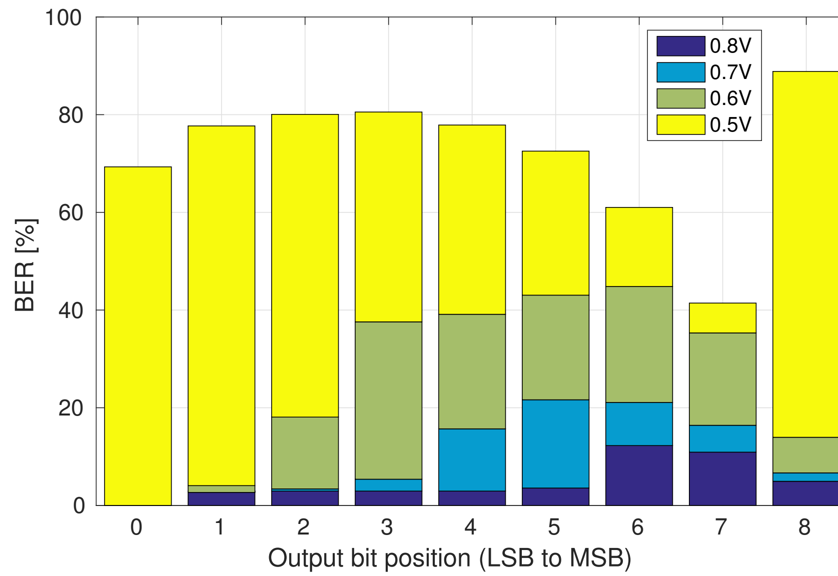


Figure 3.11 – Distribution of BER in output bits of 8-bit RCA under voltage scaling

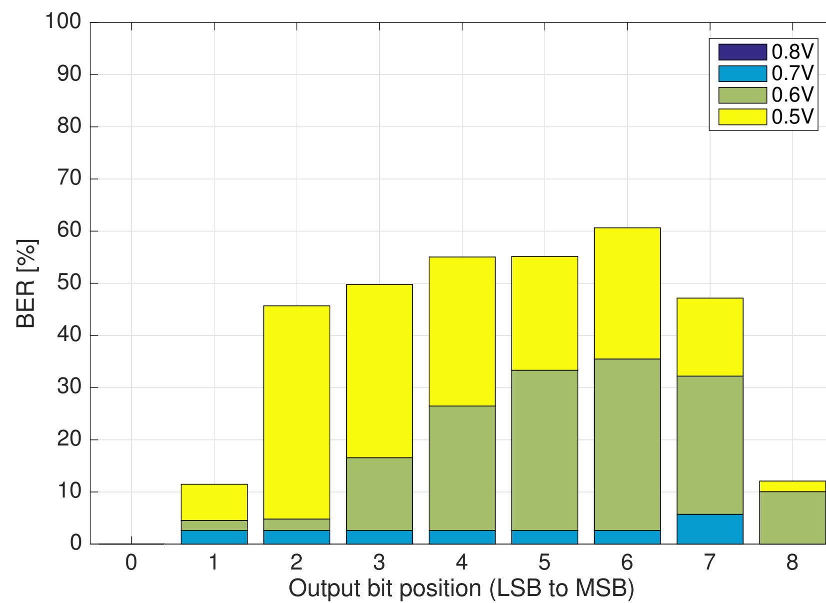


Figure 3.12 – Distribution of BER in output bits of 8-bit BKA under voltage scaling

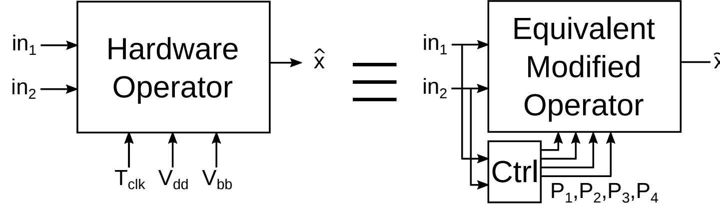


Figure 3.13 – Equivalence Between Faulty Hardware Operator and Equivalent Functionally Faulty Operator

process that minimizes the difference between the outputs of the operator and its equivalent statistical model, according to a certain metric. In this work, we used three accuracy metrics to calibrate the efficiency of the proposed statistical model:

- Mean Square Error (MSE) – average of squares of deviations between the output of the statistical model $\tilde{x}$ and the reference $\hat{x}$:

$$MSE(i) = \frac{1}{n}\sum_{i=1}^{n}(\hat{x}_i - \tilde{x}_i)^2 . \tag{3.14}$$

- Hamming distance – number of positions with bit flip between the output of the statistical model $\tilde{x}$ and the reference $\hat{x}$:

$$d_{ham}(i) = \sum_{j=0}^{N-1}(\hat{x}_{i,j} \oplus \tilde{x}_{i,j}) . \tag{3.15}$$

- Weighted Hamming distance – Hamming distance with weight for every bit position depending on their significance:

$$d_{w_ham}(i) = \sum_{j=0}^{N-1}(\hat{x}_{i,j} \oplus \tilde{x}_{i,j}) .2^j . \tag{3.16}$$

Proof of Concept: Modelling of Adders In the rest of the section, we develop a proof of concept by applying VOS on different adder configurations. All the adder configurations are subjected to VOS and characterized using the flow described in Fig. 3.10. Fig. 3.14 shows the design flow of modelling VOS operators. As shown in Fig. 3.13 rudimentary model of the hardware operators is created with the input vectors and the statistical parameters. For the given input vectors, output of both the model and the hardware operator is compared based on the defined set of accuracy metrics. The comparator shown in Fig. 3.14 generates signal to noise ratio (SNR) and Hamming distance to determine the quality of the model based on the accuracy metrics. SNR and Hamming distance are fed back to the optimization algorithm to further fine tune the model to represent the VOS operator.

In the case of adder, only one parameter P_i for the statistical model is used and is defined as C_{max}, the length of the maximum carry chain to be propagated. Hence, given the operating

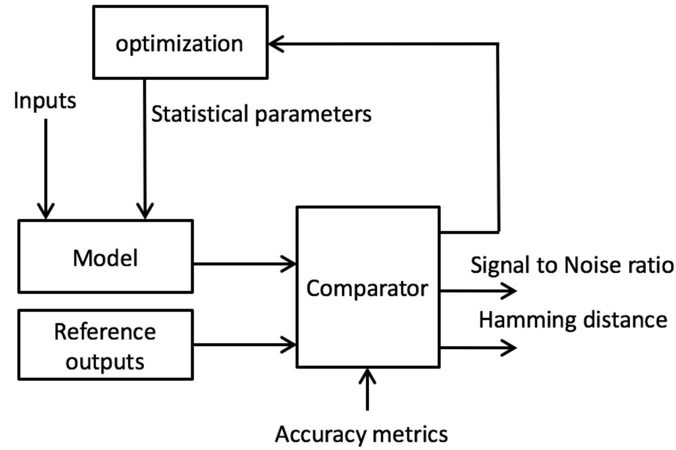


Figure 3.14 – Design flow of modelling of VOS operators

parameters $(T_{clk}, V_{dd}, V_{bb})$, where V_{dd} is supply voltage, V_{bb} is body-biasing voltage, and T_{clk} is the clock period, and a couple of inputs (in_1, in_2), the goal is to find C_{max}, minimizing the distance between the output of the hardware operator and the equivalent modified adder. This distance can be defined by the above listed accuracy metrics. Hence, C_{max} is given by:

$$C_{max}\left(in_1, in_2\right) = \underset{C \in [0,N]}{Argmin} \left\| \hat{x}\left(in_1, in_2\right), \tilde{x}\left(in_1, in_2\right) \right\|$$

where $\|x, y\|$ is the chosen distance metric applied to x and y. As the search space for characterizing C_{max} for all sets of inputs is potentially very high, C_{max} is characterized only in terms of probability of appearing as a function of the theoretical maximal carry chain of the inputs, denoted as $P\left(C_{max} = k | C_{max}^{th} = l\right)$. This way, the mapping space of 2^{2N} possibilities is reduced to $(N+1)^2/2$. Table 3.5 gives the template of the probability values needed by the equivalent modified adder to produce an output.

Table 3.5 – Carry propagation probability table of modified 4-bit adder

C_{max} \ C_{max}^{th}	0	1	2	3	4
0	1	$P(0\|1)$	$P(0\|2)$	$P(0\|3)$	$P(0\|4)$
1	0	$P(1\|1)$	$P(1\|2)$	$P(1\|3)$	$P(1\|4)$
2	0	0	$P(2\|2)$	$P(2\|3)$	$P(2\|4)$
3	0	0	0	$P(3\|3)$	$P(3\|4)$
4	0	0	0	0	$P(4\|4)$

The optimization algorithm used to construct the modified adder is shown in Algorithm 2. When the inputs (in_1, in_2) are in the vector of training inputs, output of the hardware adder configuration $\hat{x}$ is computed. Based on the particular input pair (in_1, in_2), maximum carry

chain C^{th}_{max} corresponding to the input pair is determined. Output $\tilde{x}$ of the modified adder with three input parameters (in_1, in_2, C) is computed. The distance between the hardware adder output $\hat{x}$ and modified adder output $\tilde{x}$ is calculated based on the above defined accuracy metrics for different iterations of C. The flow continues for the entire set of training inputs.

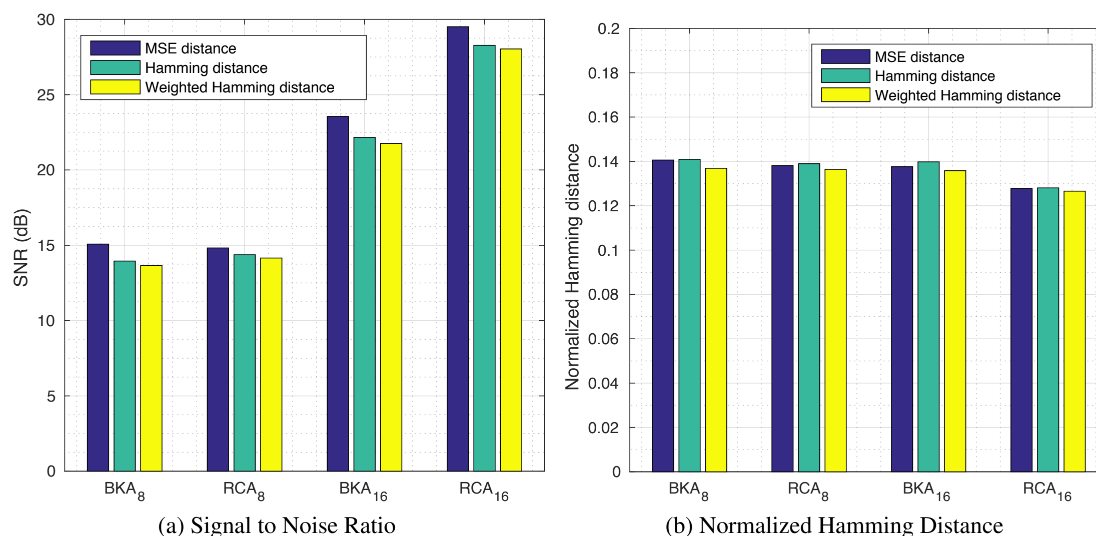
Algorithm 2 Optimization Algorithm

$P(0 : N_{bit_adder} | \, 0 : N_{bit_adder}) \leftarrow 0$
$max_dist \leftarrow +\infty$
$C_{max_temp} \leftarrow 0$
for $variable\ in_1, in_2 \in training_inputs$ **do**
  $\hat{x} \leftarrow add_hardware(in_1, in_2)$ $C^{th}_{max} \leftarrow max_carry_chain(in_1, in_2)$
  for $variable\ C \in C^{th}_{max}\ down\ to\ 0$ **do**
    $\tilde{x} \leftarrow add_modified(in_1, in_2, C)$
    $dist \leftarrow \|\hat{x}, \tilde{x}\|$
    if $dist <= max_dist$ **then**
      $dist_max \leftarrow dist$
      $C_{max_temp} \leftarrow C$
    end if
  end for
  $P(C_{max_temp} | C^{th}_{max}) + +$
end for
$P(: | :) \leftarrow P(: | :)/size(training_outputs)$

Once the offline optimization process performed, the equivalent modified adder can be used to generate the outputs corresponding to any couple of inputs in_1 and in_2. To imitate the exact operator subjected to VOS triads, the equivalent adder is used in the following way:

1. Extract the theoretical maximal carry chain C^{th}_{max} which would be produced by the exact addition of in_1 and in_2.
2. Pick of a random number, choose the corresponding row of the probability table, in the column representing C^{th}_{max}, and assign this value to C_{max}.
3. Compute the sum of in_1 and in_2 with a maximal carry chain limited to C_{max}.

For the experiments, the equivalent modified adder used is ACA, presented in Section 1.4.1.1. As a reminder, for an n-bit ACA parameterized by k, each output sum bit is calculated considering only the k previous input carries. This approximated operator is therefore chosen as its effects represent quite well the effects of VOS, with errors occurring on the critical path, i.e. the carry chain. Therefore, the control parameter used in the optimization of the model is the value of k. Figure 3.15 shows the estimation error of model of different adders based on the above defined accuracy metrics. SPICE simulations are carried out in 43 operating triads with 20K input patterns. Input patterns are chosen in such a way that all the input bits carry equal probability to propagate carry in the chain. Figure 3.15a plots the SNR of 8- and 16-bit RCA and BKA adders. MSE distance metric shows higher mean SNR, followed by Hamming distance and weighted Hamming distance metrics. Since MSE and weighted Hamming distance



(a) Signal to Noise Ratio (b) Normalized Hamming Distance

Figure 3.15 – Estimation Error of the Model for Different Adders and Distance Metrics

are taking the significance of bits into account, their resulting mean SNRs are higher than for the Hamming distance metric. Figure 3.15b shows the plot of normalized Hamming distance of all the four adders. In this plot, MSE and Hamming distance metrics are almost equal, with a slight advantage for non-weighted Hamming distance, which is expected since this metric gives all bit positions the same impact. Both the 8-bit adders have same behavior in terms of the distance between output of hardware adder and modified adder. On the other hand, 16-bit RCA is better in terms of SNR compared to its BKA counterpart. These results demonstrate the accuracy of the proposed approach to model the behavior of operators subjected to VOS in terms of approximation.

This method was presented in [3], along with error versus energy SPICE simulations under VOS. In these results, important energy savings are performed using VOS, with no or limited errors on the output. Figure 3.16 shows the results obtained for different voltage triads on a 16-bit RCA. The link between the intensity of error, represented here as BER and energy savings is clearly established, as well as the many possible tuning knobs. This emphasizes the need for models such as the one presented here, so tuning a faulty circuit does not take days to weeks.

However, the method presented in this section has the main drawback to depend on simulations results to be trained. As these simulations are extremely long, only 20,000 simulated points at best could be used for this model training, which is very low and does not ensure robustness of the generated outputs. Moreover, it is still hard to test the accuracy of this method on more complex systems, since transistor-level SPICE simulations has prohibitive computational time and memory cost.

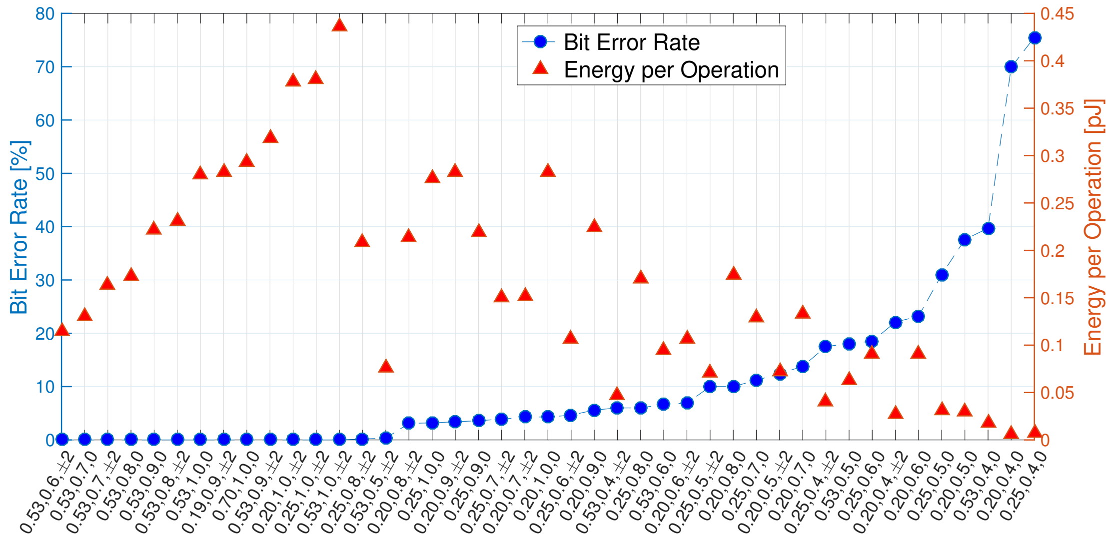


Figure 3.16 – BER and Energy for Different VOS Triads Applied to 16-bit RCA

3.4.3 Conclusion

As a conclusion for this chapter, approximate computing error modeling is a very complex task. Today, there is still no real suitable method for analytical propagation of errors at the application level as it exists for FxP arithmetic. Indeed, existing techniques always come with a major drawback – the accuracy of the estimation always has to be traded for memory or computational cost. There are two main reasons for this:

1. The output error strongly depends on the inputs. A variation of 1 bit in an input can easily switch between a perfectly accurate result to an error with the amplitude of the MSB.
2. And, as stated and developed in Chapter 1, the countless approximate operators of different natures make general rules hard to be found.

In the BWER propagation method proposed in Section 3.2, a solution implying reasonable memory cost with very fast error estimation after model training was proposed. However, as it is limited to BWER, only a limited number of metrics can be extracted, and the lack of accuracy in the estimation makes it not scalable to large systems.

What was finally observed along the many attempts during the development of this thesis and the reading of literature about integer approximate operators is that, when accuracy of error estimation is sought, Monte Carlo simulation is often the best solution. First, it is easier to extract any metric from the results, and especially metrics relative to an application. Moreover, the particularity of approximate operators is that, to be energy efficient, they have to remain quite simple and can often be represented by series of additions. Therefore, their functions can often be efficiently coded using integer functions supported by all CPUs so they execute very fast and therefore Monte Carlo simulations can remain efficient.

In this chapter, we also showed that approximate operators functions could be useful to represent complicated physical phenomena such as VOS faults, combining them to estimate these faults using Monte Carlo simulation.

In the next chapter, FxP and approximate operators paradigms are compared in terms of raw error and hardware performance, as well as in terms of error and performance regarding real-life signal processing applications. For this, only Monte Carlo simulation is used, so that the study is not dependent on the accuracy variations of models.

Chapter 4

Approximate Operators Versus Careful Data Sizing

In Chapter 1, fixed-point (FxP) and approximate (Apx) operators paradigms were presented. In Section 1.3, FxP arithmetic is presented as well as quantization error. Classical techniques for FxP refinement as well as PSD propagation method are presented in Chapter 2. Error management of Apx operators and their issues are discussed in Chapter 3. In this chapter, both arithmetic paradigms are compared.

On the one hand, FxP arithmetic is relying on the use of accurate integer operators, inaccuracy being induced by quantization process, mostly at arithmetic operators outputs. On the other hand, approximate operators rely on inaccurate designs inducing errors by nature. Therefore, this chapter compares data sizing to functional approximation.

For this, an open-source hardware performance (area, delay, power, energy) and accuracy characterization framework was developed during the thesis related to this document. This framework allows fast, accurate and user-friendly simulation-based area, delay and power estimation of approximate arithmetic operators in a general meaning – FxP, floating-point (FlP) or Apx operators such as presented in Chapter 1. It comes with built-in approximate adders and multipliers libraries, real-life applications benchmarks, and very complete scripts for results processing. The framework is presented in Section 4.1. The raw comparison between FxP and Apx paradigms is presented in Section 4.2. Finally, the performance of both are compared in signal processing applications in Section 4.3.

4.1 APXPERF: Hardware Performance and Accuracy Characterization Framework for Approximate Computing

This section presents APXPERF, which is the open-source hardware performance and accuracy characterization framework developed during the thesis. Two versions were developed. The first one, based on Bash and Matlab scripts and taking VHDL for hardware performance evaluation and C++ for error evaluation, is presented in Section 4.1.1. The second one, written

with Python3, adds an HLS layer so only a unique C++ source is needed for both functional and hardware simulation.

4.1.1 APXPERF– First Version

The first version of APXPERF, presented in [1], is a framework dedicated to approximate arithmetic operators characterization. It is composed of a hardware characterization part and an accuracy characterization part.

The hardware characterization part is based on VHDL specifications of the operators. Given an approximate operator VHDL code, the code is parsed to the clock and reset signals, as well as the two inputs and the output. The parameters of the approximate operator, which must be resolved at compile time, are set up using *generic* variables. The VHDL source is then compiled along with linked hardware libraries provided by the user using Synopsys *Design Compiler* and a gate-level design is produced. Area and delay are extracted from *PrimeTime* reports. Then, a VHDL benchmark is generated, automatically interfaced with the operator to be characterized. The benchmark is then run using *Modelsim* from Mentor Graphics. A *VCD* file containing all the transitions at every gate interface with a default 1ps granularity is generated. Finally, the *VCD* and *SDF* files and the technology libraries are passed to Synopsys PrimeTime, which produces a very accurate time-based estimation of dynamic and static power of the circuit.

In parallel, a C source of the same operator is given to a C++ framework. The function parameters (composed of the operator inputs and output and the parameters of the operator) are parsed and a test-bench is generated. Then, the simulation is run. An important number of metrics are returned by the simulation:

- Mean Square Error (MSE),
- Mean Absolute Error (MAE),
- Mean Absolute Relative Error,
- Error Rate,
- Bit Error Rate (BER),
- Bitwise Error Rate (BWER),
- Acceptance Probability (AP) given a Minimum Acceptable Accuracy (MAA),
- Minimum and Maximum Error,
- Mean of Error (or error bias),
- Power Spectral Density (PSD) of error, and
- Probability Density Function (PDF) of error.

Hardware and accuracy simulations are run taking uniformly distributed inputs on the whole range of the operator to be tested. Accuracy simulations are optimized for parallel execution with OpenMP for a high speed-up when used on a multicore machine. The overall scheme of the first version of APXPERF is depicted in Figure 4.1.

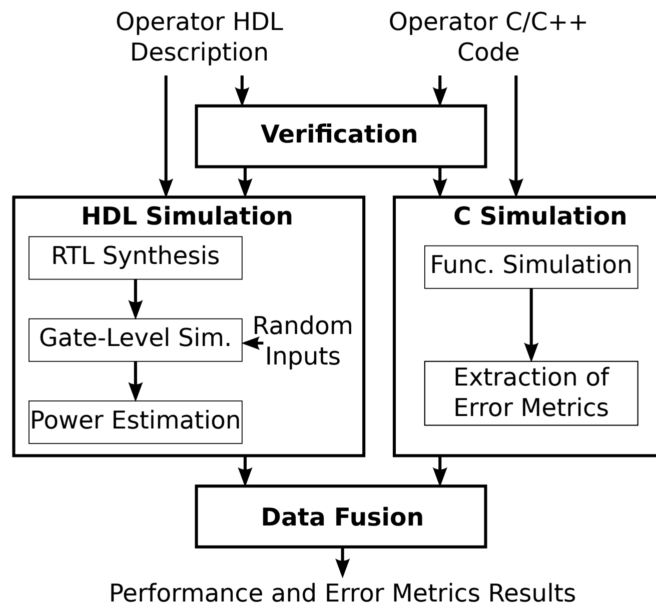


Figure 4.1 – First version of APXPERF

The framework comes with built-in C and VHDL versions of the following approximate operators:

- Almost-Correct Adder (ACA, see Section 1.4.1.1),
- Error-Tolerant Adder Version IV (ETAIV, see Section 1.4.1.2),
- IMPrecise Adder for low-power Approximate CompuTing (IMPACT, see Section 1.4.1.5),
- Approximate Array Multiplier III (AAMIII, see Section 1.4.2.1), and
- FxP adders and multipliers with various sizes.

APXPERF framework also embeds several signal processing applications, only for the accuracy evaluation part – Fast Fourier Transform (Fast Fourier Transform (FFT)), K-Means clustering algorithm, JPEG encoding and motion compensation filter for High Efficiency Video Codec (HEVC). These applications are further described in Section 4.3.

The first version of APXPERF was used for all the results in this Chapter. However, the second version, described in the next section, brings many improvements, mostly thanks to HLS and the usage of C++ templates for the parameterization of operators and simulations.

4.1.2 ApxPerf– Second Version

The second version of ApxPerf so far brings an extra-layer of high level synthesis. In this version, whose framework is described in Figure 4.2, only one source is needed for both hardware and accuracy estimation, written in C++.

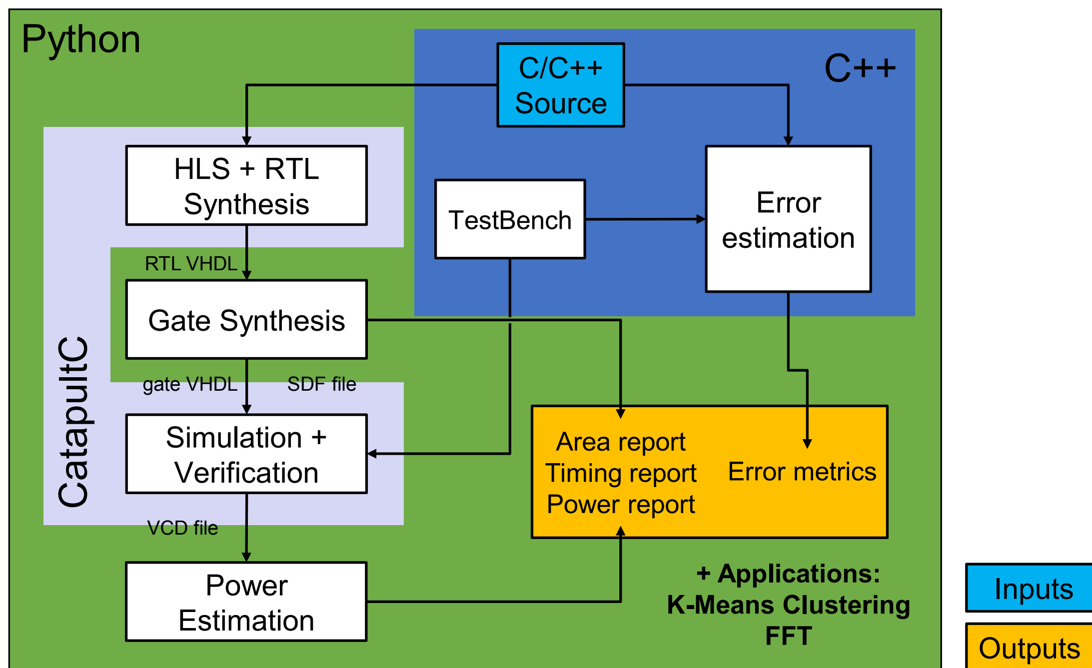


Figure 4.2 – Second version of ApxPerf

The HLS and the simulations are performed by Mentor Graphics *CatapultC*. During HLS, the Register Transfer Level (RTL) representation of the input source is generated. Then, a second compilation pass is ensured by *Design Compiler* to get a gate-level representation. The gate-level representation is then passed again to CatapultC for *Modelsim* simulation and verification using integrated *SystemC Verify* framework. Thanks to this framework, the same C++ test bench as for accuracy estimation is used for both hardware verification and generation of the *VCD* files for *PrimeTime* power estimation. This way, the statistical distribution of the generated test bench, which can be uniform or random with tunable parameters, is ensured to be the same for hardware performance and accuracy characterizations.

The accuracy estimation part returns the same error metrics as for the first version of ApxPerf described in the previous Section. The main novelty is the possibility to add any error metric to the error estimation as a plugin, with no need to modify the framework kernel. Another main evolution is the replacement of Bash and Matlab scripts by Python. This second version is consequently more portable. Moreover, except for the hardware characterization part requiring Mentor Graphics and Synopsys tools, the whole error estimation part, from simulation to results management, is not linked to any third-party software. The management of gen-

erated results and the generation of figures are performed using *Numpy* and *Matplotlib* packages of Python. An execution trace of APXPERF v2 characterization for 32-bit CT_FLOAT (see Chapter 5) is given by Listing 4.1.

Another important improvement in the framework is the integration of hardware performance characterization in embedded signal processing applications. At the time of writing this thesis, only K-means clustering and FFT were adapted from APXPERF v1.

Finally, the code of the approximate operators were replaced by a template-based C++ synthesizable library called APX_FIXED, based on Mentor Graphics AC Datatypes v3.7 under license Apache 2.0. This library included in APXPERF v2 features:

- synthesizable operator overloading:
  - unary operators: unary $-$, !, $++$, $--$,
  - relational operators: $<$, $>$, $<=$, $>=$, $==$, $!=$,
  - binary operators: $+$, $+=$, $--=$, $*$, $*=$ replaced by the approximate operators selected in the template values in the APX_FIXED instance declaration (see example below), $<<$, $<<=$, $>>$, $>>=$, and
  - assignment operator from/to an other instance of APX_FIXED.
- non-synthesizable operator overloading:
  - assignment operator from/to C++ native datatypes (float, double),
  - output operator $<<$ for easy display and writing in files.

APX_FIXED variables are represented by a fixed-point value which width, integer part width, rounding et wrapping modes (inherited from AC_FIXED) are parameterized in template as well as the name and parameters of the approximate operators to be used in additions and multiplications.

A use case of APX_FIXED library is given by Listing 4.2. In this example, the result of the operation $out = x \times y + z$ is computed. x and z are 8-bit integer numbers with 2-bit integer part in FxP representation, denoted as $(8, 2)$. y has $(10, 5)$ representation and the output out is represented on $(7, 2)$. In line 14, the APX_FIXED variables are initialized casting double precision floating-point values. The nearest representable value is set for each variable. E.g, x is set from 0.13236. As it only has 6-bit fractional part, its value is 0.125 (00.001000 in binary representation). In line 18, the operation is performed. Thanks to operator overload, the first operation $x \times y$ is performed using the approximate multiplier given in the template of x type declaration (first operand), which is classical fixed-point multiplication. As x and y are $(8, 2)$ and $(10, 5)$, the implicit result is stored in a number with representation $(18, 7)$ according to the rules discussed in Section 1.3.4. Then the implicit result is added to z. The sum of $(18, 7)$ and $(8, 2)$ representations returns an implicit $(19, 8)$ according to Section 1.3.3. The addition is performed using ACA$(18, 3)$ based on template values and the size of inputs. Finally the result is casted to the $(7, 2)$ number out. For this, the bits from position 6 to 12 of the result are extracted and put in out after truncation (because of the directive `AC_TRN` in out type `apx3_t`). This computation is fully synthesizable. During hardware optimization, the paths

Listing 4.1 – Execution trace of APXPERF v2 characterization for 32-bit CT_FLOAT

```
=================================================================
=========================   ApxPerf2.0   =========================
=================================================================
Extraction of user configuration... Done.

Operator information:
        Operator name: AddCtFloat
        Operator type: adder
        Inputs size: 32 bits
        Output size: 32 bits
        Parameters: N_exponent = 8      N_mantissa = 24 Q_mode = CT_RD
Clock period: 5 ns
Reset kind: synchronous
Technology target: 28nm FDSOI
Number of hardware simulation samples: 25000
Number of error simulation samples: 100000000

Copy of framework to temporary folder... Done.
Instrumentation of C++ source operator parameters... Done.
Search for custom hardware performance estimation test bench "sc_testbench.cpp"...
    Not found.
Search for custom error estimation test bench "err_testbench.cpp"... Not found.
Automated generation of missing test bench(es)...
- Detection of input and output types... Done.
        Input type: ct_float<8,24,CT_RD>
        Output type: ct_float<8,24,CT_RD>
- Instrumentation of hardware performance estimation test bench input generation...
    Done.
- Compilation of hardware performance estimation test bench input generation... Done.
- Generation of hardware performance estimation test bench inputs... Done.
- Instrumentation of hardware performance estimation test bench... Done.
- Instrumentation of error estimation test bench... Done.
Compilation of error estimation test bench... Done.
Execution of error estimation test bench... Done.
Copy of error estimation results to results folder... Done.
Instrumentation of Catapult C script... Done.
Execution of Catapult C script... Done.
Instrumentation of Design Compiler script... Done.
Detection of a previous compilation of technology libraries... Done.
Execution of Design Compiler script...  Done.
Instrumentation of SystemC Verify makefile... Done.
Compilation of SystemC Verify flow... Done.
Preparation of technology libraries for Modelsim... Done.
Instrumentation of Modelsim script for VCD generation... Done.
Execution of SystemC Verify flow... Done.
Instrumentation of PrimeTime script... Done.
Execution of PrimeTime script... Done.
Save of compiled technology libraries for next executions... Done.
Copy of gate-level VHDL, experiment parameters, reports and logs to results directory
    ... Done.
```

Listing 4.2 – APX_FIXED use case

```
1  #include<cstdlib>
2  #include"apx_fixed.h"
3  using namespace std;
4
5  int main(void) {
6      // Declaration of types (optional)
7      typedef apx_fixed<8,2,true,ac_q_mode::AC_TRN,ac_o_mode::
          AC_WRAP,apx_add::USE_ACA,3,-1,-1,-1,-1,apx_mul::
          USE_MULFXP,-1,-1,-1,-1,-1> apx1_t;
8      typedef apx_fixed<10,5,true,ac_q_mode::AC_TRN,ac_o_mode::
          AC_WRAP,apx_add::USE_ACA,3,-1,-1,-1,-1,apx_mul::
          USE_MULFXP,-1,-1,-1,-1,-1> apx2_t;
9      typedef apx_fixed<7,2,true,ac_q_mode::AC_TRN,ac_o_mode::
          AC_WRAP,apx_add::USE_ACA,3,-1,-1,-1,-1,apx_mul::
          USE_MULFXP,-1,-1,-1,-1,-1> apx3_t;
10
11     // Declaration of variables
12     double x_d = 0.13236, y_d = -1.54351, z_d = 0.75498;
13     double out_d;
14     apx1_t x = x_d; apx2_t y = y_d; apx1_t z = z_d;
15     apx3_t out;
16
17     // Double precision operation (non-synthesizable)
18     out_d = x_d * y_d + z_d;
19
20     // Approximate operation (synthesizable)
21     out = x * y + z;
22
23     // Displaying
24     cout << "accurate:\t" << x_d  << " * " << y_d << " + " <<
          z_d << " = " << out_d << endl;
25     cout << "approximate:\t" << x  << " * " << y << " + " << z
          << " = " << out << endl;
26
27     return EXIT_SUCCESS;
28 }
```

leading to the generation of the bits 13 to the MSB 18 would be pruned, as well as the bits from the LSB to bit position 5 because truncation does not consider them. As a matter of fact, changing rounding and overflow mode respectively to saturation and rounding-to-nearest in the template declarations would increase accuracy and hardware cost. Finally, line 25 gives the example of the overloading of output operator $<<$ in APX_FIXED. The code outputs given below reflects the successive casts and approximations.

```
accurate:       0.13236 * -1.54351 + 0.75498 = 0.550681
approximate:    0.125 * -1.5625 + 0.75 = 0.53125
```

The syntax developed in APX_FIXED library allows for fast and easy development and testing of approximate arithmetic kernels. The software flexibility of C++ and the efficiency of HLS tools allow for complex circuits to be produced and simulated with benchmarks, whose generation is easy to be fully automatized thanks to the overloading of type casting. A custom synthesizable floating-point library embedded by ApxPerf v2 called CT_FLOAT is described in the last chapter of this thesis.

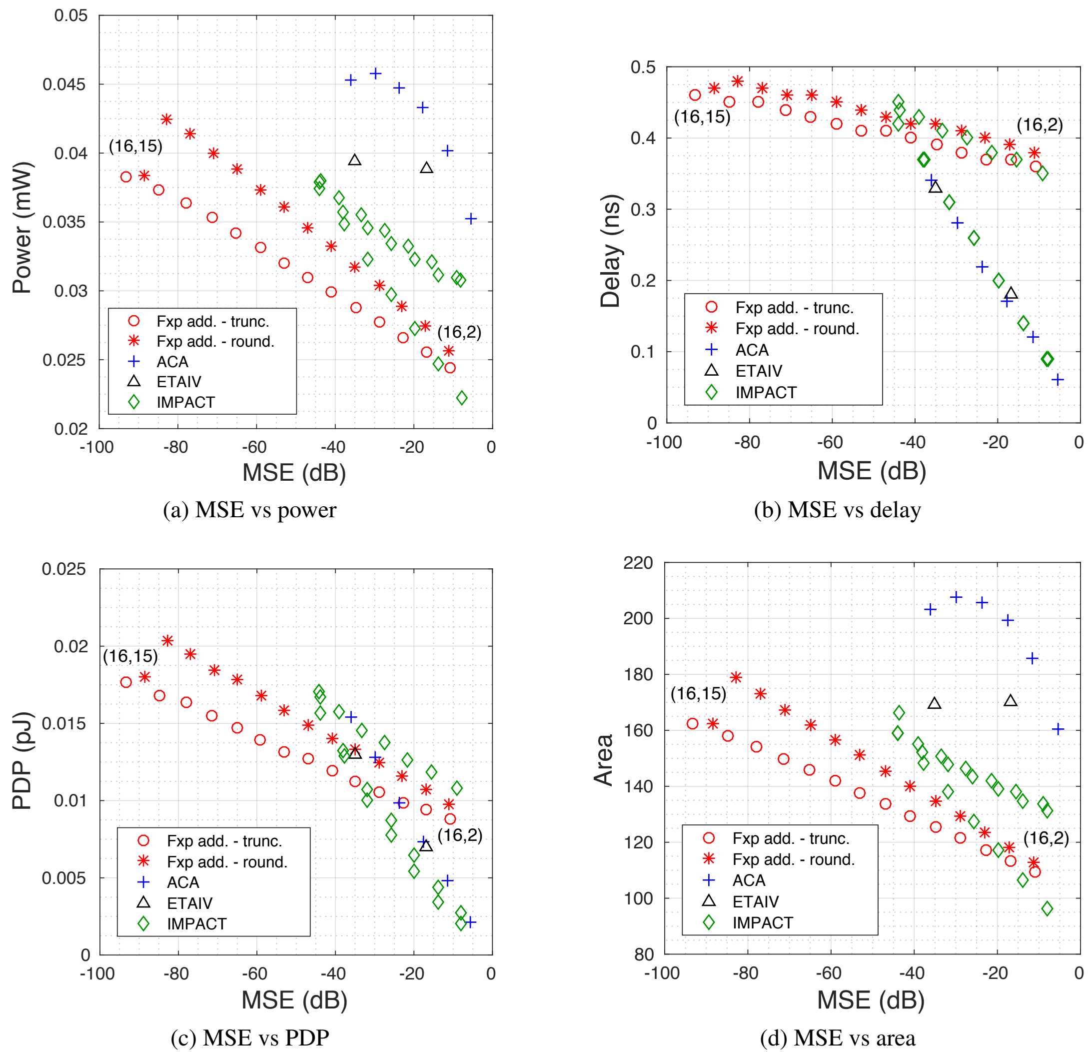
4.2 Raw Comparison of Fixed-Point and Approximate Operators

In this section, the results of the raw comparison between FxP and Apx operators are presented. They were all obtained using the first version of ApxPerfdescribed in Section 4.1.1 and were published in DATE'17 conference [2].

A first comparison of FxP and Apx operators consists in measuring the difference in their accuracy with regards to a performance metric (energy, area, delay). For this, approximate adders ACA, ETAIV and IMPACT and approximate multipliers AAMIII and FBMIII (respectively denoted as AAM and FBM in this section), described in Chapter 1, were compiled and tested with a number of bits varying from 2 to 32 and all possible combinations of parameters. FxP operators (i.e. classical integer adders and multipliers) were tested in the same way, with all possible combinations of input and output size (inputs from 2 to 32, outputs from 2 to 32 for the adders and 2 to 64 for the multipliers). All power results are given for a clock frequency of 100 MHz. With ApxPerf v1, Design Compiler (2013.03) was used for RTL synthesis with a 28nm FDSOI technology library, Modelsim (10.3c) for gate-level simulation and PrimeTime (2013.12) for power analysis. Simulation and power analysis are performed on 10^5 random input samples. The extraction of error metrics based on the C description was computed on more than 10^7 random inputs.

Results for adders are presented on Figure 4.3 and provide MSE versus power, area, delay, and PDP. For the sake of clarity, results are here only presented for 16-bit input operators. 16-bit output is considered as the correct adder and used as reference. Truncated and rounded FxP adder outputs vary from 15 to 2 bits (from left to right). For approximate adders, results are given by varying: the number of approximated LSBs (M) and types of FA for IMPACT, the maximal size of carry chain (P) for ACA, and the block size (X) for ETAIV.

What can be first noticed is that in terms of power consumption and design area, FxP operators are better than Apx operators for a same MSE, except for very-low accuracy. However, in terms of delay, most approximate operators are faster, but they cannot reach the same level



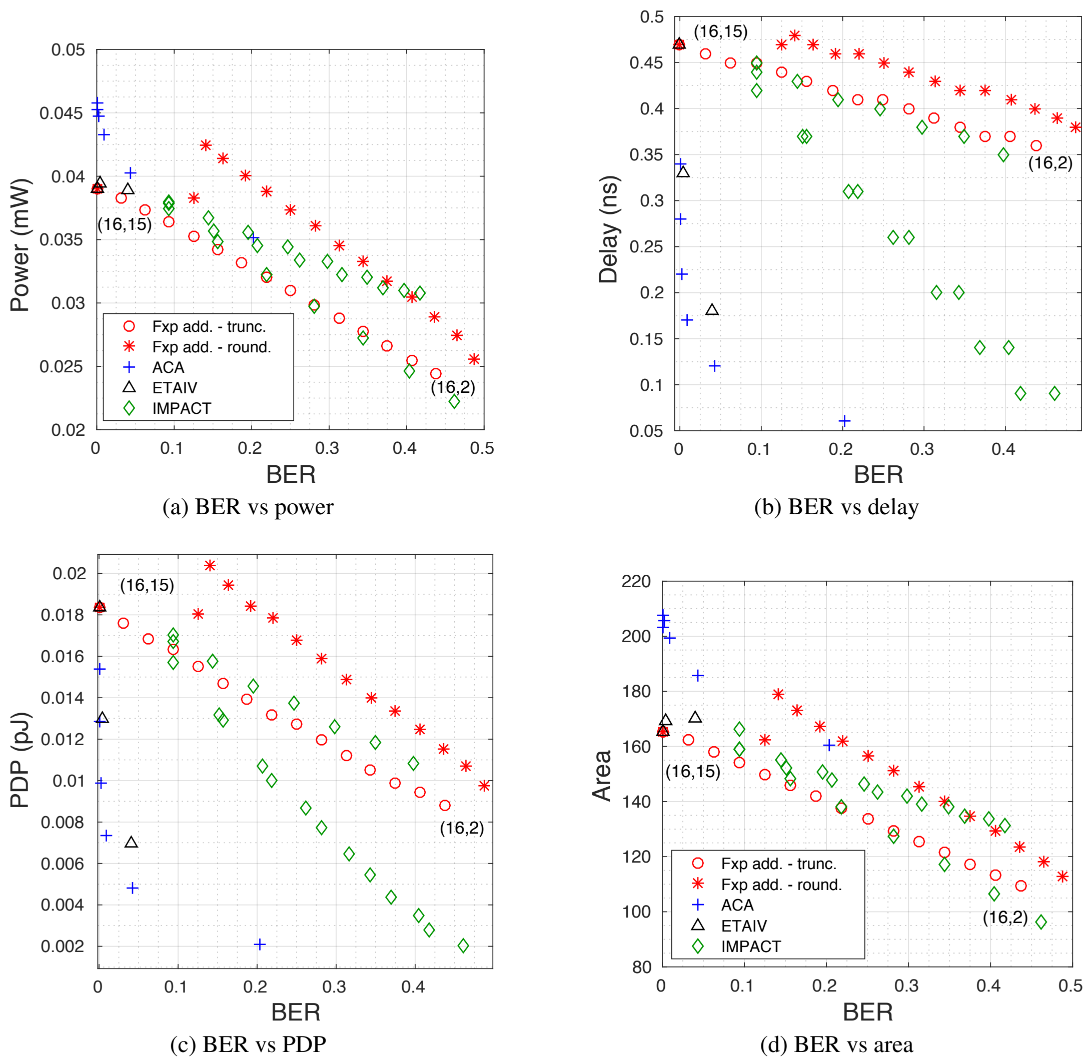
(a) MSE vs power

(b) MSE vs delay

(c) MSE vs PDP

(d) MSE vs area

Figure 4.3 – Direct comparison of 16-bit-input fixed-point and approximate adders regarding MSE



(a) BER vs power (b) BER vs delay

(c) BER vs PDP (d) BER vs area

Figure 4.4 – Direct comparison of 16-bit-input fixed-point and approximate adders regarding BER

	$\mathrm{MUL}_t\,(16,16)$	$\mathrm{AAM}\,(16)$	$\mathrm{FBM}\,(16)$
Power (mW)	0.273	0.359	0.446
Delay (ns)	0.91	1.23	0.57
PDP (pJ)	0.249	0.442	0.446
Area (μm^2)	805.2	665.5	879.5
MSE (dB)	-89.1	-87.9	-9.63
BER (%)	23.4	27.7	27.9

Table 4.1 – Direct comparison of 16-bit-input and output fixed-point and approximate multipliers

of accuracy when more than 8 bits are kept for the fixed-point output. In terms of energy, the PDP of FxP adders is quite near from approximate ones when less than 8 output bits are kept. However, ACA and IMPACT are able to spend less energy without sacrificing much accuracy.

In some applications, all output bits have the same weight on the error. Therefore results on BER metric are presented on Figure 4.4 for the same adders than previously. Approximate operators achieve very good BER performance when compared to fixed-point operators. Considering the power and area, truncated and IMPACT adders perform similarly for any fixed BER. However, for delay and energy per addition, most approximate operators perform significantly better truncated or rounded FxP operators. When not considering bit significance in the operands, FxP operators are penalized by the suppression of part of their output bits, implicitly forcing them to zero.

Results for the multipliers are presented in Table 4.1. The 16 to 32 integer multiplier is considered as the correct multiplier for accuracy reference. As AAM and FBM multipliers are fixed-width operators (16-bit inputs and output), comparison results are provided only for the truncated FxP multiplier with 16-bit output ($\mathrm{MUL}_t(16,16)$). Fixed-width MUL_t truncated multiplier reaches the best accuracy, and consumes least power. Although MUL_t is slower than FBM, its energy per multiplication (PDP) is 44% better than both approximate operators. FBM is 37% faster than MUL_t and AAM is 17% smaller. However, FBM is extremely MSE inaccurate, with 7 orders of magnitude more erroneous results than fixed-point. Both AAM and FBM are worse than MUL_t by about 19% for BER metric.

As a conclusion for this operator-level performance analysis of various approximation schemes, fixed-point operators perform better when considering the MSE metric representative of signal processing applications, while approximate adders show good BER performance. However, the importance of output bit-width was not taken into account in this results. Indeed, when the bit-width is reduced, as in truncated or rounded operators, the amount of data to transfer to load and store operator inputs and output is consequently reduced. This shortening in bit-width has a major impact on energy consumption and must be considered for real-life application. Thus, although inexact and truncated-or-rounded operators seem to reveal the same gross performance, selecting the second one will allow to decrease energy cost by avoiding the transfer and memory storage of useless erroneous bits.

4.3 Comparison of Fixed-Point and Approximate Operators on Signal Processing Applications

In this Section, the effect of fixed-point and approximate adders and multipliers is evaluated on different real-life applications, leveraging relevant and adapted metrics. Considered applications include Fast Fourier Transform (FFT), JPEG image encoding, motion compensation filtering in the context of High-Efficiency Video Coding (HEVC) decoding, and K-means clustering.

4.3.1 Fast Fourier Transform

As a classical computation kernel used in many signal processing or communication applications, FFT is relevant for this study. APXPERF v1 provides an instrumented, tunable FFT kernel. This sections presents results on a 32-sample Radix-2 FFT computed on 16-bit input/output data. In a first experiment, only the adders are considered. The total energy to compute the FFT is estimated by:

$$PDP_{FFT} = \sum_{i=1}^{N_{add}} PDP_{add,i} + \sum_{i=1}^{N_{mul}} PDP_{mul,i} \tag{4.1}$$

where N_{add} and N_{mul} are the total of additions and multiplications, respectively. Figure 4.5 shows PDP_{FFT} as a function of output Peak Signal-to-Noise Ratio (PSNR). PSNR is the maximal power of the output signal divided by the MSE, i.e:

$$PSNR(x)[dB] = 10.\log\left[\frac{\max(x^2)}{MSE(x)}\right]. \tag{4.2}$$

The exact multipliers used alongside the modified adders are optimally sized according to the adder bit-width, so they are not source of error. For any accuracy constraint, FxP adders (truncation or rounding) notably dominate approximate adders. This supremacy could be explained by two factors: the relative energy cost of multipliers with regards to adders and the need for less operand size for the multiplier when reducing the accuracy of additions. This figure also shows the great potential of energy reduction when playing with accuracy of the fixed-point operators. A first conclusion here is that reducing the FxP adder size provides a smaller entropy of the data processed, transported and stored, than keeping the same bit-width along the computations but containing errors in data since computations rely on approximate operators. The same experiment is performed using 16-bit AAM and FBM multipliers and a 16-bit truncated FxP multiplier, while keeping 16-bit exact adders. Table 4.2 shows that AAM and FxP multipliers differ only by 6 dB of of accuracy. However, AAM consumes 78% more energy that reduced-precision fixed-point equivalent. Results on the FFT comfort the conclusion of Section 4.2. Providing results with both approximate adders and multipliers in the same simulation will not lead to a different conclusion.

4.3.2 JPEG Encoding

The second application is a JPEG encoder, representative of the image processing domain. The main algorithm of this encoder is the Discrete Cosine Transform (DCT). To obtain an approx-

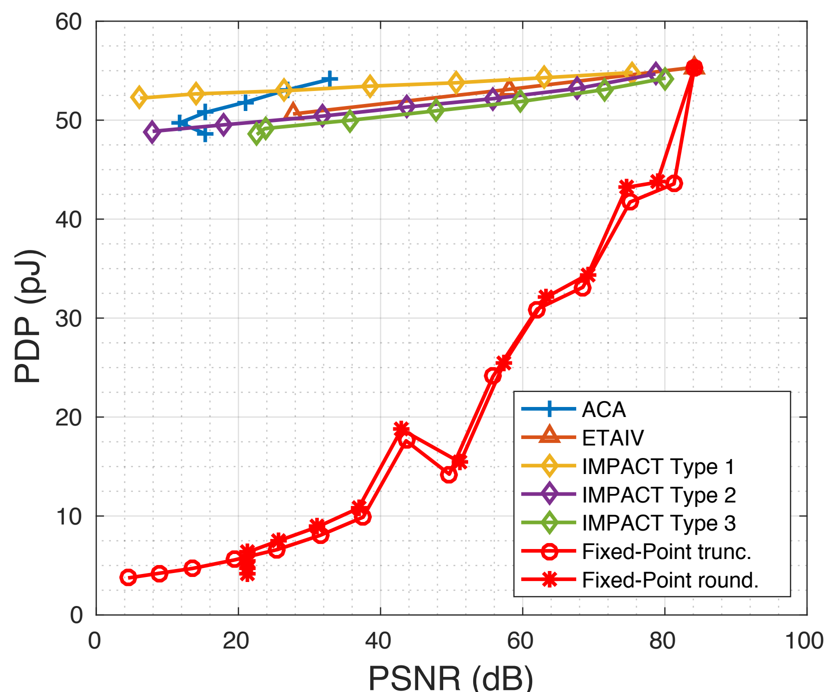


Figure 4.5 – Power Consumption of FFT-32 Versus Output PSNR Using 16-bit Approximate Adders

	$\text{MUL}_t\,(16, 16)$	$\text{AAM}\,(16)$	$\text{FBM}\,(16)$
PSNR (dB)	53.88	59.66	-18.14
PDP (pJ)	55.43	92.49	93.26

Table 4.2 – Accuracy and Energy Consumption of FFT-32 Using 16-bit Fixed-Width Multipliers

imate version of the encoder, DCT operations are computed using fixed-point or approximate operators. The quality metric to compare the exact and the approximate versions of the JPEG encoder is the Mean Structural Similarity (MSSIM) [67], which is representative of the human perception of image degradation. This metric results in a score between $[0, 1]$, 1 representing a perfect quality. To obtain Figure 4.6, the DCT energy consumption is compared for all presented approximate adders, as well as for fixed-point versions. The algorithm is applied with an encoding effort of 90% on the image $Lena$. As observed for the FFT, the fixed-point versions of the algorithm are much more energy efficient than for approximate operators, mostly thanks to the bits dropped during the calculation.

It is important to notice that the nature of the error generated by approximate operators (few high amplitude errors in general) is very problematic in the context of image processing. Figures 4.7 shows $Lena$ with four different approximations in the DCT encoding. On Figure 4.7a, the additions are replaced by a 16-bit accurate adder with the output truncated to 10 bits. On

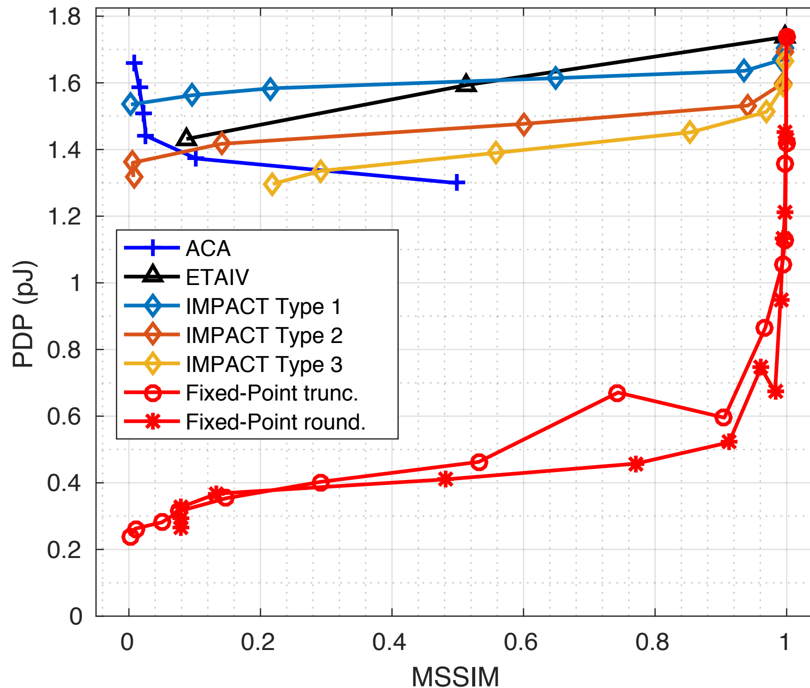
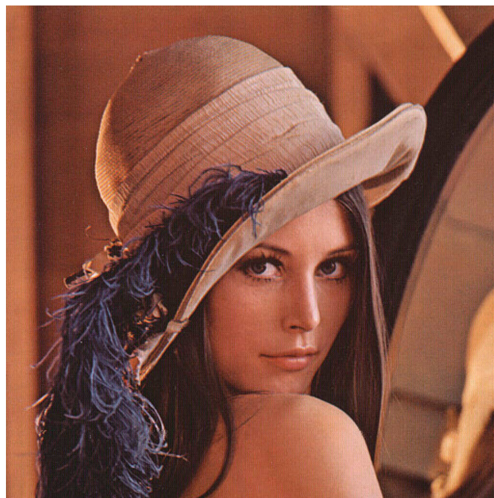


Figure 4.6 – Power Consumption of DCT in JPEG Encoding Versus Output MSSIM Using 16-bit Approximate Operators

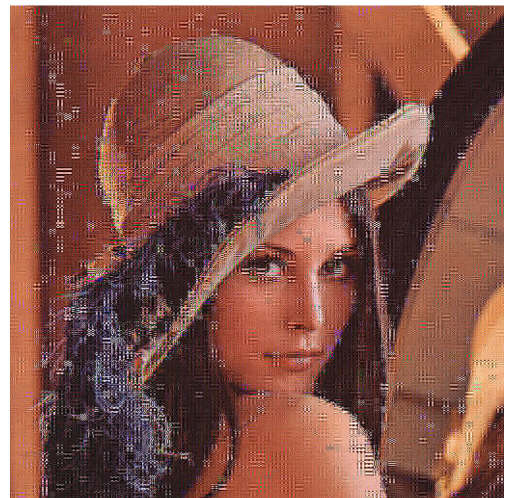
Figure 4.7b, the additions are performed with a 16-bit ACA with $x = 2$, meaning the carries are limited to a consideration of the two previous LSBs. On Figure 4.7c, the additions are replaced by a 16-bit ETAIV made of 4-bit blocks. Finally, Figure 4.7d is generated using 16-bit IMPACT with 8-bit exact addition and 8-bit approximate addition using approximation number 3 (see Section 1.4.1.5). The best visual result clearly occurs for the fixed-point addition. The result of IMPACT adder is also quite good thanks to its 8-bit accurate adder on the MSB part, but as showed above, its larger output implies an important energy overhead. Moreover, some visual artefacts are present in sharp details. For ETAIV, the image is quite degraded with horizontal and vertical lines giving a noisy appearance to the image. For ACA, strong artefacts are visible everywhere.

4.3.3 Motion Compensation Filter for HEVC Decoder

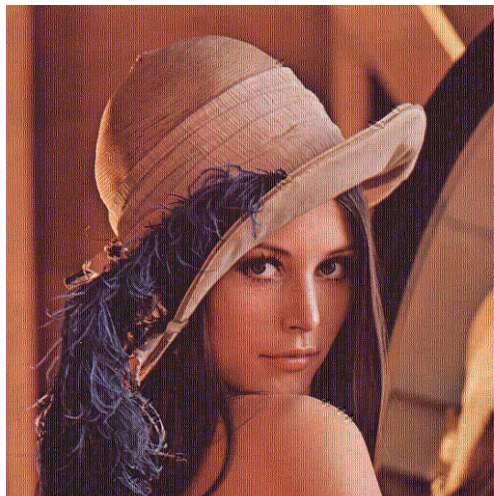
HEVC is the new generation of video compression standard. Efficient prediction of a block from the others requires fractional position Motion Compensation (MC) carried-out by interpolation filters. These MC filters are modified using fixed-point and approximate operators to test their accuracy and energy efficiency. Previously described MSSIM metric is used to determine the output accuracy of the filter on a classical signal processing image. Table 4.3 gives the energy spent by the MC filter replacing all its additions by adders producing an MSSIM of approximately 0.99. In their 16-bit version, ACA and ETAIV can only reach respectively



(a) $\text{ADD}_t(16, 10)$



(b) $\text{ACA}(16, 2)$



(c) $\text{ETAIV}(16, 4)$

(d) $\text{IMPACT}(16, 8, 3)$

Figure 4.7 – *Lena* Encoded with DCT Instrumented With Different 16-bit Approximate Operators

0.96 and 0.98. In any case, and as discussed above, the multiplier overhead provokes an energy consumption which is 4.6 times superior for the approximate version than for the truncated FxP version. For multiplier replacement, Table 4.3 shows that both 16-bit AAM and FBM produce an accuracy similar to fixed-width truncated FxP multiplier. Moreover, replacing multipliers by FBM in the MC filter do not lead to an important energy overhead, which makes it competitive considering that its delay is 37% inferior to $\mathrm{MUL}_t(16,16)$ according to Table 4.1. However, AAM suffers from an important energy overhead.

	MSSIM	Adder Energy (pJ)	Min. Mult. Energy (pJ)	Total Energy (pJ)
$\mathrm{ADD}_t(16,10)$	99.29%	1.39E−2	4.39E−2	0.898
$\mathrm{ACA}(16,12)$	96.45%	1.54E−2	2.49E−1	4.20
$\mathrm{ETAIV}(16,4)$	98.02%	1.30E−2	2.49E−1	4.17
$\mathrm{IMPACT}(16,6,3)$	99.67%	1.00E−2	2.49E−1	4.12

Table 4.3 – Accuracy and Energy Consumption of Distance Computation for HEVC Filter Using 16-bit Input Adders

	MSSIM	Multiplier Energy (pJ)	Min. Adder Energy (pJ)	Total Energy (pJ)
$\mathrm{MUL}_t(16,16)$	99.918%	2.49E−1	1.83E−2	3.77
$\mathrm{AAM}(16)$	99.909%	4.42E−1	1.83E−2	6.48
$\mathrm{FBM}(16)$	99.907%	2.54E−1	1.83E−2	3.85

Table 4.4 – Accuracy and Energy Consumption of Distance Computation for HEVC Using 16-bit Input Multipliers

4.3.4 K-means Clustering

The last experiment presented in this section is K-means clustering. Given a bidimensional point cloud, this algorithm classifies them finding centroids and assigning each point to the cluster defined by the nearest centroid. At the core of K-means clustering is distance computation. More details about K-means clustering are given in Section 5.3.1. For the experiment, 5 sets of 5.10^3 points of data were generated around 10 random points with a Gaussian distribution. The accuracy metric is the success rate, from 0 to 1, representing the proportion of points classified in the correct cluster. Table 4.5 presents the success rate and energy spent in distance computation replacing the exact adders by fixed-point and approximate versions. For truncated fixed-point version, the energy spent in multiplication is inherently inferior to approximate, thanks to the reduction of bit-width, leading to a total energy reduction by more than half for a success rate of 99%, and even by nearly 10 for a success rate of about 86%. Table 4.6 shows K-means clustering classification success rate and energy spent in distance computation performing multiplication using 16-bit input FxP and approximate multipliers.

	Success Rate	Adder Energy (pJ)	Min. Mult. Energy (pJ)	Total Energy (pJ)
$ADD_t(16,11)$	99.14%	1.55E−2	9.36E−2	2.03E−1
$ACA(16,12)$	99.10%	1.54E−2	2.49E−1	5.13E−1
$ETAIV(16,4)$	99.43%	1.30E−2	2.49E−1	5.11E−1
$IMPACT(16,6,3)$	99.67%	1.00E−2	2.49E−1	5.08E−1
$ADD_t(16,8)$	86.00%	1.27E−2	2.40E−2	6.06E−2
$ACA(16,8)$	86.06%	9.85E−3	2.49E−1	5.08E−1
$ETAIV(16,2)$	63.25%	7.00E−3	2.49E−1	5.05E−1
$IMPACT(16,10,1)$	87.29%	1.26E−2	2.49E−1	5.11E−1

Table 4.5 – Accuracy and Energy of Distance Computation for K-means Clustering Using 16-bit Input Adders for Different Success Rates

AAM achieves similar accuracy than fixed-width truncated accurate multiplier, with 99% classification success rate. However, it presents an energy overhead of 75%. FBM achieves very poor performance for K-means, with only 10% success, which is equivalent to prune 12 output bits of a FxP multiplier.

	Success Rate	Multiplier Energy (pJ)	Min. Adder Energy (pJ)	Total Energy (pJ)
$MUL_t(16,16)$	99.84%	2.49E−1	1.83E−2	5.15E−1
$AAM(16)$	99.43%	4.42E−1	1.83E−2	9.02E−1
$FBM(16)$	10.27%	2.54E−1	1.83E−2	5.27E−1
$MUL_t(16,4)$	10.87%	2.04E−1	1.24E−3	4.09E−1

Table 4.6 – Accuracy and Energy of Distance Computation for K-means Clustering Using 16-bit Input Multipliers

In spite of the theoretical competitiveness of approximate operators, their advantages are likely to be lost at application level. Indeed, at the difference of fixed-point operators, accuracy reduction is obtained by simplifying the operator structure but not by reducing the operator output bit-width. This reduces the energy of the considered operator, but does not have a positive impact on the other operators, as it is the case for fixed-point.

4.4 Considerations About Arithmetic Operator-Level Approximate Computing

In this Chapter, two types of hardware approximation were compared: fixed-point arithmetic relying on truncated and rounded accurate operators with careful data sizing, and approximate operators. A direct comparison using the ApxPerf framework showed that both techniques are competitive, depending on the observed error and performance metrics.

However, stepping back observing some state-of-the-art applications showed that using approximate operators often leads to an important overhead since the designed architecture manipulates larger data containing in average more erroneous useless information bits. Approximate operators output showing high error entropy compared to quantized exact output for which all the error is virtually contained in dropped bits, the error generated by approximate operators may have a huge impact in downstream operations using their output. Mathematically, it is always preferable to propagate many low significance errors (symbolized by dropped bits in fixed-point paradigm) than scarcer high significance errors. Indeed, in most approximate operators proposed in literature, the probability for high significance errors to occur is never negligible, leading to errors with amplitudes as high as the represented dynamic.

More generally, it has been shown that comparing the raw performance of operators does not necessarily reflect their performance in a given application context. Hence, a major stake for hardware approximation is now to be considered at a higher level to take more parameters into consideration, leveraging relevant error metrics. An effort must also be made in the research for more efficient approximate multipliers, since they are responsible for the majority of power consumption in computation-intensive applications. However, considering approximate operators in embedded processors to replace or enhance their integer arithmetic unit might still be a good option, since processor data size is fixed and cannot be application specific.

After a conclusion in favor of fixed-point compared to functional approximate arithmetic operators, the studies lead in next chapter drop approximate operators to compare fixed-point and custom floating-point paradigms.

Chapter 5

Fixed-Point Versus Custom Floating-Point Representation in Low-Energy Computing

In Chapter 4, we compared classical fixed-point arithmetic with operator-level approximate computing. The general conclusion was the superiority of fixed-point arithmetic thanks to lower error entropy making error more robust to deterioration in propagation.

In this Chapter, fixed-point arithmetic is compared to custom floating-point arithmetic. As a reminder, fixed-point arithmetic is presented in Section 1.3 and floating-point arithmetic in Section 1.2. To perform this comparison, the study was led using the second version of APXPERF, described in Section 4.1.2. This version embeds a synthesizable custom floating-point library called CT_FLOAT presented in Section 5.1 and developed in the context of this thesis. In this section, CT_FLOAT is compared to other custom floating-point libraries to first show its efficiency. Then, stand-alone fixed-point and floating-point paradigms are compared in Section 5.2 to appreciate their differences in terms of accuracy and hardware performance. Finally, in Section 5.3, both representations are compared on signal processing applications, K-means clustering and FFT, leveraging relevant metrics.

5.1 CT_FLOAT: a Custom Synthesizable Floating-Point Library

The second version of APXPERF framework was presented in Section 4.1.2. It allows for fast and user-friendly hardware characterization of approximate operators written in C++ thanks to HLS, leveraging Catapult C, Design Compiler, ModelSim and PrimeTime, and error characterization thanks to C++ benchmarks. As mentioned in Section 4.1.2, APXPERF v2 comes with built-in approximate operators libraries such as APX_FIXED containing approximate integer adders and multipliers in fixed-point representation. This section presents CT_FLOAT, the main operator library of APXPERF v2.

5.1.1 The CT_FLOAT Library

CT__FLOAT is the template-based synthesizable C++ custom library embedded into APXPERF v2 used in this Chapter. There exist two versions of it:

- a first version based on Mentor Graphics AC__INT datatype, made for Catapult HLS but also stand-alone error estimation and
- a second version based on Xilinx Vivado HLS integer library AP__INT made for Xilinx FPGA target using Vivado.

Both versions are provided in the same source code and activated through C++ pre-compiler directives. The implementation of CT__FLOAT, as for APX__FIXED, features:

- Synthesizable operator overloading:
  - unary operators: unary $-$, !, $++$, $--$,
  - relational operators: $<$, $>$, $<=$, $>=$, $==$, $!=$,
  - binary operators: $+$, $+=$, $- -=$, $*$, $*=$, $<<$, $<<=$, $>>$, $>>=$, and
  - assignment operator from/to another instance of CT__FLOAT.
- Non-synthesizable operator overloading:
  - assignment operator from/to C++ native datatypes (float, double),
  - output operator $<<$ for easy display and writing in files.

Other built-in functions allow easy manipulation of floating-point values, such as test functions to get information about the extreme representable values for a given floating-point representation, to know if a given value is representable, etc. The declaration of an instance of CT__FLOAT requires three template parameters:

1. the exponent width e,
2. the mantissa width m, and
3. the rounding mode used in arithmetic operators and changes of representation. Currently, four rounding modes are available, given by Table 1.5 in Chapter 1.

The value of mantissa width m includes the implicit 1 (see below). The representation also includes a sign bit. Therefore, the total number of bits in memory is equal to $e + m$.

As mentioned above, two synthesizable operators are available: addition and multiplication. Unlike APX__FIXED, the output representation of these operators is not determined to be prevented from under/overflows. Indeed, if the inputs are on (e_1, m_1) and (e_2, m_2) representation, the output representation (e_o, m_o) is given by:

$$\begin{aligned} e_o &= \max(e_1, e_2) \\ m_o &= \max(m_1, m_2)\,. \end{aligned} \tag{5.1}$$

If the rounding modes of both inputs are different (discouraged), then the first one parsed in the C++ code is selected.

CT_FLOAT representation and arithmetic operators were created to remain simple for energy efficiency yet minimally secured, thanks to the combination of several choices. First, CT_FLOAT mantissa is represented in $[1, 2[$ with an implicit 1. This allows a 1-bit accuracy benefit in general. However, subnormal numbers are not handled, which implies that a certain range of numbers are not representable around 0. The exponent is represented in a biased representation. The bias is not customizable for the moment and is set at the center of the exponent range like in IEEE 754 representation. Using biased representation instead of two's-complement results in simpler exponent value comparisons, which are omnipresent in arithmetic operators.

An important choice is that no flag bits are returned. In general, such flags are returned to indicate zero-case or subnormal cases for further management. However, these flags imply more bits to transfer in hardware (generally two), and the pipeline may be broken during the management of the corresponding special cases, leading to extra-energy consumption. In return for not raising zero-case flag, the number zero is directly managed by the arithmetic operators. For this, the value 0 must be representable. To represent 0, the nearest representable negative value from 0 is used. One representable value is sacrificed, but it does not imply any change in comparison operators for instance. A slight overhead is then necessary in the arithmetic operators to detect the 0 value at the input. For the addition/subtraction, if one of the inputs is worth 0, the second is selected. No extra-delay is implied since a simple short path may exist in parallel to the original adder. At the addition/subtraction output, the special value representing zero must be issued when the computed output mantissa is a vector of 0, leading to a slight control overhead. It is also insured that only two strictly opposed added values can issue 0. For the multiplication, 0 detection at the input returns 0 value, which only implies a very small overhead. It is insured that only a multiplication by 0 can return 0.

As subnormals are not representable by CT_FLOAT, the output is always saturated to the smallest absolute possible representable value with the same sign. Towards infinity, the operators do not under/overflow. Saturation to the highest absolute representable value of same sign is returned.

The combination of these choices implies a slight overhead in the operators which is not spent in external hardware management. The local management implies less data stored or long-distance transferred, which represents global energy savings.

A use case of CT_FLOAT is given by Listing 5.1. In this example, an FIR filter is applied to random data. First, on line 9, an array of CT_FLOAT is initialized with the impulse response of the filter. The coefficients are parsed to 16-bit CT_FLOAT with $e = 7$ and $m = 9$. Then, input and output signals x and y are declared on line 13. Random inputs are generated in double representation, parsed to $\texttt{ct_float}(7, 9)$ and stored in array x. The random generation presented in this example is not synthesizable. Then, the synthesizable FIR filter is applied. Additions and multiplication are overloaded with the operators developed in CT_FLOAT library. Finally, the resulting FIR filter is displayed using display operator $<<$ overloading.

Listing 5.1 – Synthesizable CT_FLOAT FIR filter

```
#include<cstdlib>
#include<ctime>
#include"ctfloat.h"
using namespace std;

#define N_FIR 9
#define N_DATA 50

ct_float<7, 9, CT_RD> h[N_FIR] = { -1.55107884796477e-18,
    -0.0226639854595526, 1.04697822237622e-17,
    0.273977082565524, 0.497373805788057, 0.273977082565524,
    1.04697822237622e-17,
        -0.0226639854595526, -1.55107884796477e-18 };

int main(void) {
    ct_float<7, 9, CT_RD> x[N_DATA], y[N_DATA - N_FIR];

    srand(time(NULL));

    // Generation of uniform random data in [-1,1] - not
        synthesizable
    for (int i = 0; i < N_DATA; i++) {
        x[i] = ((double) rand() / (double) RAND_MAX);
        x[i] = (rand() % 2) ? x[i] : -x[i];
    }

    // Filtering - synthesizable
    for (int i = 0; i < N_DATA - N_FIR; i++) {
        y[i] = x[i] * h[N_FIR - 1];
        for (int j = 1; j < N_FIR; j++) {
            y[i] += x[i + j] * h[N_FIR - j - 1];
        }
    }

    //Displaying result - not synthesizable
    cout << "FIR Filter impulse response:" << endl;
    for (int i = 0; i < N_FIR; i++) {
        cout << h[i] << endl;
    }
    cout << endl;

    cout << "Outputs:" << endl;
    for (int i = 0; i < N_DATA-N_FIR; i++) {
        cout << y[i] << endl;
    }

    return EXIT_SUCCESS;
}
```

The execution of Listing 5.1 results in:

```
FIR Filter impulse response:
-1.54838e-18    -0.022644       1.04626e-17     0.273438        0.49707 0.273438
            1.04626e-17     -0.022644       -1.54838e-18
Outputs:
0.169434        -0.0915527      -0.0427246      0.464844        0.509766
    0.0458984       0.0330811       0.582031        0.789062        0.388672
    -0.14209        -0.325195       0.0435791       0.535156        0.435547
    -0.0112305
```

APXPERF also comes with a random number generation library. This simplifies the generation of integer and floating-point values, using uniform or normal distributions with user-selected parameters. Several functions are available to guarantee that the generated random values are in the range of the representable values of the considered integer, fixed-point or floating-point operator.

For custom floating-point operators such as CT_FLOAT, it is also possible to generate couple of inputs which guarantee the activation of the close path of addition (see Section 1.2.2) with a given probability. It is important to have this path activated for a certain percentage of input couples to perform fair Monte Carlo time-based dynamic power estimation, as it is the most energy-costly computingnpath of the addition. However, when performing totally Monte Carlo simulation using inputs uniformly distributed on the whole range of the representable value of a floating-point operator, the close path only has a very low probability to be activated. Therefore, it is important to consider this feature in the generation of random inputs.

In the next section, stand-alone performance of CT_FLOAT facing other custom floating-point libraries is evaluated, using APXPERF and its random number generation library.

5.1.2 Performance of CT_FLOAT Compared to Other Custom Floating-Point Libraries

Reconfigurable architectures like FPGA are more and more used in many domains. The most recent and impacting example is the FPGA chip found in Apple's iPhone 7 and suspected to be used for artificial intelligence [68]. These conditions illustrate the interest of customizable floating-point architectures. Indeed, combining the ease of use of floating-point representation associated to low-energy small data width make these architectures very promising for the future of reconfigurable architectures. The past years have hosted the creation of several customizable floating-point libraries.

Mentor Graphics, in its synthesizable C++ libraries AC Datatypes [69], proposes the custom floating-point class AC_FLOAT. Based on the fixed-point library AC_FIXED, AC_FLOAT allows for light floating-point computation, thanks to simple operators. The mantissa in the representation is not normalized and has no implicit 1. This allows for easy management of subnormals, but induces a potential loss of accuracy in computations. The mantissa is represented in signed two's complement, so the sign information is contained in the mantissa instead of using an extra sign bit. However, there is no benefit to this choice since two's complement represents a loss of 1 bit of precision compared to unsigned representation. The choice of two's complement representation on the mantissa also turns comparison operator more com-

plex. Moreover, many cases are not handled such as zero or infinity. AC_FLOAT also supports custom exponent bias, but managing the exponent bias comes with an overhead.

CT_FLOAT, presented in Section 5.1.1 and embedded in APXPERF, offers a balance between computational safety and simplicity. Inspired by AC_FLOAT, it is written in C++ for HLS. Two versions of CT_FLOAT do exist, one based on Mentor Graphics AC_INT data type, made for Mentor Graphics Catapult C, and the other one based on AP_INT data type from Xilinx, used in Vivado for FPGA targets.

FLOPOCO (for Floating-Point Cores, but not only) is a generator of arithmetic cores [70]. Also based on C++, it has its own synthesis engine and directly returns VHDL. More than simple arithmetic operators, it is able to generate optimized floating-point computing cores performing complex arithmetic expressions. In this Section, we will only get interested in FLOPOCOś custom floating-point addition and multiplication. The main difference of FLOPOCO's floating-point representation is the extra 2-bit exception field transported in data. Like for CT_FLOAT subnormals are not handled by FLOPOCO. Unlike AC_FLOAT both CT_FLOAT and FLOPOCO do not support custom exponent bias.

Other alternatives such as VFLOAT [71, 72] or OptiFEX [73] do exist but are not taken into account in the study led in this chapter. VFLOATproposes IEEE 754-2008 compliant customizable computing cores for existing FPGA. OptiFEX generates floating-point computing cores targeting FPGA like FLOPOCO.

Table 5.1 recapitulates the different known properties of AC_FLOAT, CT_FLOAT and FLOPOCO floating-point representation. In this table, the number of additional bits in the representation is taking for reference a representation with implicit 1 in the mantissa and with one bit of sign in the representation. For an equal general accuracy, AC_FLOAT needs one more bit on the mantissa than CT_FLOAT and FLOPOCO. However, with its 2-bit exception field, FLOPOCO has the representation requiring the largest width, but also the highest computing reliability.

The hardware performance comparison process for AC_FLOAT, CT_FLOAT and FLOPOCO is as follows. All operators are characterized for an Application Specific Integrated Circuit (ASIC) target in 28nm FDSOI @ 1.0V, 25C. All designs are generated with a clock of 200 MHz. As AC_FLOAT and CT_FLOAT are compatible with APXPERF v2, this framework is used to perform the hardware performance characterization process. For time-based power analysis, the random inputs generated for adder/subtracter characterization are ensuring an activation of the close path for at least 50% of the computations. For FLOPOCO, VHDL files of the operators and test benches are generated using Stratix IV target and disabling all possible hardware acceleration which could allocate DSPs blocks used in FPGAs. Then, the design is compiled using Design Compiler, characterized using FLOPOCO's generated benchmark in ModelSim, and power is estimated using PrimeTime. However, to our knowledge, the benchmark generated by FLOPOCO does not insure any proportion of activation of the close path, so the time-based estimated dynamic power could be underestimated. FLOPOCO's benchmark top design does not consider the input and output data registers, whereas APXPERF generated benchmark does. This grossly represents about 5 to 10% underestimation in the total power for FLOPOCO operators, which has to be kept in mind for the analysis of results. All operators are

	AC_FLOAT	CT_FLOAT	FLOPOCO
Custom exp. bias	✓	✗	✗
Mantissa Implicit 1	✗	✓	✓
Zero and inf. exception flags	✗	✗	✓
Zero and inf. internal handling	✗	✓	✗
Subnormal exception flag	✗	✗	✓
Subnormal internal handling	✓	✗	✗
Additional bits in representation	+1	+0	+2

Table 5.1 – Main Properties of Custom Floating-Point Libraries AC_FLOAT, CT_FLOAT and FLOPOCO

generated so they execute in 1 cycle. It may not be the most efficient implementation because of possible glitches, but it is a good starting point for a fair comparison.

To estimate the energy spent per operation, we also introduce a fair metric, which is the *total energy spent before stabilization*. Indeed, in literature, energy per operation is often estimated whether:

- using the total energy per clock cycle, or
- using Power-Delay Product (PDP), which is the multiplication of the average dynamic power of the operation.

In the first case, a fair comparison between two different operators is strongly dependent on their difference of slack. Indeed, let us imagine two operators op_1 and op_2 which have the same size and the same static power. if op_1 stabilizes twice as fast as op_2 with a same dynamic power, then we would naturally tend to say that $E(op_1) = \frac{1}{2} \times E(op_2)$. However, with the total energy per clock cycle metric, $E(op_1)$, if the slack is high, then op_1 and op_2 will seem to have very close energy per operation, which is indeed false. With the PDP metric, the static power is not considered. Therefore, if op_1 and op_2 have very different static power, which is true if they do not have quite equivalent area, then the energy per operation will be too much in favor of the larger operator.

To be perfectly fair, the energy per operation must consider the whole energy (static and dynamic) spent before stabilization such as depicted in Figure 5.1. Considering the average static power P_{s}, the average dynamic power P_{d}, the critical path delay T_{cp}, the clock period $T_{\texttt{clk}}$ and the number of latency cycles N_{c}, the total energy spent before stabilization E_{op} is

$$E_{\text{op}} = P_{\text{s}} \times T_{\text{cp}} \times N_{\text{c}} + P_{\text{d}} \times T_{\texttt{clk}}. \quad (5.2)$$

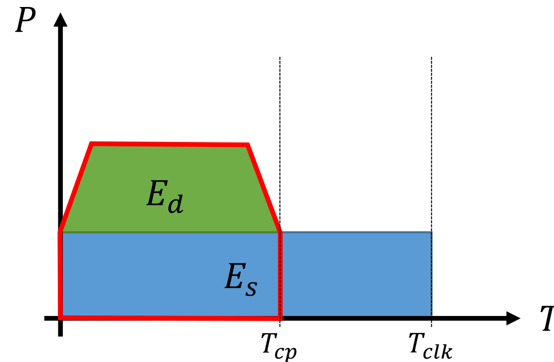


Figure 5.1 – Representation of the Power Spent by a Circuit in One Cycle. The area of the red polygon represents the total energy spent before stabilization. E_s is the static energy and E_d the dynamic energy.

Using Equation 5.2 implies that both static energy and dynamic energy are considered. Moreover, increasing the clock period on a same circuit should give the exact same energy per operation since increasing $T_{\texttt{clk}}$ will proportionally decrease the average dynamic power $P_{\texttt{d}}$ given by the tool (since integrated on a proportionally longer time), while not modifying neither the static power nor the operator critical path. Therefore, the proposed energy estimation metric gets rid of both the issues of the classical previously described metrics. Also, taking the number of cycles like in Equation 5.2 for pipelined operators into consideration provides a fair comparison between operators having a different number of cycles. Indeed, let consider two operators op_1 and op_2, where op_2 is the same circuit as op_1 but pipelined in 2 cycles (instead of 1 for op_1). Flip-flops excluded, both the circuits are the same. The energy overhead brought by flip-flops in op_2 is to some extent compensated by the smaller fan-outs in the circuit. This means that dynamic power of op_1 and op_2 are very close. However, if the pipeline is efficiently chosen, the critical path of op_2 should be twice as small as op_1. Considering this hypothesis, the energy per operation of both op_1 and op_2 should be the same. This is translated by Equation 5.2 by compensating the division of $T_{\texttt{cp}}$ with the number of cycles $N_{\texttt{c}}$.

As a conclusion, with Equation 5.2, we have a measure of the energy per operation which can be considered as robust to:

- modification of the slack due to different clock periods, and
- pipelining the operator.

With this metric, different operators, operating in slightly different conditions of frequency and pipelining can be legitimately compared. From this point, the *total energy spent before stabilization* metric is used each time energy per operation is mentioned.

For the custom floating-point comparative study, half-precision and single-precision floating-point were tested. Half-precision corresponds to an exponent represented on 5 bits and a mantissa on 11 bits. Single-precision has an 8-bit exponent and a 24-bit mantissa. Both addition/-

subtraction and multiplication were tested on each of these precisions. The results of the comparative studies for 16-bit (resp. 32-bit) addition/subtraction (resp. multiplication) are given in Tables 5.2, 5.4, 5.3 and 5.5. The two last lines of the tables refer to the relative performance of CT_FLOAT towards AC_FLOAT (resp. FLOPOCO) (e.g., CT_FLOAT area is 2.15% higher than AC_FLOAT).

	Area (μm^2)	Critical path (ns)	Total power (mW)	Energy per operation (pJ)
AC_FLOAT	312	1.44	1.84E−1	9.07E−1
CT_FLOAT	318	1.72	2.13E−1	1.05
FLOPOCO	361	2.36	1.84E−1	9.06E−1
CT_FLOAT/AC_FLOAT	**+2.15%**	**+19.4%**	**+15.4%**	**+15.7%**
CT_FLOAT/FLOPOCO	**-11.8%**	**-27.0%**	**+15.7%**	**+15.8%**

Table 5.2 – Comparative Results for 16-bit Custom Floating-Point Addition/Subtraction with $F_{\texttt{clk}} = 200$MHz

	Area (μm^2)	Critical path (ns)	Total power (mW)	Energy per operation (pJ)
AC_FLOAT	488	1.18	2.15E−1	1.05
CT_FLOAT	389	1.13	1.76E−1	8.59E−1
FLOPOCO	361	1.52	1.34E−1	6.50E−1
CT_FLOAT/AC_FLOAT	**-20.4%**	**-4.24%**	**-18.2%**	**-18.2%**
CT_FLOAT/FLOPOCO	**+7.68%**	**-25.6%**	**+31.7%**	**+32.1%**

Table 5.3 – Comparative Results for 16-bit Custom Floating-Point Multiplication with $F_{\texttt{clk}} = 200$MHz

At first sight, the three custom floating-point libraries give results in the same order of magnitude. For 16-bit addition/subtraction, CT_FLOAT is 15% more energy-costly than both AC_FLOAT and FLOPOCO, despite being as large as AC_FLOAT and 12% smaller than FLOPOCO. The fastest 16-bit adder/subtracter is AC_FLOAT, followed by CT_FLOAT, which is 19% slower but 27% faster than FLOPOCO. All performance are slightly in favor of AC_FLOATfor 16-bit addition/subtraction.

For 16-bit multiplication, AC_FLOAT is beaten by both CT_FLOATand FLOPOCO. FLOPOCO's multiplier is the smallest and with the lowest energy consumption. However, CT_FLOAT is 25% faster but consumes 32% more energy. However, it must be kept in mind that there are registers in the inputs and outputs of CT_FLOAT and AC_FLOAT which are not present for FLOPOCO, so the real gap should be narrower.

32-bit addition/subtraction shows very similar energy for AC_FLOAT, CT_FLOAT and FLOPOCO. Indeed, CT_FLOAT is 9% worse than AC_FLOAT and 4% better than FLOPOCO. Again, FLOPOCO is the slowest operator, CT_FLOAT being 27% faster.

The energy of 32-bit multiplication is strongly in favor of CT_FLOAT, which saves more

	Area (μm^2)	Critical path (ns)	Total power (mW)	Energy per operation (pJ)
AC_FLOAT	678	2.49	4.46E−1	2.21
CT_FLOAT	720	2.84	4.86E−1	2.41
FLOPOCO	772	4.10	5.05E−1	2.51
CT_FLOAT/AC_FLOAT	**+6.06%**	**+14.1%**	**+8.92%**	**+9.12%**
CT_FLOAT/FLOPOCO	**-6.85%**	**-30.8%**	**-3.69%**	**-4.15%**

Table 5.4 – Comparative Results for 32-bit Custom Floating-Point Addition/Subtraction with $F_{\texttt{clk}} = 200$MHz

	Area (μm^2)	Critical path (ns)	Total power (mW)	Energy per operation (pJ)
AC_FLOAT	1,689	2.19	1.02	5.03
CT_FLOAT	1,469	2.30	5.84E−1	2.70
FLOPOCO	2,890	3.20	1.03	5.07
CT_FLOAT/AC_FLOAT	**-13.0%**	**+5.02%**	**-42.8%**	**-46.3%**
CT_FLOAT/FLOPOCO	**-49.2%**	**-28.2%**	**-43.3%**	**-46.8%**

Table 5.5 – Comparative Results for 32-bit Custom Floating-Point Multiplication with $F_{\texttt{clk}} =$ 200MHz

than 45% more energy than both AC__FLOAT and FLOPOCO. CT__FLOAT is the 13% smaller than FLOPOCOand 49% smaller than FLOPOCO. However, AC__FLOAT is 5% faster.

In conclusion, AC__FLOAT, CT__FLOAT and FLOPOCO addition/subtraction and multiplication are quite competitive the one towards the others. Though they all have different features (implicit 1 or not, particular cases management, etc.), they all are quite close in terms of performance. Nevertheless, FLOPOCO generally produces the largest and slowest operators, but not always with the highest energy consumption. This can be explained by the fact that AC__FLOAT and CT__FLOAT operators are generated by a different software than FLOPOCOand therefore the basic integer arithmetic operators architecture used may not be the same. Also, the values of the inputs for power estimation are generated differently for AC__FLOAT and CT__FLOAT on one side, and FLOPOCO on the other side, thus activating differently the far and close paths during simulations. However, the main interesting conclusion of this study is to show that the proposed custom floating-point library CT__FLOAT competes with the other existing libraries and gives slightly comparable performance results.

In the following section, CT__FLOAT is used as a reference for the comparison with fixed-point arithmetic, first in stand-alone versions, and then leveraging classical signal processing applications.

5.2 Stand-Alone Comparison of Fixed-Point and Custom Floating-Point

Because of the different nature of floating-point and fixed-point errors, this section only compares them in terms of area, delay, and energy. Indeed, floating-point error magnitude strongly depends on the amplitude of the represented data. Low-amplitude data have low error magnitude, while high amplitude data have much higher error magnitude. Floating-point error is only homogeneous considering relative error. Oppositely, fixed-point has a very homogeneous error magnitude, uniformly distributed between well-known bounds. Therefore, its relative error depends on the amplitude of the represented data. It is low for high amplitude data and high for low amplitude data. This duality makes these two paradigms impossible to be atomically compared using the same error metric. The only interesting error comparison which can be performed is to use these two representations in the same application, which is done in Section 5.3 on FFT and K-means clustering.

The study in this section and in the rest of this chapter is performed using ApxPerf v2 as mentioned before, and with the ct_float library for custom floating-point. A 100 MHz clock is set for designing and estimating performance. All the other parameters and targets are the same as for previous section. Energy per operation is estimated using detailed power results given by PrimeTime at gate level. and is estimated using the metric described in previous Section by Equation 5.2.

In this section, 8-, 10-, 12-, 14- and 16-bit fixed-width operators are compared. For each of these bit-widths, several versions of the floating-point operators are estimated with different exponent widths. 25.10^3 uniform couples of input samples are used for each operator characterization. The random generation embedded by ApxPerf v2 insures that 25% of the floating-point adder inputs activate the close path of the operator, which has the highest energy by nature. Adders and multipliers are all tested in their fixed-width version, meaning their number of input and output bits are the same. The output is obtained using truncation of the result.

Figure 5.2 (resp. Figure 5.3) shows the area, delay and energy of adders (resp. multipliers) for different bit-widths, relative to the corresponding fixed-point operator. $\text{FlP}_N(k)$ represents N-bit floating-point with k-bit exponent width. As discussed above, floating-point adder has an important overhead compared to fixed-point adder. For any configuration, results show that area and delay are around $3\times$ higher for floating-point. As a consequence, the higher complexity of the floating-point adder leads to $5\times$ to $12\times$ more energy per operation.

Results for the multipliers are very different. Indeed, floating-point multipliers are 2-3$\times$ smaller than for fixed-point. Indeed, the control part of floating-point multiplier is much less complicated than for the adder. Moreover, as multiplication is applied only on the mantissa, the multiplication is always applied on a smaller number of bits for floating-point than for fixed-point. Timing is also slightly better for floating-point, but not as much as area since an important number of operand shifts may be needed during computations. The impact of these shifts has an important impact on the energy per operation, especially for large mantissas. This brings floating-point to suffer an overhead of $2\times$ to $10\times$ on the energy per operation.

For a good interpretation of these results, it must be kept in mind that, in a fixed-point

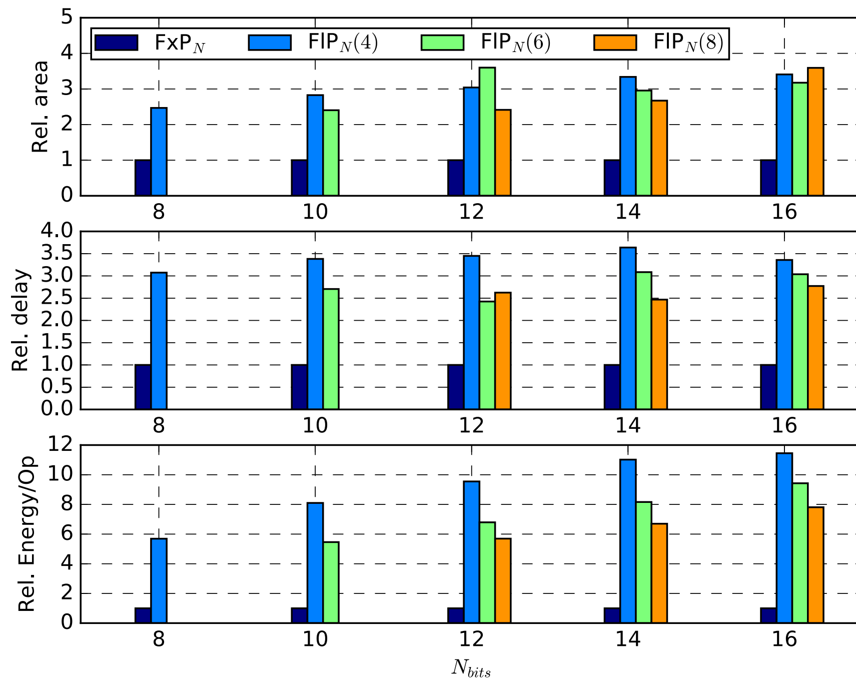


Figure 5.2 – Relative Area, Delay and Energy per Operation Comparison Between Fixed-Point and Floating-Point for Different Fixed-Width Adders

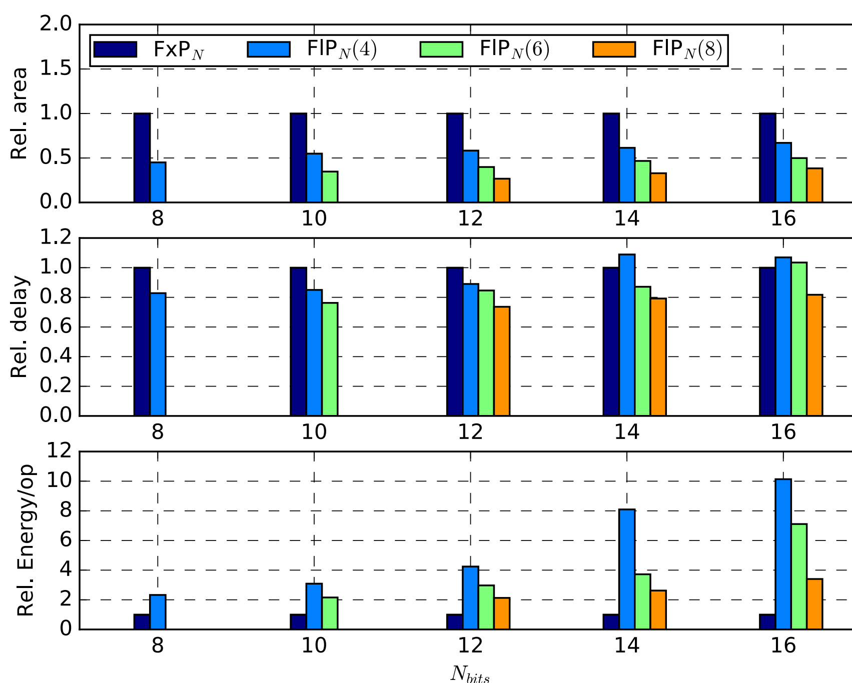


Figure 5.3 – Relative Area, Delay and Energy per Operation Comparison Between Fixed-Point and Floating-Point for Different Fixed-Width Multipliers

application, data shifting is often needed at many points in the application. The cost of shifting this data does not appear in the preliminary results presented in this section. However, for floating-point, data shifting is directly contained in the operator hardware, which is reflected in the results. Thus, the important advantage of fixed-point showed by Figures 5.2 and 5.3 must be tempered by the important impact of shifts when applied in applications.

5.3 Application-Based Comparison of Fixed-Point and Custom Floating-Point

In this section, floating-point and fixed-point operators are compared in the context of their use in applications. Indeed, as stated below, they have very different error nature and thus their error can not be legitimately compared in the previous stand-alone comparison. First, both paradigms are compared using the K-Means clustering algorithm in Section 5.3.1, these results were published in [4]. Then, FxP and FlP operators are compared on the FFT algorithm in Section 5.3.2. These results are issued from the work of Romain Mercier during an undergraduate internship at IRISA.

5.3.1 Comparison on K-Means Clustering Application

This section describes the K-means clustering algorithm and gives the comparative results for FxP and FlP. First, the principle of K-means method is described. Then, the specific algorithm used in this case study is detailed.

5.3.1.1 K-Means Clustering Principle, Algorithm and Experimental Setup

K-means clustering is a well-known method for vector quantization, which is mainly used in data mining, e.g. in image classification or voice identification. It consists in organizing a multidimensional space into a given number of clusters, each being totally defined by its centroid. A given vector in the space belongs to the cluster in which it is nearest from the centroid. The clustering is optimal when the sum of the distances of all points to the centroids of the cluster they belong to is minimal, which corresponds to finding the set of clusters $S = \{S_i\}_{i\in[0,k-1]}$ satisfying

$$\arg\min_S \sum_{i=1}^{k} \sum_{x\in S_i} \|x - \mu_i\|^2 , \qquad (5.3)$$

where μ_i is the centroid of cluster S_i. Finding the optimal centroids position of a vector set is mathematically NP-hard. However, iterative algorithms such as Lloyd's algorithm allow us to find good approximations of the optimal centroids by an estimation-maximization process, with a linear complexity (linear with the number of clusters, with the number of data to process, with the number of dimensions and with the number of iterations).

The iterative Lloyd's algorithm [74] is used in our case study. It is applied to bidimensional sets of vectors in order to have easier display and interpretation of the results. From now, we will only refer to the bidimensional version of the algorithm. Figure 5.4 shows results of K-Means on a random set of input vectors, obtained using double-precision floating-point computation with a very restrictive stopping condition. Results obtained this way are considered as the reference golden output in the rest of the paper.

The algorithm consists of three main steps:

1. Initialization of the centroids.

2. Data labelling.

3. Centroid position update.

Steps 2 and 3 are iterated until a stopping condition is met. In our case, the main stopping condition is when the difference of the sums of all distances from data points to their cluster's centroid between two iterations is less than a given threshold. A second stopping condition is the maximum number of iterations, required to avoid the algorithm getting stuck when the arithmetic approximations performed are too high to converge. The detailed algorithm for one dimension is given by Algorithm 3. Input data are represented by the vector $data$ of size N_{data}, output centroids by the vector c of size k. The accuracy target for stopping condition is defined by acc_target and the maximum allowed number of iterations by max_iter. In our study, we use several values for acc_target, and max_iter is set to 150, which is never reached in

Algorithm 3 K-Means Clustering (1 Dimension)

Require: $k \leq N_{data}$
  $err \leftarrow +\infty$
  $cpt \leftarrow 0$
  $c \leftarrow \text{init_centroids}(data)$
  do ▷ Main loop
    $old_err \leftarrow err$
    $err \leftarrow 0$
    $c_tmp[0 : k-1] \leftarrow 0$
    $min_distance \leftarrow +\infty$
    for $d \in \{0 : N_{data} - 1\}$ **do**
      $min_distance \leftarrow +\infty$
      for $i \in \{0 : k-1\}$ **do** ▷ Data labelling
        $distance \leftarrow \text{distance_comp}(data[d], c[i])$
        if $distance < min_distance$ **then**
          $min_distance \leftarrow distance$
          $labels[d] \leftarrow i$
        end if
      end for
      $c_tmp[labels[d]] \leftarrow c_tmp[labels[d]] + data[d]$
      $counts[labels[d]] \leftarrow counts[labels[d]] + 1$
      $err \leftarrow err + min_distance$
    end for
    for $i \in \{0 : k-1\}$ **do** ▷ Centroids position update
      if $counts[i] \neq 0$ **then**
        $c[i] \leftarrow c_tmp[i]/counts[i]$
      else
        $c[i] \leftarrow c_tmp[i]$
      end if
    end for
    $cpt \leftarrow cpt + 1$
  while $(|err - old_err| > acc_target) \vee (cpt < max_iter)$

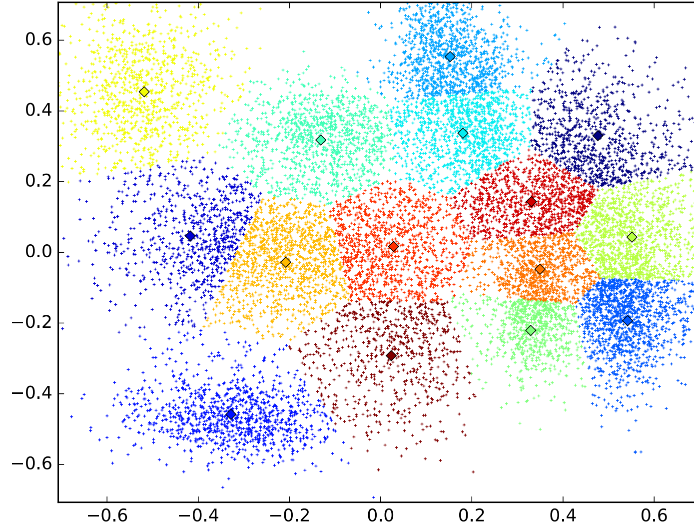


Figure 5.4 – 2-D K-means Clustering Golden Output Example, Obtained Using Floating-Point Double-Precision

practice.

The impact of fixed-point and floating-point arithmetic on performance and accuracy is evaluated considering the distance computation function $\mathrm{distance_comp}$, defined by:

$$d \leftarrow (x - y) \times (x - y). \tag{5.4}$$

The computation is written this way instead of using the square function in order to let the HLS determine the intermediate types, thanks to C++ native types overloading implemented in CT_FLOAT and AC_FIXED, which are used for floating-point and fixed-point implementation, respectively. All the other parts of the computations are implemented using double-precision floating-point, and their contribution to the performance cost is not evaluated. Using a whole approximate K-means application would require these operations to be approximated the same way as distance computation. However, as distance computation is the most complex part of the algorithm and as it is the deepest operation in the inner loops, its impact on accuracy and performance is the most critical.

In the 2D case, the distance computation becomes

$$d \leftarrow (x_0 - y_0) \times (x_0 - y_0) + (x_1 - y_1) \times (x_1 - y_1), \tag{5.5}$$

which is equivalent to 1 addition, 2 subtractions, and 2 multiplications. However, as distance computation is cumulative on each dimension, the hardware implementation relies only on 1 addition (accumulation), 1 subtraction, and 1 multiplication.

The experimental setup is divided into two parts: accuracy and performance estimation. Accuracy estimation is performed on 20 data sets composed of 15.10^3 bidimensional data samples. These data samples are all generated in a square delimited by the four points $\{\pm\sqrt{2}, \pm\sqrt{2}\}$, using Gaussian distributions with random covariance matrices around 15 random mean points. Several accuracy targets are used to set the stopping condition: 10^{-2}, 10^{-3}, 10^{-4}. The reference for accuracy estimation is IEEE-754 double-precision floating-point. Figure 5.4 is an example of a typical golden output for the experiment. The error metrics for the accuracy estimation are:

- the Mean Square Error of the resulting cluster Centroids (CMSE), and
- the classification Error Rate (ER) in percents, which is defined as the proportion of points not being tagged by the right cluster identifier.

The lower the CMSE, the better the estimated position of centroids compared to golden output. Energy estimation is performed using the first of these 20 data sets, limited to 20.10^3 iterations of distance computation for time and memory purposes. As data sets were generated around 15 points, the number of clusters researched is also set to 15. Performance and accuracy of the K-Means clustering experiment, from input data generation to result processing and graphs generation, is fully available in the open-source APXPERF v2 framework, which is used for the whole study.

5.3.1.2 Experimental Results on K-Means Clustering

Section 5.2 showed that fixed-point additions and multiplications consume less energy than floating-point for the same bitwidth. However, these results do not yet consider the impact of the arithmetic on accuracy. This section details the impact of accuracy on the bidimensional K-means clustering algorithm.

A first qualitative study on the K-Means clustering showed that, to get correct results (no artifacts), floating-point data must have a minimal exponent width of 5 bits in distance computation (smaller exponents are too inaccurate in low distance computations) and fixed-point data a minimal number of 3 bits for its integer part. Thus, all the following results use these two parameters. Area, latency and energy of distance computed by Equation 5.5 are provided. The total energy of the application is defined as

$$E_{\text{K-means}} = E_{\text{dc}} \times (N_{\text{it}} + N_{\text{cycles}} - 1) \times N_{\text{data}}, \tag{5.6}$$

where E_{dc} is the energy per distance computation calculated from the data extracted with APXPERF, N_{it} the average number of iterations necessary to reach K-means stopping condition, N_{cycles} the number of stages in the pipeline of the distance computation core (automatically determined by HLS), and N_{data} the number of processed data per iteration.

Results for 8-bit and 16-bit FlP and FxP arithmetic operators are detailed in Table 5.6, with stopping condition set to 10^{-4}. For the 8-bit version of the algorithm, several interesting results can be highlighted. First, the custom floating-point version is twice as large as fixed-point version and floating-point distance computation consumes $2.44\times$ more energy than fixed-point. However, the floating-point version of K-means converges in 8.35 cycles on average against 14.9 cycles for fixed-point. This makes floating-point version for the whole K-means

	$\text{ct_float}_8(5)$	$\text{ct_float}_{16}(5)$	$\text{ac_fixed}_8(3)$	$\text{ac_fixed}_{16}(3)$
Area (μm^2)	392.3	1148	180.7	575.1
N_{cycles}	3	3	2	2
E_{dc} (nJ)	1.23E−4	5.99E−4	5.03E−5	3.25E−4
N_{it}	8.35	59.3	14.9	65.1
$E_{\text{K-means}}$ (nJ)	38.24	1100	23.90	644.34
CMSE	1.75E−3	3.03E−7	1.85E−2	3.28E−7
Error Rate	35.1 %	2.94 %	62.3 %	0.643 %

Table 5.6 – 8- and 16-bit Performance and Accuracy for K-Means Clustering Experiment

algorithm consuming only $1.6\times$ more energy than fixed-point. Moreover, floating-point version has a huge advantage in terms of accuracy. Indeed, CMSE is $10\times$ better for floating-point and ER is $1.8\times$ better. Figures 5.5a and 5.5b show the output for floating-point and fixed-point 8-bit computations, applied on the same inputs than the golden output of Figure 5.4. A very neat stair-effect on data labelling is clearly visible, which is due to the high quantization levels of the 8-bit representation. However, in the floating-point version, the positions of clusters centroid is very similar to the reference, which is not the case for fixed-point.

For the 16-bit version, all results are in favor of fixed-point, floating-point being twice bigger and consuming $1.7\times$ more energy. Fixed-point also provides slightly better error results (2.9% for ER vs. 0.6%). Figures 5.5c and 5.5d show output results for 16-bit floating-point and fixed-point. Both are very similar and nearly equivalent to the reference, which reflects the high success rate of clustering.

The competitiveness of FlP over FxP on small bit-widths and the higher efficiency of FxP on larger bit-widths is confirmed by Figure 5.6 depicting energy vs. classification error rate. Indeed, for different accuracy targets ($10^{-\{2,3,4\}}$), only 8-bit floating-point provides higher accuracy for a comparable energy cost, while 10- to 16-bit fixed-point versions reach an accuracy equivalent to floating-point with much lower energy. The stopping condition does not seem to have a major impact on the relative performance.

5.3.2 Comparative Results on Fast Fourier Transform Application

In the previous section, a comparative study between fixed-point and custom floating-point was performed on K-means. We showed that, contrary to what could be expected, floating-point was very competitive for small bit-width, besides being easier to manage due to its high flexibility. In this section, a similar study is performed on the Fast Fourier Transform (FFT). The error study, generation and analysis of results were performed by undergraduate intern Romain Mercier. The hardware performance estimation part was obtained using ApxPerf v2. The original study also included approximate integers operators, which will not be discussed here since a study on approximate operators in FFT has already been done in Section 4.3.1.

The implementation of the studied FFT is Radix-2 Decimation-In-Time (DIT) FFT, which is the most common form of the Cooley-Tukey algorithm [75]. For the hardware estimation,

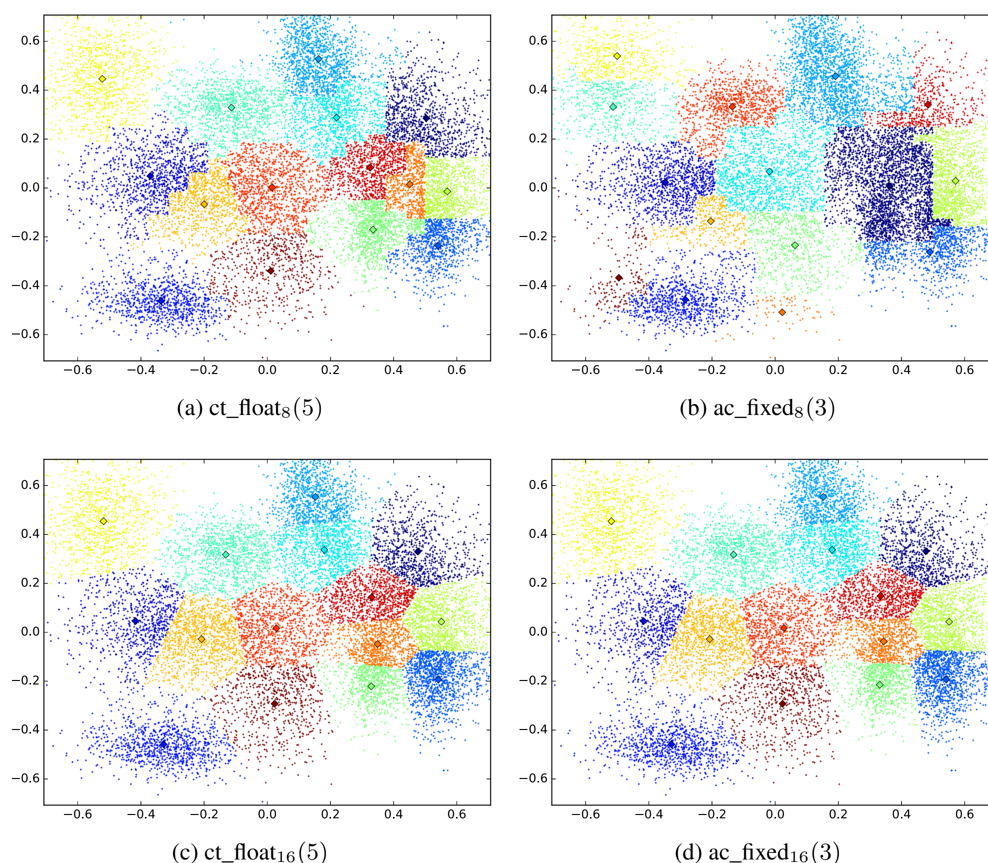


Figure 5.5 – K-Means Clustering Outputs for 8- and 16-bit floating-point and fixed-point with Accuracy Target of 10^{-4}

only the core of computation of the FFT is considered, i.e. the computation of:

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k$$
$$X_{k+\frac{N}{2}} = E_k - e^{-\frac{2\pi i}{N}k} O_k. \tag{5.7}$$

This leads to the hardware implementation of 6 additions/subtractions and 4 multiplications. For each version of the FFT, all constants and variables are represented with the same parameters (same bit-width, same integer part width for FxP, same exponent width for FlP). The absence of over/underflow for FxP version is ensured. For FlP version, the repartition of the exponent and mantissa widths is chosen for giving the smallest error after exhaustive search. For hardware performance estimation, only FFT-16 was characterized. The error metric used for the study of error is the Mean Square Error (MSE) at the output compared to double-precision floating-point FFT.

Energy per operation related to error for FFT-16 is depicted in Figure 5.7. On the x-axis, the energy derived from Equation 5.2 of the computing core of FFT is given in pJ. On the y-

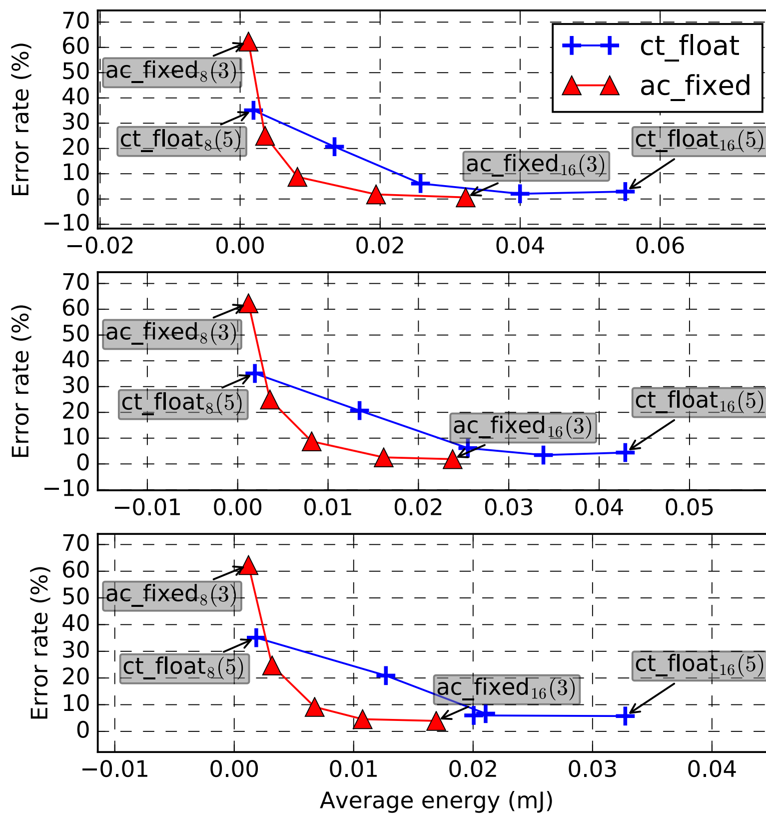


Figure 5.6 – Energy Versus Classification Error Rate for K-Means Clustering with Stopping Conditions of 10^{-4} (Top), 10^{-3} (Center) and 10^{-2} (Bottom)

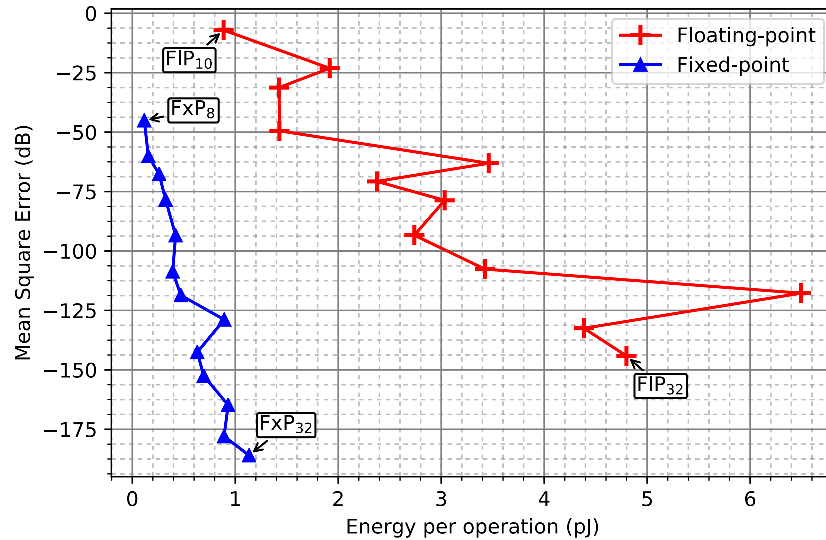


Figure 5.7 – Fixed-Point and Floating-Point Energy per Operation vs MSE for FFT-16 for Different Bit-Widths

axis, the MSE is represented in dB. The best error-energy trade-off is best when approaching the bottom-left corner. For each curve, each point going from the top left to the bottom right represents an increase of two in the bit-width.

In this experiment, the advantage is clearly in favor of fixed-point. Indeed, for any identical bit-width, fixed-point outperforms floating-point in terms of energy and accuracy. As already showed in Section 5.2, floating-point operations, additions in particular, are much more expensive than fixed-point in return for an increased accuracy on larger dynamic. However, FFT output quality is not as dependent on accuracy on a dynamic as large as for K-means clustering for instance. This makes floating-point even less accurate than fixed-point at equal bit-width, because of a smaller significant part, mantissa for floating-point, all bits for fixed-point. Indeed, in the experiment, the exponent takes 7 bits of the total width, which are not assigned to more accuracy on the significant part. Another interesting point is the data points presenting an *energy peak*, which are occurring for 12-, 18- and 28-bit floating-point and 22-bit fixed-point. These peaks are most probably due to differences of implementation in the HLS process. E.g, larger adder or multiplier structures may have been selected by the tool to meet constraint of delays, leading to energy overhead.

5.4 Conclusion and Discussion about the use of Fixed-Point and Floating-Point arithmetic for Energy-Efficient Computing

A raw comparison of floating-point and fixed-point arithmetic operators gives an advantage in area, delay and energy efficiency for fixed-point. However, the comparison on a real application like the K-means clustering algorithm provides interesting features to custom floating-point

arithmetic. Indeed, for K-means, contrary to what would have been expected, floating-point arithmetic tends to show better results in terms of energy/accuracy trade-off for very small bit-widths (8 bits in our case). However, increasing this bit-width still leads to an important area, delay and energy overhead of floating-point. The most interesting results occur for 8-bit floating-point representation. With only 3 bits of mantissa, which corresponds to only 3-bit integer adders and multipliers, the results are better than 8-bit fixed-point integer operators. This is obviously due to the adaptive dynamic offered by floating-point arithmetic at operation level, whereas fixed-point has a fixed dynamic which is disadvantageous for low-amplitude data and distance calculation. However, non-iterative algorithms should be tested to know if small floating-point keeps its advantage.

Floating-point representation showed its limitations on the FFT experiment. Indeed, the significant bits of the mantissa sacrificed to the exponent represent a penalty in an application whose output quality is not as dependent on a high accuracy on enhanced dynamic as the K-means clustering is. However, the gap between fixed-point and floating-point in this context could probably be narrowed with different architecture choices, such as exponent bias and subnormal numbers support. Nevertheless, these features would come with inevitable area, delay and energy overheads.

From a hardware-design point of view, custom floating-point is costly compared to fixed-point arithmetic. Fixed-point benefits from *free* data shifting between two operators, as outputs of one operator only need to be connected to the inputs of the following in the datapath. However, from a software-design point of view, shifts between fixed-point computing units must be effectively performed, which leads to a non-negligible delay and energy overhead. Oppositely, floating-point computing units do not suffer from this overhead, since data shifting is implemented in the operators and managed by the hardware at runtime. Thanks to this feature, floating-point exhibits another important advantage which is the ease of use, since software development is faster and more secured.

Hence, in the aim of producing general-purpose low-energy processors, small-bitwidth floating-point arithmetic can provide major advantages compared to classical integer operators embedded in microcontrollers, with a better compromise between ease of programming, energy efficiency and computing accuracy.

Conclusion

To face the predicted end of Moore's Law, this thesis proposes to look into approximate architectures. Indeed, it has been showed that most applications can be computed with relaxed accuracy without affecting their output quality, or with a tolerable degradation. Several levels of architectural approximations are possible, listed in Chapter 1. In this thesis, the opportunities to save energy using approximate arithmetic are highlighted. They concern floating-point, fixed-point, and approximate integer adders and multipliers. In this document, four main contributions were proposed. First, to our knowledge, there was no existing critical and comparative study of existing approximate arithmetic operators considering an equivalent number of references, so the first Chapter of this document can constitute a base.

Two techniques for the estimation of the output error of approximate systems were proposed, making the second main contribution. Concerning fixed-point systems, a fast and scalable method leveraging the spectral shape of error was described in Chapter 2. This technique has the same accuracy as statistical propagation techniques, but with much lower complexity in the analytical model construction. The development and results of this new approach were presented in [1]. In the first part of Chapter 3, a technique based on bitwise-error rate propagation was proposed for approximate operators. Compared to others, this technique is a good compromise between memory cost and accuracy. However, the very different natures of existing approximate operators make it very hard to find generally good models. After exploring a high number of solutions, the conclusion is that no model approaches the accuracy of Monte Carlo without equaling or exceeding its complexity. Contrary to fixed-point paradigm, it seems that approximate arithmetic operators can not be safely be used without preliminary simulations.

A third contribution was to use error behavior of approximate operators to estimate the effect of VOS on accurate operators. Indeed, simulating VOS requires transistor-level simulation, which is extremely long and memory-costly. Using combinations of approximate operators trained on real data allows for fast estimation of the effects of VOS in systems too large to be simulated. This contribution was published in [3].

Finally, two comparative studies of existing approximate paradigms, supported by the creation and usage of our open-source framework APXPERF, constitute the fourth contribution. First, fixed-point and approximate operators are compared. In their stand-alone versions, both are quite competitive. As most approximate operators are based on shortening the carry chains, they are generally fast and they generate scarce but high amplitude error, depending on their parameters. In opposition, fixed-point always generate errors because of quantization at the output, but this error is always small and well characterized. Therefore, they have quite similar error in average, whereas approximate operators tend to be faster thanks to shorter critical

path. However, our study showed that when comparing both paradigms in real signal processing applications, fixed-point is much better for two reasons. Firstly, its error entropy is minimal, only standing in the dropped bits which are the LSBs. Therefore, the propagation of this error across the system leads to a contained amplification. On the contrary, high-entropy approximate operator error potentially occurring at any bit significance leads to drastically important amplification effects, and the application output may be strongly degraded. Secondly, dropping bits at the output of fixed-point operators instead of keeping the same bit-width during computations for approximate operators has an effect which had never been pointed out until now, which is the need for smaller downstream operators and for less memory. These two reasons make quantization by far superior to operator-level approximation. The only reason why approximate operators could be considered is for constant bit-width processing like in CPUs and when computation must necessarily be faster than what fixed-point can offer. This study was published in [2].

The second study puts face-to-face fixed-point arithmetic and floating-point arithmetic in the context of low-energy computing. Generally, floating-point is associated to high accuracy computing, but with important hardware cost. In our study, we consider small-width floating-point across our custom library CT_FLOAT. After a comparison between different existing custom floating-point libraries showing that CT_FLOAT is competitive, we use this library combined with APXPERF to evaluate its cost facing fixed-point. In their stand-alone versions, floating-point addition/subtraction is as expected much more costly than fixed-point. For multiplication, the gap is tight as floating-point uses smaller integer multiplier and as the control overhead in floating-point multiplier is reasonably small. Two applications were used to evaluate the real overhead of floating-point. In K-means clustering, small floating-point in surprisingly competitive with fixed-point. Indeed, K-means requires accurate computations both for small and large distance, which are advantaging floating-point thanks to its high accuracy on larger dynamic than fixed-point. In the context of FFT, whose quality is less impacted by inaccuracy on small amplitude signals, fixed-point strongly outperforms floating-point. Indeed, the large dynamic of floating-point does not bring enough improvement in this case to compensate for its overhead. As a conclusion, small-width floating-point competitiveness in terms of energy-error ratio is strongly dependent on the application. Nevertheless, despite a generally higher energy consumption for floating-point compared to fixed-point, its ease of use makes it very interesting for fast development. Having a CPU embedding small bit-width floating-point processing units could be very interesting for general-purpose low-power computing. The study on K-means clustering application was published in [4].

The work constituting this thesis comes with the following conclusions:

- Approximate operators error degradation across a system is difficult to estimate efficiently when excluding Monte Carlo simulation. However, approximate operators can be used to reproduce the effects of physical phenomena such as VOS.

- If stand-alone approximate arithmetic operators are generally competitive, they show important limitations when used in real-life signal processing applications compared to classical fixed-point.

- Floating-point energy cost is able to compete with fixed-point when used in applications that require accuracy at different amplitude scales. In this context, using floating-point for low-energy computing has to be considered.

This thesis also opens-up new perspectives. Firstly, it has been showed that in general, using approximate operators leads to important errors. Therefore, there is a strong need to find new approximate operators with better error performance. Some of them are on a good track. With DRUM approximate multiplier for instance, no high amplitude errors can be performed, as it is based on floating-point paradigm applied to a fixed representation. The idea of mixing fixed-point and floating-point paradigms is a good idea for speed, though it imposes the storage of many useless zeros in the LSBss. Other operators like the GDA approximate adder propose runtime-configurable approximate adders, which shows interest in the context of energy-autonomous embedded devices that may need to run in different levels of power consumption depending on the remaining stored energy and the amount of energy being harvested. Finally, the concept of exact adders based on error-corrected approximate operators proposed by VLSA deserves to be further investigated.

Secondly, models for approximate operators error propagation need to be developed, though we concluded in this study that their various natures are an impediment to the good performance of general models. In our mind, new models should consider the nature of the approximation performed on each operator. Therefore, the general model would be an aggregation of several models, and the one that adapts best to a considered approximate operators would be selected when needed.

Finally, floating-point paradigm for low-energy computing has a high potential for new research. First, the best compromise between control overhead and accuracy should be investigated in this context, for instance the management of subnormals. The question if the accuracy benefits from supporting these low amplitude numbers is important enough to compensate for the hardware overhead. The customization of the exponent bias also has to be investigated. With a constant bias, adding one more bit in the exponent increases the dynamic as much towards infinity than towards zero. However, it is not interesting to have a dynamic which is too large towards infinity, since resources could be allocated to even better accuracy around zero. Nevertheless, introducing a fully parameterizable bias would bring an important memory and control overhead. A solution might be to have several possible predefined biases that could be used to operate at different amplitude scales with moderated overhead. The interest of handling particular cases such as under/overflows also needs to be investigated. Once all these investigations, the structure of a low-energy small-bit-width processing unit should be proposed with innovative features. For instance, if exponent management is faster, it could be possible to use a single exponent management unit for two mantissa management units to save area.

All these opportunities should be considered for future research and the proposition of new energy-efficient architectures, which are a major stake to overcome the predicted end of Moore's Law.

Acronyms

AAM Approximate Array Multiplier. 68, 70, 72–74, 77, 107, 108, 114, 117–119, 140, 143, 144, 148, 149, 194, 197

AAMI Approximate Array Multiplier I. 68–70, 72, 192, 197

AAMII Approximate Array Multiplier II. 68–70, 72, 192, 197

AAMIII Approximate Array Multiplier III. 68, 70–74, 77, 114, 135, 140, 193

AC2A Accuracy-Configurable Adder. 50–55, 58, 60, 61, 107, 192, 197

ACA Almost Correct Adder. 38–43, 46, 56, 107, 108, 111, 114, 116–120, 128, 135, 140, 143, 146, 192, 194

ACGPI Approximate Carry Generation Procedure version I. 80–83, 197

ACGPII Approximate Carry Generation Procedure version II. 81–83, 193, 197

AMA Approximate Mirror Adder. 62, 64

AMA1 Approximate Mirror Adder type 1. 62–65, 192

AMA2 Approximate Mirror Adder type 2. 62–65, 192

AMA3 Approximate Mirror Adder type 3. 62, 64, 65

AP Acceptance Probability. 45–49, 69, 70, 76, 134, 197

Apx Approximate. 109, 133, 140

ASIC Application Specific Integrated Circuit. 156

BER Bit Error Rate. 52, 94, 107, 109, 122, 129, 130, 134, 142, 143, 194, 195

BKA Brent-Kung Adder. 30, 31, 128, 129

BWER Bitwise-Error Rate. 6, 109–117, 119–122, 130, 134, 194, 198

CLA Carry-Lookahead Adder. 29, 30, 46, 51, 52, 58, 197

CMSE Mean Square Error of the resulting cluster Centroids. 167, 168

CORDIC COordinate Rotation DIgital Computing. 11, 14

CRN Convergent Rounding to Nearest. 24–26

CSA Carry-Save Adder. 34, 66, 67

CSGCI Control Signal Generating Cells of type I. 43, 45, 74

CSGCII Control Signal Generating Cells of type II. 45, 46

CSLA Carry-Select Adder. 29

DCT Discrete Cosine Transform. 62, 65, 144–147, 195

DIT Decimation-In-Time. 168

DRUM Dynamic Range Unbiased Multiplier. 87–90, 107, 108, 114, 117–119, 175, 193, 194, 198

DSP Digital Signal Processing blocks. 10, 121, 156

DWT Discrete Wavelet Transform. 101, 102, 104–106, 193, 194

EDA Energy-Delay-Area. 66

EEV Error Event Vector. 110–112

ER Error Rate. 167, 168

ETA Error-Tolerant Adder. 43, 107

ETAI Error-Tolerant Adder type I. 43–47, 50, 52, 55, 61, 74, 107, 192

ETAII Error-Tolerant Adder type II. 43, 46–48, 50, 192, 197

ETAIIM Error-Tolerant Adder type II Modified. 43, 47, 48, 50, 52, 192

ETAIV Error-Tolerant Adder type IV. 43, 47–50, 55, 114, 135, 140, 146, 192, 197

ETM Error-Tolerant Multiplier. 74, 76, 77, 193

FA Full Adder. 27, 29, 33, 34, 61, 62, 70, 81, 140

FBM Fixed-width modified-Booth-encoded Multiplier. 107, 140, 143, 144, 148, 149

FBMI Fixed-width modified-Booth-encoded Multiplier version I. 78, 80, 81, 83, 85, 86, 197

FBMII Fixed-width modified-Booth-encoded Multiplier version II. 78, 80, 81, 83, 86, 87, 197

FBMIII Fixed-width modified-Booth-encoded Multiplier version III. 83–87, 140, 193

FDSOI Fully Depleted Silicon On Insulator. 14

FFT Fast Fourier Transform. 135, 137, 144, 145, 161, 163, 168, 169, 171, 172, 174

FIR Finite Impulse Response. 101, 103, 153

FlP Floating-Point. 15–21, 27, 87, 89, 133, 163, 164, 167–169, 197

FPGA Field-Programmable Gate Array. 10, 14, 121, 155, 156

FWM Fixed-Width Multiplier. 68

FxP Fixed-Point. 20–24, 27, 31, 36, 40, 41, 62, 77, 93–95, 107–109, 130, 131, 133, 135, 137, 140, 143, 144, 148, 163, 164, 167–169, 191, 194

GDA Gracefully-Degrading Adder. 55–61, 107, 175, 192, 197

HA Half Adder. 33, 34, 81

HCA Han-Carlson Adder. 38, 66

HEVC High Efficiency Video Codec. 135, 144, 146, 148, 198

HLS High Level Synthesis. 12, 17, 21, 121, 134–136, 140, 151, 152, 156, 166, 167, 171

HPC High Performance Computing. 11

IDCT Inverse Discrete Cosine Transform. 64, 65

IIR Infinite Impulse Response. 101, 103

IMPACT IMPrecise Adder for low-power Approximate CompuTing. 62, 64, 65, 114, 116, 118–120, 135, 140, 143, 146, 192, 194

IOT Internet Of Things. 10

KSA Kogge-Stone Adder. 31, 32, 38, 50, 66

LFA Ladner-Fischer Adder. 50, 66

LPA Lu's Parallel Adder. 38–41, 46, 52, 58, 107, 192

LSB Least Significant Bit. 16, 19, 20, 24, 29, 37, 43, 47, 48, 54, 56, 62, 64, 65, 68, 74, 76, 77, 87, 110–112, 116, 140, 146, 174, 175

LTI Linear and Time-Invariant. 6, 94, 99, 103

LUT Look-Up Table. 10

MA Mirror Adder. 61–63, 192, 197

MAA Minimum Acceptable Accuracy. 45–49, 69, 76, 134, 197

MAE Mean Absolute Error. 134

MC Motion Compensation. 146, 148

MSB Most Significant Bit. 16, 20–22, 24, 29, 30, 34, 35, 37, 43, 47, 48, 50, 62, 68, 69, 74, 76, 87, 89, 110, 116, 120, 130, 140, 146

MSE Mean Square Error. 95, 98, 101, 128, 129, 134, 140, 141, 143, 144, 169, 171, 194, 195

MSSIM Mean Structural Similarity. 145, 146, 195

MXOR Modified XOR. 45

PDF Probability Density Function. 109, 134

PDP Power-Delay Product. 46, 76, 77, 140, 143, 157, 193

PQN Pseudo-Quantization Noise. 95

PSD Power Spectral Density. 6, 94, 97–101, 103–106, 133, 134, 193, 198

PSNR Peak Signal-to-Noise Ratio. 62, 64, 86, 144, 145, 195

RCA Ripple Carry Adder. 27, 29, 30, 34, 46, 58, 62, 64, 66, 128–130, 194

RD Rounding towards $-\infty$. 22, 24–26

RDF Random Dopant Fluctuations. 13, 14

RN Rounding to Nearest. 22, 24–26

RTL Register Transfer Level. 136

RU Rounding towards $+\infty$. 22, 24–26

RZ Rounding towards 0. 22, 23, 25

SFG Signal Flow Graph. 94, 99, 100, 193

SIMD Single Instruction Multiple Data. 14

SNR Signal-to-Noise Ratio. 78, 86, 128, 129

SoC System on Chip. 68

SQNR Signal-to-Quantization-Noise Ratio. 94

VLCSA-1 Variable Latency Carry Select Adder. 58

VLSA Variable Latency Speculative Adder. 41–43, 52, 175, 192

VLSI Very Large Scale Integration. 13, 34

VOS Voltage OverScaling. 6, 12–14, 52, 53, 64, 107, 121–131, 173, 174, 192, 194

WPKSA Weighted-Pruned Kogge-Stone Adder. 67

Publications

[1] **B. Barrois**, K. Parashar, and O. Sentieys, “Leveraging power spectral density for scalable system-level accuracy evaluation,” in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 750–755, March 2016.

[2] **B. Barrois**, O. Sentieys, and D. Menard, “The hidden cost of functional approximation against careful data sizing - a case study,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 181–186, March 2017.

[3] R. Ragavan, **B. Barrois**, C. Killian, and O. Sentieys, “Pushing the limits of voltage overscaling for error-resilient applications,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 476–481, March 2017.

[4] **B. Barrois** and O. Sentieys, “Customizing fixed-point and floating-point arithmetic - a case study in k-means clustering,” in *IEEE International Workshop on Signal Processing (SiPS), 2017*, October 2017.

Bibliography

[5] "Sunway TaihuLight supercomputer specifications." http://www.nsccwx.cn/wxcyw/soft1.php?word=soft&i=46. Accessed: 2017-06-26.

[6] G. E. Moore, "Cramming more components onto integrated circuits, electronics,(38) 8," 1965.

[7] R. S. Williams, "What's next? [the end of moore's law]," *Computing in Science Engineering*, vol. 19, pp. 7–13, Mar 2017.

[8] L. D. Xu, W. He, and S. Li, "Internet of things in industries: A survey," *IEEE Transactions on Industrial Informatics*, vol. 10, pp. 2233–2243, Nov 2014.

[9] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Context aware computing for the internet of things: A survey," *IEEE Communications Surveys Tutorials*, vol. 16, pp. 414–454, First 2014.

[10] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, pp. 414–425, June 1995.

[12] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama, "Variable supply-voltage scheme for low-power high-speed cmos digital design," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 454–462, Mar 1998.

[13] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," in *Proceedings of the 2010 international conference on Power Aware Computing and Systems*, pp. 1–8, 2010.

[14] J. E. Volder, "The cordic trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, pp. 330–334, Sept 1959.

[15] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[16] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, pp. 10–20, Nov 2004.

[17] P. Roy, R. Ray, C. Wang, and W. F. Wong, "Asac: Automatic sensitivity analysis for approximate computing," *SIGPLAN Not.*, vol. 49, pp. 95–104, June 2014.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," *SIGPLAN Not.*, vol. 47, pp. 301–312, Mar. 2012.

[19] B. Grigorian and G. Reinman, "Dynamically adaptive and reliable approximate computing using light-weight error analysis," in *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 248–255, July 2014.

[20] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 124–134, ACM, 2011.

[21] J. G. Pandey, A. Karmakar, C. Shekhar, and S. Gurunarayanan, "An fpga-based fixed-point architecture for binary logarithmic computation," in *2013 IEEE Second International Conference on Image Information Processing (ICIIP-2013)*, pp. 383–388, Dec 2013.

[22] A. Cristiano, I. Malossi, Y. Ineichen, C. Bekas, and A. Curioni, "Fast exponential computation on simd architectures," in *HiPEAC 2015 - 1st Workshop On Approximate Computing (WAPCO)*, 2015.

[23] J.-M. Muller, N. Brisebarre, F. De Dinechin, C.-P. Jeannerod, V. Lefevre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of floating-point arithmetic.* Springer Science & Business Media, 2009.

[24] B. Widrow, "A study of rough amplitude quantization by means of nyquist sampling theory," *IRE Transactions on Circuit Theory*, vol. 3, pp. 266–276, Dec 1956.

[25] B. Widrow, "Statistical analysis of amplitude-quantized sampled-data systems," *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, vol. 79, pp. 555–568, Jan 1961.

[26] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals: Architectures and Features.* Wiley-IEEE Press, 1997.

[27] I. Koren, *Computer Arithmetic Algorithms.* A. K. Peters, Ltd., 2nd ed., 2001.

[28] B. Parhami, *Computer Arithmetic.* Oxford University Press, 2nd ed., 2010.

[29] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, pp. 786–793, Aug 1973.

[30] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Transactions on Electronic Computers*, vol. EC-13, pp. 14–17, Feb 1964.

[31] L. Dadda, "Some schemes for parallel multipliers," *Alta frequenza*, vol. 34, no. 5, pp. 349–356, 1965.

[32] M. F. Schilling, "The longest run of heads," *The College Mathematics Journal*, pp. 196–207, 1990.

[33] S.-L. Lu, "Speeding up processing with approximation circuits," *IEEE Computer*, vol. 37, pp. 67–73, Mar 2004.

[34] A. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *Proceedings of Design, Automation and Test in Europe (DATE'08)*, pp. 1250–1255, March 2008.

[35] K. Y. Kyaw, W. L. Goh, and K. S. Yeo, "Low-power high-speed multiplier for error-tolerant application," in *IEEE International Conference of Electron Devices and Solid-State Circuits (EDSSC'10)*, pp. 1–4, Dec 2010.

[36] N. Zhu, W. L. Goh, and K. S. Yeo, "An enhanced low-power high-speed adder for error-tolerant application," in *Proceedings of the 12th International Symposium on Integrated Circuits (ISIC'09)*, pp. 69–72, Dec 2009.

[37] N. Zhu, W. L. Goh, G. Wang, and K. S. Yeo, "Enhanced low-power high-speed adder for error-tolerant application," in *International SoC Design Conference (ISOCC'10)*, pp. 323–327, Nov 2010.

[38] R. Ladner and M. Fischer, "Parallel prefix computation," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[39] A. B. Kahng and S. Kang, "Accuracy-configurable adder for approximate arithmetic designs," in *Proceedings of 49th Design Automation Conference (DAC'12)*, pp. 820–825, June 2012.

[40] N. Zhu, W. L. Goh, W. Zhang, K. S. Yeo, and Z. H. Kong, "Design of low-power high-speed truncation-error-tolerant adder and its application in digital signal processing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, pp. 1–5, Aug 2009.

[41] "Standard performance evaluation corporation (spec) cpu2006." http://www.spec.org/cpu2006.

[42] R. Ye, T. Wang, F. Yuan, R. Kumar, and Q. Xu, "On reconfiguration-oriented approximate adder design and its application," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*, pp. 48–54, Nov 2013.

[43] K. Du, P. Varman, and K. Mohanram, "High performance reliable variable latency carry select addition," in *Design, Automation Test in Europe Conference Exhibition (DATE'12)*, pp. 1257–1262, March 2012.

[44] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," in *International Symposium on Low Power Electronics and Design (ISLPED'11)*, pp. 409–414, Aug 2011.

[45] A. Lingamneni, C. Enz, J.-L. Nagel, K. Palem, and C. Piguet, "Energy parsimonious circuit design through probabilistic pruning," in *Proceedings of IEEE/ACM Design, Automation Test in Europe Conference (DATE'11)*, pp. 1–6, March 2011.

[46] S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, vol. 43, pp. 90–95, Feb 1996.

[47] J. M. Jou and S. R. Kuang, "Design of low-error fixed-width multiplier for dsp applications," *Electronics Letters*, vol. 33, pp. 1597–1598, Sep 1997.

[48] L.-D. Van, S.-S. Wang, and W.-S. Feng, "Design of the lower error fixed-width multiplier and its application," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, pp. 1112–1118, Oct 2000.

[49] S.-J. Jou and H.-H. Wang, "Fixed-width multiplier for dsp application," in *Proceedings of International Conference on Computer Design*, pp. 318–322, 2000.

[50] K.-J. Cho, K.-C. Lee, J.-G. Chung, and K. Parhi, "Design of low-error fixed-width modified booth multiplier," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, pp. 522–531, May 2004.

[51] T.-B. Juang and S.-F. Hsiao, "Low-error carry-free fixed-width multipliers with low-cost compensation circuits," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, pp. 299–303, June 2005.

[52] S. Hashemi, R. I. Bahar, and S. Reda, "Drum: A dynamic range unbiased multiplier for approximate applications," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 418–425, Nov 2015.

[53] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, pp. 62:1–62:33, Mar. 2016.

[54] K. M. Alajel, W. Xiang, and J. Leis, "Error resilience performance evaluation of h.264 i-frame and jpwl for wireless image transmission," in *2010 4th International Conference on Signal Processing and Communication Systems*, pp. 1–7, Dec 2010.

[55] H. R. Wu, W. Lin, and L. J. Karam, "An overview of perceptual processing for digital pictures," in *2012 IEEE International Conference on Multimedia and Expo Workshops*, pp. 113–120, July 2012.

[56] E. Özer, A. P. Nisbet, and D. Gregg, “Stochastic bit-width approximation using extreme value theory for customizable processors,” in *Compiler Construction*, pp. 250–264, Springer, 2004.

[57] K. Parashar, R. Rocher, D. Menard, and O. Sentieys, “A hierarchical methodology for word-length optimization of signal processing systems,” in *23rd Int. Conf. on VLSI Design (VLSID)*, pp. 318–323, 2010.

[58] D. Novo, I. Tzimi, U. Ahmad, P. Ienne, and F. Catthoor, “Cracking the complexity of fixed-point refinement in complex wireless systems,” in *IEEE Work. on Signal Processing Systems (SiPS'13)*, pp. 18–23, 2013.

[59] B. Widrow and I. Kollár, *Quantization Noise: Roundoff Error in Digital Computation, Signal Processing, Control, and Communications.* Cambridge University Press, 2008.

[60] G. Constantinides, “Perturbation analysis for word-length optimization,” in *11th Annual IEEE Symposium on FCCM*, pp. 81–90, 2003.

[61] D. Menard, R. Rocher, and O. Sentieys, “Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 55, Nov. 2008.

[62] J.-W. Weijers, V. Derudder, S. Janssens, F. Petré, and A. Bourdoux, “From MIMO-OFDM algorithms to a real-time wireless prototype: a systematic Matlab-to-hardware design flow,” *EURASIP J. Appl. Signal Process.*, vol. 2006, pp. 138–138, Jan. 2006.

[63] L. B. Jackson, “Roundoff-noise analysis for fixed-point digital filters realized in cascade or parallel form,” *IEEE Transactions on Audio and Electroacoustics*, vol. 18, no. 2, pp. 107–122, 1970.

[64] K. Ogata, *Modern Control Engineering.* Upper Saddle River, NJ, USA: Prentice Hall PTR, 4th ed., 2001.

[65] P. Brodatz, *Textures: A Photographic Album for Artists and Designers.* New-York: Dover Publications, 1966.

[66] J. Huang, J. Lach, and G. Robins, “A methodology for energy-quality tradeoff using imprecise hardware,” in *Proceedings of IEEE/ACM Design Automation Conference (DAC'12)*, pp. 504–509, June 2012.

[67] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, “Image quality assessment: from error visibility to structural similarity,” *IEEE Trans. on Image Processing*, vol. 13, pp. 600–612, April 2004.

[68] A. Tilley, “This mysterious chip in the iphone 7 could be key to apple's ai push.” `https://www.forbes.com/sites/aarontilley/2016/10/17/iphone-7-fpga-chip-artificial-intelligence/#1880dc003c69`, October 2017. Forbes.

[69] "Mentor graphics ac datatypes v3.7." `https://www.mentor.com/hls-lp/downloads/ac-datatypes`. Synthesizable C++ libraries.

[70] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with flopoco," *IEEE Design Test of Computers*, vol. 28, pp. 18–27, July 2011.

[71] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, pp. 20:1–20:17, July 2016.

[72] X. Fang and M. Leeser, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, pp. 20:1–20:17, July 2016.

[73] A. Mahzoon and B. Alizadeh, "Optifex: A framework for exploring area-efficient floating point expressions on fpgas with optimized exponent/mantissa widths," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, pp. 198–209, Jan 2017.

[74] S. Lloyd, "Least squares quantization in pcm," *IEEE Trans. on Information Theory*, vol. 28, pp. 129–137, March 1982.

[75] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

List of Figures

1 45 years of microprocessor trend data . . . 10

1.1 12-bit floating-point number . . . 16
1.2 Dual-path floating-point adder . . . 18
1.3 Basic floating-point multiplication . . . 19
1.4 12-bit fixed-point number . . . 21
1.5 Distribution of continuous signal quantization error . . . 23
(a) Quantization error distribution for rounding towards $-\infty$ . . . 23
(b) Quantization error distribution for rounding towards $+\infty$ . . . 23
(c) Quantization error distribution for rounding to nearest . . . 23
1.6 Representation of FxP quantization error as an additive noise . . . 23
1.7 Conventional rounding vs convergent rounding . . . 24
(a) Conventional rounding . . . 24
(b) Convergent rounding . . . 24
1.8 Quantization and rounding of a fixed-point number . . . 26
1.9 Fixed-point addition process . . . 28
1.10 One-bit addition function – Full adder . . . 28
1.11 8-bit Ripple Carry Adder . . . 29
1.12 8-bit Carry-Select Adder . . . 29
1.13 16-bit Brent-Kung Adder . . . 31
1.14 16-bit Kogge-Stone Adder . . . 32
1.15 General integer multiplication principle . . . 32
1.16 General visualization of multiplication summand grid . . . 32
1.17 Full adder compression . . . 33
1.18 Half adder transformation . . . 33
1.19 Wallace and Dadda trees . . . 33
(a) Wallace Tree . . . 33
(b) Dadda tree . . . 33
1.20 6-bit signed array multiplier . . . 34
1.21 Structures of AHA and AFA . . . 35
(a) Structure of AHA . . . 35
(b) Structure of AFA . . . 35
1.22 Sequential multiplier . . . 35

1.23 Probability for the longest carry chain of a 64-bit adder to be inferior to x as a function of x . . . 37
1.24 16-bit Sample Adder (LPA) with $k = 4$ . . . 39
1.25 Distribution of calculations for carry propagation matrix products [34] . . . 40
1.26 Consideration of carries in LPA and ACA output computation . . . 40
1.27 Error maps of 8-bit LPA and ACA adders for different values of k . . . 41
(a) $n = 8, k = 2$ . . . 41
(b) $n = 8, k = 4$ . . . 41
1.28 Hardware implementation of VLSA [34] . . . 42
1.29 Delay and area results for ACA with different bitwidth [34] . . . 43
1.30 Principle of ETAI . . . 44
1.31 ETAI control block . . . 45
1.32 ETAI accuracy simulation results [35] . . . 46
1.33 Hardware implementation of ETAII . . . 47
1.34 Hardware implementation of ETAIIM . . . 48
1.35 Consideration of carries in ETAIV output computation . . . 48
1.36 Error maps of 16-bit ETAIV for different values of X . . . 49
(a) $n = 16, X = 2$ . . . 49
(b) $n = 16, X = 3$ . . . 49
1.37 Hardware implementation of ETAIV . . . 49
1.38 Consideration of carries in AC2A output computation . . . 51
1.39 Accuracy vs power for AC2A and other approximate adders under VOS . . . 53
1.40 Structure of proposed n-bit GDA composed of 4 $n/4$-bit sub-adders . . . 56
1.41 GDA hierarchical prediction scheme . . . 57
1.42 GDA reconfigurable prediction scheme . . . 57
1.43 Error vs delay for an identical power consumption for GDA and AC2A . . . 61
(a) Worst-case error . . . 61
(b) Error rate . . . 61
(c) Average error . . . 61
1.44 Original and approximate MA transistor view . . . 63
(a) Original MA . . . 63
(b) Simplified MA . . . 63
(c) AMA1 . . . 63
(d) AMA2 . . . 63
1.45 DCT/IDCT test results for IMPACT . . . 64
1.46 Flowchart for probabilistic pruning optimization process [45] . . . 66
1.47 16-bit Weighted-Pruned Kogge Stone Adder (WPKSA) . . . 66
1.48 Probabilistic pruning results for Kogge-Stone and Han-Carlson adders [45] . . . 67
1.49 Two's-complement signed 6-bit AAMI structure . . . 68
1.50 Two's-complement signed 6-bit AAMII structure . . . 69
1.51 AAO, AA and ND-ND cells . . . 69
(a) AAO cell . . . 69
(b) AA cell . . . 69
(c) ND-ND cell . . . 69

1.52 Exhaustive search of $K_{\theta<n}$ and $K_{\theta=n}$ for a 6-bit multiplier 71
1.53 8-bit signed AAMIII structure 72
1.54 Error maps of 4-bit, 8-bit and 16-bit AAMIII 73
(a) $n = 4$ 73
(b) $n = 8$ 73
(c) $n = 16$ 73
1.55 Example of ETM multiplication process [35] 75
1.56 Structure of a 12-bit ETM [35] 75
1.57 Accuracy evaluation by simulation for ETM [35] 76
1.58 PDP for 12-bits ETM and array multiplier [35] 77
1.59 Summand grid for an 8-bit fixed-width Booth multiplier with LP_{major}-based error correction [49] 78
1.60 Partial product generation 79
1.61 Karnaugh map representation of approximate carry for $n = 10$ 80
(a) a_carry_0 80
(b) a_carry_1 80
1.62 Approximate carry generation circuit using ACGPII for $n = 32$ 82
1.63 LP_{minor} level discrimination for the design of FBMIII 84
1.64 8-bit FBMIII schematized structure 85
1.65 DRUM input unbiasing process 87
1.66 Structure of DRUM 88
1.67 Area and power benefits of 16-bit DRUM [52] 89
1.68 Error maps of 16-bit DRUM 90
(a) $n = 8, k = 2$ 90
(b) $n = 8, k = 4$ 90
(c) $n = 8, k = 6$ 90

2.1 Extraction of the mean square error of a fixed-point system 95
2.2 Comparison of noise parameters propagation using traditional flat, PSD agnostic and proposed PSD methods 97
(a) Traditional flat method: propagation of μ_i, σ_i^2 from each noise to output . 97
(b) PSD agnostic: blind propagation of μ_i, σ_i^2. Proposed PSD method: propagation of μ_i, σ_i^2, and PSD_i 97
2.3 SFG cycle breaking process example 100
(a) Cyclic SFG 100
(b) Equivalent acyclic SFG 100
2.4 Band-pass frequency filtering scheme 102
2.5 1-level DWT coder and decoder 102
2.6 E_d versus fractional bit-width d 103
2.7 E_d versus number of PSD samples N_{PSD} 104
2.8 Execution time in seconds and speed up for frequency filtering and DWT systems versus the number of PSD samples 105
(a) Frequency filtering 105
(b) Daubechies 9/7 DWT 105

2.9 Output frequency repartition of the fixed-point error after DWT encoding and decoding . . . 106
(a) Simulation . . . 106
(b) PSD estimation . . . 106

3.1 Error maps of 8-bit FxP quantization process and approximate operators . . . 108
(a) 8-to-4-bit quantization . . . 108
(b) 8-bit ACA, $k = 2$ . . . 108
(c) 8-bit AAM . . . 108
(d) 8-bit DRUM, $k = 4$ . . . 108
3.2 Propagation of BWER across an operator . . . 109
3.3 Extraction of Binary Error Event Vectors for BWER Model Training . . . 112
3.4 Example of Conditional Probabilities Training for $n = 4$, $m = 4$ and $k = 2$ . . . 113
3.5 Convergence of BWER Training in Function of k . . . 115
3.6 Tree Operation Structure with Three Stages . . . 116
3.7 BWER Estimation and Simulation Results for Stand-Alone ACA with $x = 2$ and $k = 4$ . . . 117
3.8 Evolution of estimation error D_B with k for different configurations of 16-bit approximate stand-alone adders and multipliers . . . 118
(a) ACA – Adder . . . 118
(b) IMPACT – Adder . . . 118
(c) AAM – Multiplier . . . 118
(d) DRUM . . . 118
3.9 Evolution of estimation error D_B with k for different configurations of 16-bit approximate adders with different number of stages . . . 119
(a) ACA – Adder . . . 119
(b) IMPACT – Adder . . . 119
3.10 Proposed Design Flow for Arithmetic Operator Characterization . . . 123
3.11 Distribution of BER in output bits of 8-bit RCA under voltage scaling . . . 125
3.12 Distribution of BER in output bits of 8-bit BKA under voltage scaling . . . 125
3.13 Equivalence Between Faulty Hardware Operator and Equivalent Functionally Faulty Operator . . . 126
3.14 Design flow of modelling of VOS operators . . . 127
3.15 Estimation Error of the Model for Different Adders and Distance Metrics . . . 129
(a) Signal to Noise Ratio . . . 129
(b) Normalized Hamming Distance . . . 129
3.16 BER and Energy for Different VOS Triads Applied to 16-bit RCA . . . 130

4.1 First version of ApxPerf . . . 135
4.2 Second version of ApxPerf . . . 136
4.3 Direct comparison of 16-bit-input fixed-point and approximate adders regarding MSE . . . 141
(a) MSE vs power . . . 141
(b) MSE vs delay . . . 141

(c) MSE vs PDP . . . 141
(d) MSE vs area . . . 141
4.4 Direct comparison of 16-bit-input fixed-point and approximate adders regarding BER . . . 142
(a) BER vs power . . . 142
(b) BER vs delay . . . 142
(c) BER vs PDP . . . 142
(d) BER vs area . . . 142
4.5 Power Consumption of FFT-32 Versus Output PSNR Using 16-bit Approximate Adders . . . 145
4.6 Power Consumption of DCT in JPEG Encoding Versus Output MSSIM Using 16-bit Approximate Operators . . . 146
4.7 $Lena$ Encoded with DCT Instrumented With Different 16-bit Approximate Operators . . . 147
(a) $\text{ADD}_t(16, 10)$ . . . 147
(b) $\text{ACA}(16, 2)$ . . . 147
(c) $\text{ETAIV}(16, 4)$ . . . 147
(d) $\text{IMPACT}(16, 8, 3)$ . . . 147

5.1 Representation of the Power Spent by a Circuit in One Cycle . . . 158
5.2 Relative Area, Delay and Energy per Operation Comparison Between Fixed-Point and Floating-Point for Different Fixed-Width Adders . . . 162
5.3 Relative Area, Delay and Energy per Operation Comparison Between Fixed-Point and Floating-Point for Different Fixed-Width Multipliers . . . 163
5.4 2-D K-means Clustering Golden Output Example, Obtained Using Floating-Point Double-Precision . . . 166
5.5 K-Means Clustering Outputs for 8- and 16-bit floating-point and fixed-point with Accuracy Target of 10^{-4} . . . 169
(a) $\text{ct_float}_8(5)$ . . . 169
(b) $\text{ac_fixed}_8(3)$ . . . 169
(c) $\text{ct_float}_{16}(5)$ . . . 169
(d) $\text{ac_fixed}_{16}(3)$ . . . 169
5.6 Energy Versus Classification Error Rate for K-Means Clustering . . . 170
5.7 Fixed-Point and Floating-Point Energy per Operation vs MSE for FFT-16 for Different Bit-Widths . . . 171

List of Tables

1.1 IEEE 754 normalized floating-point representation 17
1.2 Cost of FlP addition vs integer addition 18
1.3 Cost of FlP multiplication vs integer multiplication 20
1.4 Mean and variance of quantization error 25
1.5 Rounding direction vs round/sticky bit 26
1.6 Full adder truth table 28
1.7 Radix-4 modified Booth encoding 36
1.8 Bounds on the longest run of 1's with high probability 38
1.9 Logical equations of CSGC type I and II 45
1.10 AP as a function of MAA and carry propagation block size for 32-bit ETAII . . 47
1.11 AP as a function of MAA and X for 32-bit ETAIV 49
1.12 Simulation results for Error-Tolerant Adders [37] 50
1.13 Estimated parameters of the approximate computation part of a 16-bit AC2A relatively to a conventional 16-bit CLA 51
1.14 Comparison of 16-bit AC2A approximate part with other adders 52
1.15 Error correction cycles in AC2A 54
1.16 Accuracy and power consumption of 4-stage pipelined 32-bit AC2A as a function of the active mode 55
1.17 Comparison between 32-bit GDAs and exact and approximate static adders . . 59
1.18 Accuracy-configurable implementation of AC2A and GDA 60
1.19 Truth tables of accurate and approximate MA cells 63
1.20 Accuracy comparison of AAMI and AAMII [47] 70
1.21 Accuracy comparison of signed AAM versions I, II and III [48] 73
1.22 Area ratio comparing to original parallel adder for AAM versions I, II and III [48] 74
1.23 Equivalence between Radix-4 modified-Booth-encoded symbol Y_i and control bits in partial product generation 78
1.24 Representation of approximate carry values 80
1.25 Accuracy comparison for FBMI and FBMII using ACGPI 81
1.26 Approximate carry signals generated by ACGPI and ACGPII for $n = 8$ 82
1.27 Comparison of delay and area of ACGPI and ACGPII [50] 82
1.28 Accuracy comparison for FBMI and FBMII using ACGPII 83
1.29 Value of $P\left(pp_{i,j} = k\right)$ in LP 85
1.30 Accuracy comparison for FBMI and FBMIII 86

1.31 Error results for 16-bit DRUM 89

2.1 Relative error power estimation statistics E_d 103
2.2 Comparison of E_d between PSD agnostic method and proposed PSD method . 104

3.1 Storage Cost of BWER Propagation Full Data Structure 110
3.2 Storage Cost of BWER Propagation Data Structure for 16-bit Addition and Multiplication Depending on Reduction Method 111
3.3 Partial Input BWER for the Example 113
3.4 BWER Propagation and Simulation Time of Stand-Alone Approximate Operators 120
3.5 Carry propagation probability table of modified 4-bit adder 127

4.1 Direct comparison of 16-bit-input and output fixed-point and approximate multipliers 143
4.2 Accuracy and Energy Consumption of FFT-32 Using 16-bit Fixed-Width Multipliers 145
4.3 Accuracy and Energy Consumption of Distance Computation for HEVC Filter Using 16-bit Input Adders 148
4.4 Accuracy and Energy Consumption of Distance Computation for HEVC Using 16-bit Input Multipliers 148
4.5 Accuracy and Energy of Distance Computation for K-means Clustering Using 16-bit Input Adders for Different Success Rates 149
4.6 Accuracy and Energy of Distance Computation for K-means Clustering Using 16-bit Input Multipliers 149

5.1 Main Properties of Custom Floating-Point Libraries AC_FLOAT, CT_FLOAT and FLOPOCO 157
5.2 Comparative Results for 16-bit Custom Floating-Point Addition/Subtraction with $F_{\mathtt{clk}} = 200\text{MHz}$ 159
5.3 Comparative Results for 16-bit Custom Floating-Point Multiplication with $F_{\mathtt{clk}} = 200\text{MHz}$ 159
5.4 Comparative Results for 32-bit Custom Floating-Point Addition/Subtraction with $F_{\mathtt{clk}} = 200\text{MHz}$ 160
5.5 Comparative Results for 32-bit Custom Floating-Point Multiplication with $F_{\mathtt{clk}} = 200\text{MHz}$ 160
5.6 8- and 16-bit Performance and Accuracy for K-Means Clustering Experiment . 168

Abstract

The physical limits being reached in silicon-based computing, new ways have to be found to overcome the predicted end of Moore's law. Many applications can tolerate approximations in their computations at several levels without degrading the quality of their output, or degrading it in a acceptable way. This thesis focuses on approximate arithmetic architectures to seize this opportunity. Firstly, a critical study of state-of-the-art approximate adders and multipliers is presented. Then, a model for fixed-point error propagation leveraging power spectral density is proposed, followed by a model for bitwise-error rate propagation of approximate operators. Approximate operators are then used for the reproduction of voltage over-scaling effects in exact arithmetic operators. Leveraging our open-source framework ApxPerf and its synthesizable template-based C++ libraries apx_fixed for approximate operators, and ct_float for low-power floating-point arithmetic, two consecutive studies are proposed leveraging complex signal processing applications. Firstly, approximate operators are compared to fixed-point arithmetic, and the superiority of fixed-point is highlighted. Secondly, fixed-point is compared to small-width floating-point in equivalent conditions. Depending on the applicative conditions, floating-point shows an unexpected competitiveness compared to fixed-point. The results and discussions of this thesis give a fresh look on approximate arithmetic and suggest new directions for the future of energy-efficient architectures.