N° d'ordre NNT : 2025ISAL0079

THESE de DOCTORAT DE L'INSA LYON,
membre de l'Université de Lyon

Ecole Doctorale 512
Informatique et Mathématiques

Spécialité/ discipline de doctorat : Informatique

Soutenue publiquement le 17/09/2025, par :

Orégane Desrentes

Hardware Arithmetic Acceleration for Machine Learning and Scientific Computing

---

Devant le jury composé de :

Muller	Jean-Michel	Directeur de Recherche	CNRS	Président du jury
Bruguera	Javier Diaz	Senior Principal Design Eng.	ARM	Rapporteur
Sentieys	Olivier	Professeur des Universités	Université de Rennes	Rapporteur
Chotin	Roselyne	Professeure des Universités	Sorbonne Université	Examinatrice
Collange	Caroline	Chargée de Recherche	INRIA de Rennes	Examinatrice
De Dinechin	Florent	Professeur des Universités	INSA de Lyon	Directeur de thèse
Dupont de Dinechin	Benoît	Directeur de la technologie	Kalray S.A.	Co-enc. de thèse

Département FEDORA – INSA Lyon - Ecoles Doctorales

SIGLE	ECOLE DOCTORALE	NOM ET COORDONNEES DU RESPONSABLE
ED 206 CHIMIE	**CHIMIE DE LYON** **https://www.edchimie-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL 7, Avenue Jean Capelle 69621 Villeurbanne CEDEX **secretariat@edchimie-lyon.fr**	**M. Stéphane DANIELE** C2P2-CPE LYON-UMR 5265 Bâtiment F308, BP 2077 43 Boulevard du 11 novembre 1918 69616 Villeurbanne CEDEX **directeur@edchimie-lyon.fr**
ED 341 E2M2	**ÉVOLUTION, ÉCOSYSTÈME, MICROBIOLOGIE, MODÉLISATION** **http://e2m2.universite-lyon.fr** Sec. : Bénédicte LANZA Bâtiment Atrium, 43, Boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX Tél : 04.72.44.83.62 **secretariat.e2m2@univ-lyon1.fr**	**Mme Sandrine CHARLES** Université Claude Bernard Lyon 1 UFR Biosciences, Bâtiment Mendel 43, boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX **e2m2.codir@listes.univ-lyon1.fr**
ED 205 EDISS	**INTERDISCIPLINAIRE SCIENCES-SANTÉ** **http://ediss.universite-lyon.fr** Sec. : Bénédicte LANZA Bâtiment Atrium, 43, Boulevard du 11 Novembre 1918 69622 Villeurbanne CEDEX Tél : 04.72.44.83.62 **secretariat.ediss@univ-lyon1.fr**	**M. Jérôme LAMARTINE** Université Claude Bernard Lyon 1 Equipe Intégrité Fonctionnelle du Tissu Cutané LBTI - UMR5305 CNRS IBCP - 7 passage du Vercors, 69367 Lyon CEDEX 07 Tel : 04.72.72.26.66 **jerome.lamartine@univ-lyon1.fr**
ED 34 EDML	**MATÉRIAUX DE LYON** **http://ed34.universite-lyon.fr** Sec. : Yann DE ORDENANA Tél : 04.72.18.62.44 **yann.de-ordenana@ec-lyon.fr**	**M. Stéphane BENAYOUN** Ecole Centrale de Lyon Laboratoire LTDS 36 avenue Guy de Collongue 69134 Ecully CEDEX Tél : 04.72.18.64.37 **stephane.benayoun@ec-lyon.fr**
ED 160 EEA	**ÉLECTRONIQUE, ÉLECTROTECHNIQUE, AUTOMATIQUE** **https://edeea.universite-lyon.fr** Sec. : Philomène TRECOURT Bâtiment Charlotte PERRIAND 69621 Villeurbanne CEDEX Tél : 04.72.43.71.70 **secretariat.edeea@insa-lyon.fr**	**M. Philippe DELACHARTRE** INSA LYON Laboratoire CREATIS Bâtiment Blaise Pascal, 7 avenue Jean Capelle 69621 Villeurbanne CEDEX Tél : 04.72.43.88.63 **philippe.delachartre@insa-lyon.fr**
ED 512 INFOMATHS	**INFORMATIQUE ET MATHÉMATIQUES** **http://edinfomaths.universite-lyon.fr** Sec. : Renée EL MELHEM Bât. Blaise PASCAL 7, Avenue Jean Capelle 69621 Villeurbanne CEDEX **infomaths@univ-lyon1.fr**	**M. Dragos IFTIMIE** Université Claude Bernard Lyon 1 Bâtiment Braconnier 21 Avenue Claude BERNARD 69 622 Villeurbanne CEDEX Tél : 04.72.44.79.58 **direction.infomaths@listes.univ-lyon1.fr**
ED 162 MEGA	**MÉCANIQUE, ÉNERGÉTIQUE, GÉNIE CIVIL, ACOUSTIQUE** **http://edmega.universite-lyon.fr** Sec. : Philomène TRECOURT Bâtiment Charlotte PERRIAND 69621 Villeurbanne CEDEX Tél : 04.72.43.71.70 **mega@insa-lyon.fr**	**M. Etienne PARIZET** INSA Lyon Laboratoire LVA Bâtiment St. Exupéry 25 bis av. Jean Capelle 69621 Villeurbanne CEDEX **etienne.parizet@insa-lyon.fr**
ED 483 ScSo	**ScSo**[1] **https://edsciencessociales.universite-lyon.fr** Sec. : Mélina FAVETON Tél : 04.78.69.72.76 Université Lyon 2 Campus Berges du Rhône, Bureau BEL 309 18, Quai Claude BERNARD 69365 LYON CEDEX 07 **melina.faveton@univ-lyon2.fr**	**M. Bruno MILLY** Univ. Lyon 2 Campus Berges du Rhône 18, quai Claude Bernard 69365 LYON CEDEX 07 Bureau BEL 319 **bruno.milly@univ-lyon2.fr**

[1] ScSo : Histoire, Géographie, Aménagement, Urbanisme, Archéologie, Science politique, Sociologie, Anthropologie

Contents

List of Figures

List of Tables

List of Algorithms

Résumé

Dans un monde axé sur les données, l'apprentissage artificiel et le calcul scientifique sont devenus de plus en plus importants, justifiant l'utilisation d'accélérateurs matériels dédiés. Cette thèse explore la conception et l'implémentation d'unités arithmétiques pour de tels accélérateurs dans le Massively Parallel Processor Array de Kalray.

L'apprentissage artificiel nécessite des multiplications de matrices qui opèrent sur des formats de nombres très petits. Dans ce contexte, cette thèse étudie l'implémentation du produit-scalaire-et-addition en précision mixte pour différents formats de 8 et 16 bits (FP8, INT8, Posit8, FP16, BF16), en utilisant des variantes d'une technique classique de l'état-de-l'art, l'accumulateur long. Elle introduit également des techniques permettant de combiner différents formats d'entrée. Des méthodes radicalement différentes sont étudiées pour passer à l'échelle vers la grande dynamique des formats 32 et 64 bits utilisés en calcul scientifique.

Cette thèse étudie également l'évaluation de certaines fonctions élémentaires. Un opérateur pour la fonction exponentielle (cruciale pour le calcul de la fonction softmax) étend une architecture de l'état-de-l'art pour accepter des formats d'entrée multiples. La fonction racine carrée inverse (utilisée pour la normalisation des couches) est accélérée en combinant des techniques d'état-de-l'art pour la réduction de la dynamique, des tables multipartites correctement arrondies et des techniques logicielles de raffinement itératif.

Mot-clefs : architecture des ordinateurs, arithmétique des ordinateurs, nombre flottant, produit scalaire, fonctions élémentaires

Abstract

In a data-driven world, machine learning and scientific computing have become increasingly important, justifying dedicated hardware accelerators. This thesis explores the design and implementation of arithmetic units for such accelerators in Kalray's Massively Parallel Processor Array.

Machine learning requires matrix multiplications that operate on very small number formats. In this context, this thesis studies the implementation of mixed-precision dot-product-and-add for various 8-bit and 16-bit formats (FP8, INT8, Posit8, FP16, BF16), using variants of a classic state-of-the-art technique, the long accumulator. It also introduces techniques to combine various input formats. Radically different methods are studied to scale to the larger range of 32-bit and 64-bit formats common in scientific computing.

This thesis also studies the evaluation of some elementary functions. An operator for exponential function (crucial for softmax computations) extends a state-of-the-art architecture to accept multiple input formats. The inverse square root function (used for layer normalisation) is accelerated by combining state-of-the-art techniques for range reduction, correctly rounded multipartite tables, and software iterative refinement techniques.

Keywords: computer hardware, computer arithmetic, floating point, dot product, elementary functions

Remerciements

Je remercie les membres de mon jury Caroline Collange, Roselyne Chotin, Jean-Michel Muller, et les rapporteurs Olivier Sentieys et Javier Diaz Bruguera pour leur lecture attentive de mon manuscrit.

Je tiens à remercier tout particulièrement mes encadrants : mon directeur de thèse Florent de Dinechin et mon référent scientifique et co-encadrant Benoît Dupont de Dinechin. Merci Benoît pour ton engagement dans la recherche, merci d'avoir lu (et de te souvenir !) *tous* les papiers, un super-pouvoir qui a été précieux pendant ma thèse. Merci Florent pour ton aide, tes conseils, ton orientation, pour nos échanges de gribouillis sur un coin de tableau, pour m'avoir transmis l'amour de l'arithmétique matérielle et du tikz, pour ton implication dans la végétalisation du labo, ...

Merci à Pierre Cochard pour avoir sauvé FloPoCo de CMake, avoir amélioré le confort de développement de mille façons différentes, et m'avoir empêché d'y mettre trop de bêtises. Merci infiniment pour ton hospitalité et ton soutien pendant la préparation de la soutenance (et du pot !). L'odeur du thé sera à jamais associée au calme rassurant de ton appartement dans mon esprit.

Pour leur soutien moral et technique à Kalray, je remercie Julien Le Maire et Arthur Vianes.

Merci à mes différents voisins de bureau, Maël, Pierre R. et Sylvain à Kalray, et Agathe, Bastien, Jurek, Luc, Louis, Maxime C., Maxime P., Tommy, Romain, Pierre C. au CITI.

Merci aux membres permanents de l'équipe Émeraude, Anastasia, Anouchka, Cécilia, Christine, Linda, Romain, Stéphane, Tanguy et Yann.

Merci à Donald Knuth et Leslie Lamport pour LaTeX et la classe `book` avec lequel j'ai écrit cette thèse, à Guillaume Salagnac pour son script `latex-compile` ainsi que son support très réactif, et à Detexify et Tex Stack Exchange pour avoir déjà posé et répondu à toutes mes questions.

Merci aux différents professeurs qui ont nourri ma curiosité au fil des années: Anabelle, Nadège, Mme Bellecourt, Isabel, Mme Richelet, M. Clary et Daniel Hirschkoff.

Pour leur accompagnement musical, je remercie l'orchestre AMOSUD pour la bouffée d'air frais qu'ont été les répétitions pendant ces trois ans, ainsi que Owl City qui fut le fond musical de ma rédaction.

Pour leur soutien pendant et avant cette thèse, merci à mes amis, Alyd, Chloé, Davy, Fiona, Gaëlle, Gaétan, Jelle, Jérôme, Joseph, Léo, Loup, Lucy, Max, Pierre, Rebecca, Rémi, Tolgahan, Thibault, Véronique,...

Merci à ma famille, en particulier mes parents, Auxence et Françoise pour leur soutien, ainsi qu'à la famille Ollivier.

Merci Antoine.

Introduction

In today's data-driven world, there is a growing demand for advanced hardware solutions capable of handling the complex computational requirements of machine learning and scientific computing applications. This trend is pushed by the increasing size and complexity of data sets, such as those used in deep learning models and scientific simulations. These applications often involve processing vast amounts of data in parallel and performing complex mathematical operations, which can be computationally intensive and time-consuming.

These computational requirements have been growing exponentially since the 60's, following the supply dictated by *Moore's Law*: *The number of transistors in an integrated circuit doubles about every two years*. While the promised growth in computational power is running out of steam, the demand shows no sign of stopping and, even worse, seems to snowball.

Deep learning models, for instance, involve training artificial neural networks with large and intricate data sets, which can require significant computational resources. Similarly, scientific simulations in fields such as global climate models, molecular dynamics, and fluid dynamics can involve modelling complex physical phenomena, which can also necessitate large-scale computations.

To address the pressing need for more efficient hardware solutions, researchers and industry professionals have been exploring various approaches. One such approach is the development of manycore processors, which are designed to perform multiple computations in parallel using a large number of processing cores. An example of a manycore processor is the Massively Parallel Processor Array (MPPA) developed by Kalray. Kalray is a fabless company specializing in manycore accelerators for edge artificial intelligence and data centre storage applications. The company's mission is to provide high-performance and cost-efficient processing solutions for data-intensive workloads.

During my tenure with the company, I had the opportunity to contribute to the development of hardware accelerators for the MPPA as the designer for the floating-point arithmetic units. The research presented in this thesis is a reflection of my work on the MPPA and aims to contribute to the existing knowledge base, specifically in the areas of matrix multiplication and numerical function implementations. Matrix multiplication is a fundamental operation in many machine learning and scientific computing applications, and efficient implementation of this operation is crucial for achieving high performance. Numerical functions, such as square root and exponential functions, are also commonly used in these applications and their implementation on hardware can significantly impact the overall performance of the system.

This thesis contributes to the field of computer arithmetic, which is the art of

implementing such mathematical operations in computing devices and studying their properties, performance but also accuracy.

In this thesis, we propose novel designs for matrix multiplication and numerical function implementations on the MPPA architecture. We evaluate the performance of our designs using both theoretical analysis and simulations, and compare them with state-of-the-art implementations.

This thesis is organised as follows:

In Part I, we provide the necessary context for understanding the motivation behind our research. Chapter 1 provides an overview of how computers are constructed, from the microelectronics to the architecture of the system, highlighting the various trade-offs imposed by physics and challenges faced by hardware designers. Chapter 2 introduces general-purpose number formats: integers, fixed-point and floating-point formats. Chapter 3 discusses the computational demands of machine learning, and Chapter 4 describes the implementation of fixed-point arithmetic in hardware.

In Part II, we focus on matrix multiplication, which is a fundamental operation in machine learning algorithms. Chapter 5 discusses exact dot product algorithms for small precisions. Chapter 6 relaxes exactness to be able to scale to larger precisions. Chapter 7 presents methods for correctly rounding dot products in the large precisions used in scientific computing.

Finally, in Part III, we explore the implementation of various numerical functions on fixed-point hardware. Chapter 8 describes the state-of-the-art hardware implementation methods for numerical functions, followed by Chapters 9 and 10, which respectively focus on the implementation of the reciprocal square root and exponential functions.

This thesis has lead to 2 patents and 4 publications, which are detailed in the conclusion (see Chap. 10.4).

Part I

Context

Chapter 1

From Electronics to Computers

(...) a CPU is literally a rock that we tricked into thinking not to oversimplify: first you have to flatten the rock and put lightning inside it

– @daisyowl, 2017, on twitter

A *computer* is defined by the Cambridge dictionary as an electronic machine that is used for storing, organising, and finding words, numbers, and pictures, for doing calculations, and for controlling other machines.

Computers are everywhere, from coffee machines, to internet routers, phones, mega-servers, and of course the personal computer used to write this thesis.

As their name implies, a main application of computers is computing. Computers operate on data that is stored in a way it can understand and access, called a memory. They should be able to be programmed to do different computations depending on the required task, or the contents of the data.

A computer as we imagine it often includes a keyboard, a monitor, a mouse. This thesis focuses on the heart of the computer, the chip, which is a very large electronic circuit, built on a plate of semiconductor material. Circuits are built with electronic components like transistors and wires.

This chapter will first describe those electronic components, then how they are assembled into circuits. It also discusses the overall architecture of a computer, giving insight on how the arithmetic units designed in this thesis fit into the larger computer system. Finally, it presents the various tools used to aid circuit design in this thesis.

1.1 Electronics

As components get smaller, and computer systems grow larger and more complex, physics gets harder to ignore for hardware designers. The metrics commonly used in the literature to evaluate a circuit include its area, its power consumption, and the time taken to perform the computation: the latency. Those metrics are linked, often involving trade-offs where improving one metric worsens the others. This section aims to provide insight on how design choices impact those metrics, and what are the physical limitations impacting circuit design.

1.1.1 Semiconductors and Diodes

Semiconductor

Semiconductor materials like germanium or silicon are the basis for computers [89]. Those semiconductors are electric insulators at room temperature, but can conduct electricity in specific cases.

A pure crystal of semiconductor is "doped" by adding impurities. It can be doped positively (*p-type*) or negatively (*n-type*). *p-type* doping adds gallium or boron to the semiconductor crystal, causing a lack of electrons, or a surplus of electron holes compared to the pure crystal. *n-type* doping is done by adding arsenic or phosphorus, causing a surplus of electrons.

Simple Usage: a Diode

The simplest way a semiconductor is used is in a *p-n junction* diode (Fig. 1.1). This means that the diode is made by placing a *p-doped* crystal next to a *n-doped* crystal.

When no tension is applied to the diode (Fig. 1.1), silicon at the junction exchange electrons so that a small barrier of non-doped (and insulated) silicon is created. This zone is called the depletion region.

When sufficient voltage (Fig. 1.2, left) is applied between the poles of the diode, in this case from the $-$ to the $+$ pole, the diode lets current flow. This direction is

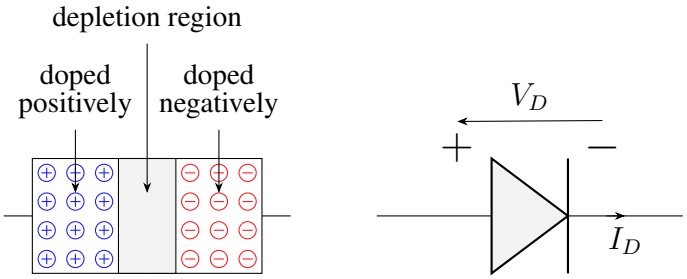


Figure 1.1: *p-n junction* or diode (left) and its representation (right).

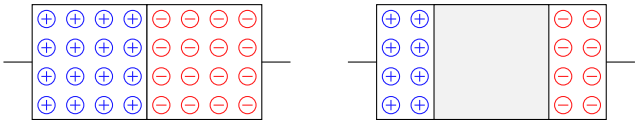


Figure 1.2: Diode when positive voltage is applied (left) and when negative voltage is applied (right).

called the forward bias of the diode. In this situation, electrons flow contrary to the direction of the current, from the $+$ pole to the $-$ pole. If the voltage is large enough, electrons charge negatively the silicon barrier from the negative side, reducing the size of the depletion region. Once the barrier is removed, the current can flow.

When voltage is applied in the other direction, the reverse bias, current cannot flow. The electrons, who would be circulating from the $+$ to the $-$ poles, meet with positively charged silicon atoms, increasing the size of the depletion region of insulating silicon (Fig. 1.2, right). This prevents the flow of current.

In essence, a diode can unidirectionally conduct electricity when the voltage applied to it is greater than a threshold ($0.7\,\mathrm{V}$ for silicon, $0.3\,\mathrm{V}$ for germanium). This property can be used, with some modification to the component, to create a electronically controllable switch[1].

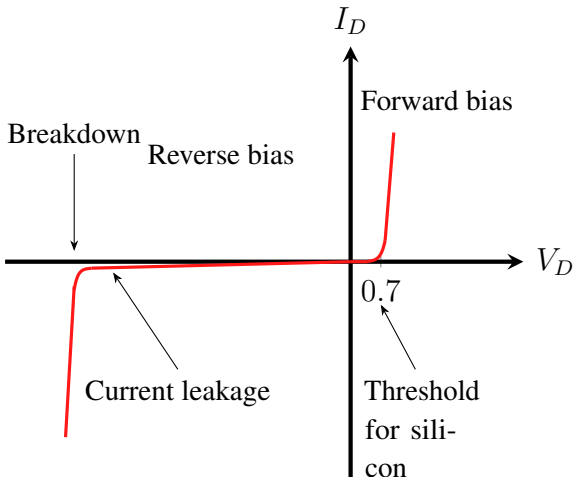


Figure 1.3: Graph of the characteristic of a diode.

[1]A switch is a component that closes (current can pass) or opens (current cannot pass) an electrical circuit. A common example is a light switch, where mechanical action from the user enables to close the electrical circuit (turning on the lights) or to open the circuit (turning off the lights).

If the voltage is too strong in reverse bias, the diode can break (Fig. 1.3). Broken components can completely change the behaviour of a circuit, and exposing semiconductors to a voltage it cannot handle is to be avoided at all cost. In the remainder of this thesis, it is assumed that semiconductors are all working.

Heat Generation

Some superficial currents can still be present in reverse bias, called leakage current. The power dissipated by the diode is $P_D = V_D \times I_D$, the voltage times the intensity of the current. This dissipated power is exclusively transformed into heat. Leakage current means that a diode will generate a little heat even when it is not conducting electricity. When in forward bias, the diode generates a lot of heat, as a lot of current passes through it.

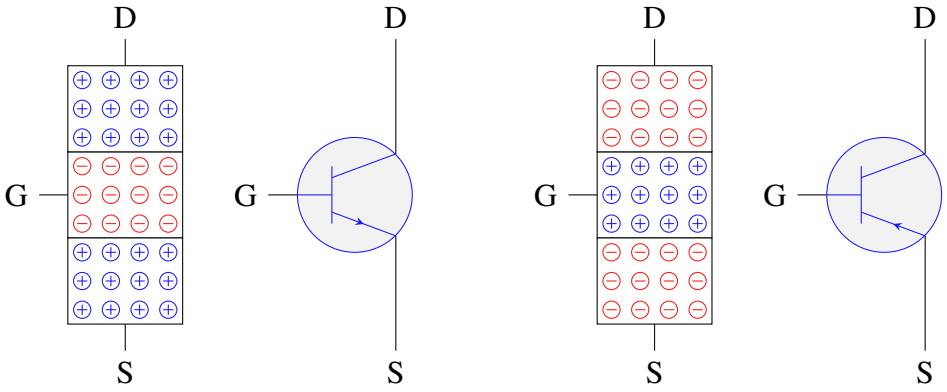
Germanium was historically the first material used as semiconductor, as it is easier to purify than silicon. However, silicon is the first choice for modern computers as it is more energy efficient, with less current leakage, and easier to mass manufacture.

The voltage threshold for silicon is about $0.7\,\mathrm{V}$ at room temperature ($25\,^\circ\mathrm{C}$), and reduces by $2\,\mathrm{mV}$ for every degree the temperature rises (about $0.6\,\mathrm{V}$ at $75\,^\circ\mathrm{C}$). If the voltage threshold gets too low, the functioning of the resulting circuit can be altered. This makes temperature control and power dissipation in semiconductor circuits crucial to their functioning.

1.1.2 Transistor

Current-Controlled Transistor

The first kind of transistor is a *p-n-p junction* or *n-p-n junction* (Fig. 1.4). The transistor is connected to three wires, the gate (G), the drain (D) and the source (S).



(a) *p-n-p junction* transistor. **(b)** *n-p-n junction* transistor.

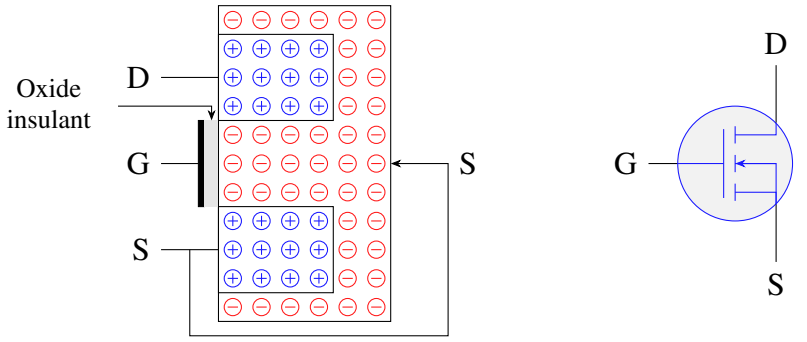
Figure 1.4: Current-controlled transistors and their representation.

The gate controls whether current can pass through the transistor, as if it was controlling a switch.

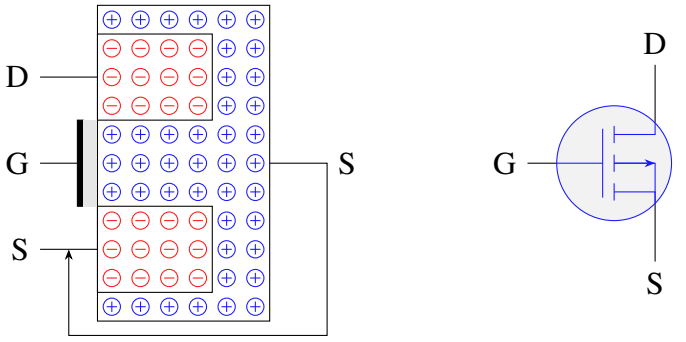
This type of transistor is controlled by variations in current. For example with a *n-p-n* transistor, current flowing from G to S leads to electrons entering the p zone

from G at the junction, and enables current to pass through from D to S. The same thing happens in reverse for a *p-n-p* transistor.

Voltage-Controlled Transistor



(a) *p-channel* transistor.



(b) *n-channel* transistor.

Figure 1.5: Voltage-controlled transistors and their representation.

In the case of computers, voltage controlled (Fig. 1.5) transistors are preferred to current controlled ones.

The technology of transistors commonly used is called E-MOSFET:

- *FET* means Field Effect Transistor, which means that the transistor is controlled by the electrical field (which is linked to the voltage).
- *MOS* means Metal-Oxide-Semiconductor, which is how the transistor is constructed: the gate is metal, the oxide electrically isolates the two sides of the gate, and the other side of the capacity is the semiconductor.
- *E* means that it is a Enrichment MOSFET transistor, the transistor is off when the voltage between the gate and the source is zero. The other way around would be a Depletion MOSFET transistor.

Voltage controlled transistors (Fig. 1.6) have a capacity effect between the gate and the semiconductor junction. When tension V_{in} is high, charges accumulate on both sides of the capacity. This changes the state of the semiconductor and closes the circuit between D and S.

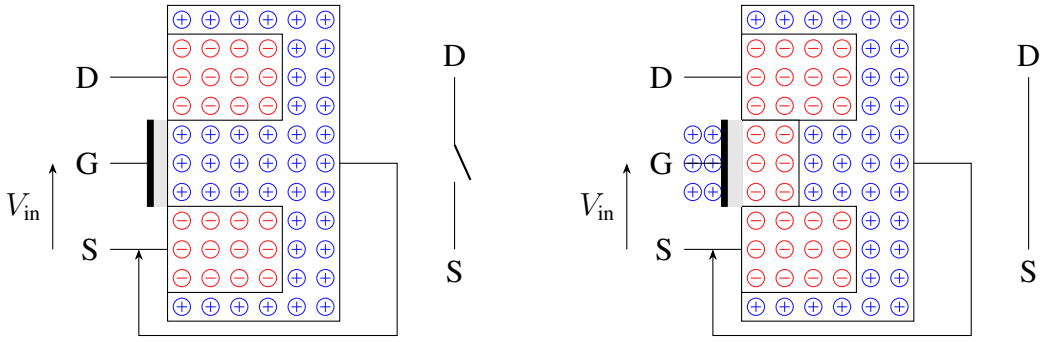


Figure 1.6: Functioning of a MOS n-channel transistor, without voltage between G and S (left), and with voltage between G and S (left), as well as their switch equivalent.

Current (ideally) only flows through the transistors when V_{in} changes, making them very energy efficient compared to current-controlled transistor. When the transistor changes state, current flows which dissipates power (following the same approximation $P = V \times I$ as the diode). The power dissipated when changing state is called dynamic power. Of course, there are still leakage currents, thus leakage power, at a much smaller magnitude, consuming power as soon as the circuit is on.

When parts of the circuit is unused, one can save dynamic power consumption by keeping the inputs constant. To save leakage power, that part of the chip must be unpowered.

A disadvantage of the voltage controlled transistors is that due to capacitance effects, the transistors are slow to change state. Stronger current can be used to charge the capacity faster, which makes the transistor faster to change state, however stronger currents means more heat. Smaller capacities also charge faster for the same current intensity but it reduces how much current can pass from the Drain to the Source. However, the capacity of a transistor uncharges into the previous transistor in the circuit: making one transistor faster this way makes the next one slower.

CMOS Technology

Computers currently use a technology called CMOS (Complementary MOS) to assemble the E-MOSFET transistors, in which p-channel transistors and an n-channel transistors are used in pairs.

The simplest example is an inverter (Fig. 1.7). The output wire V_{out} is connected to both the Drains of the transistors. If V_{in} is larger than the threshold, the transistors connect the ground to the output wire. If V_{in} is smaller than the threshold, the transistors connect the power wire to the output wire.

The p-channel transistor circuit conditionally connects the output to the wire with a strong voltage Vdd, enabling to output a voltage of Vdd V. The n-channel transistor circuit conditionally connects the output to the ground, enabling to output $0\,\mathrm{V}$. Those two circuits are dual of each other: if the p-channel transistor circuit has two transistor in series, the n-channel transistor circuit will have them in parallel, like for a NOR gate (Fig. 1.8).

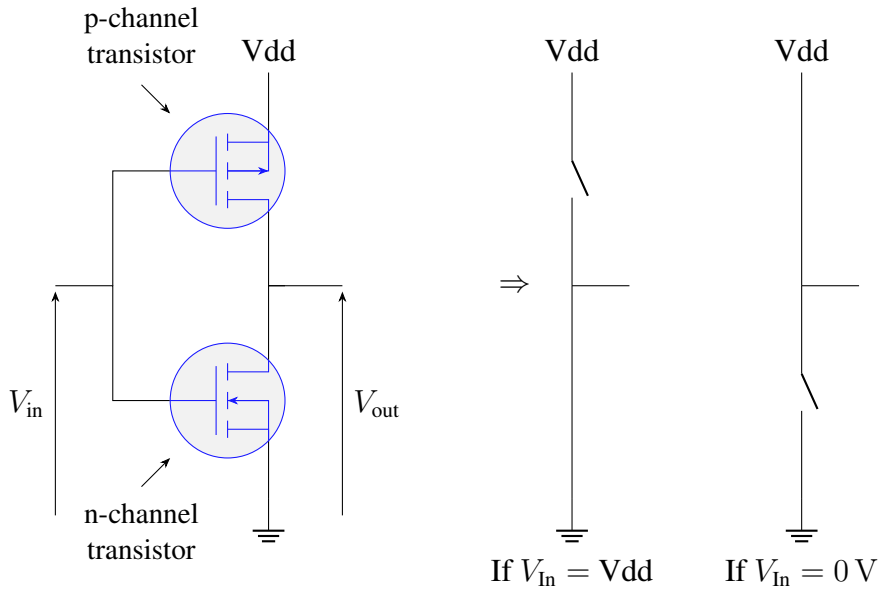


Figure 1.7: CMOS inverter.

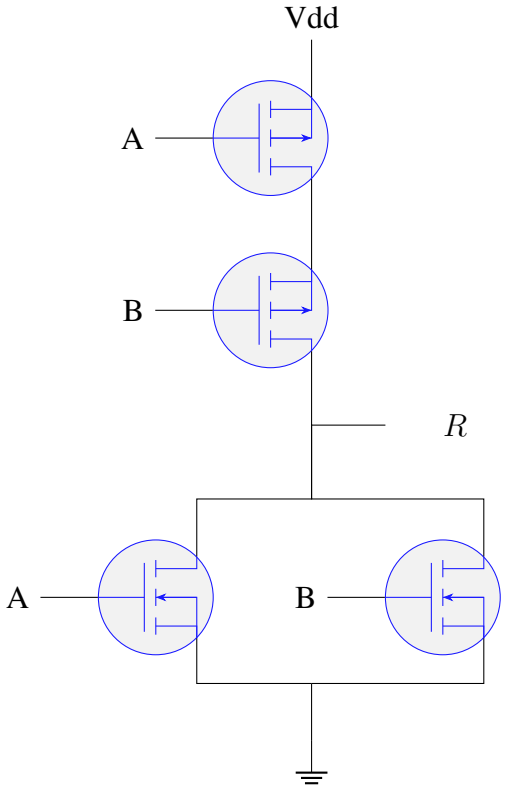


Figure 1.8: CMOS NOR gate.

Construction of Transistors in a Semiconductor Plate

MOSFET transistors were a revolution not only because of the voltage control, but also because their construction enables them to be etched into silicon plates where they are packed very densely together. The nano-scale circuit is constructed in multiple steps, using for each step blueprints called masks. First, p-doped and n-doped silicon are placed on the substrate, then oxide, the metal gate, some routing metal layers, and then substrate again to create multiple transistor layers on the chip.

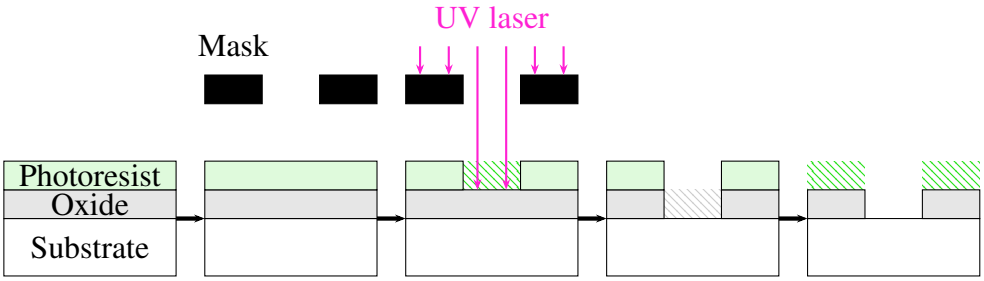


Figure 1.9: Some steps of photolithography for the oxide layer (not to scale).

This nano-scale etching of the circuit on the plate is made using a process called photolithography (Fig.1.9). Taking as example the oxide layer, which constructs the insulation between the gate and the semiconductor. The silicon plate is first oxidised forming a layer of insulant SiO_2. It is then covered in a material called photoresist. A mask (Fig. 1.10) of the desired circuit is placed above the plate, and an UV laser shining through will remove the photoresist where the plate is not covered by the mask, by making it soluble in a developer solution. In the etching phase, a chemical agent removes the oxide layer, except where it is protected by the photoresist. The remainder of the photoresist is then cleaned off using the developer solution and the laser.

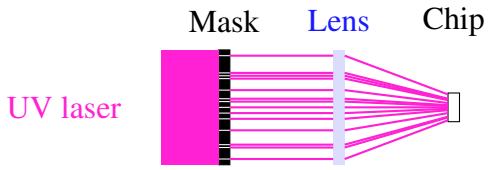


Figure 1.10: Lens system enabling nano-scale etching with a larger mask.

This technique enables the mask to be much larger that the resulting chip, as a system of lenses can shrink the mask image.

1.1.3 Wires, Heat, and the Speed of Light

When creating a circuit, different transistors are assembled and linked together with wires made of a conducting material, like aluminium. A wire is used to transfer information, and has two states: *on* (high voltage) or *off* (low voltage). An ideal wire can conduct current with no resistance effects.

A real wire, however, has a resistance that cannot be ignored. This resistance results in a small loss of voltage proportional to the length of the wire. This effect

heats the circuit, and voltage at the end of the wire can end up being below the semiconductor threshold of $0.7\,\mathrm{V}$, preventing the information *on* from being transferred. The heat is inevitable, but buffer transistors can be placed along the wire to amplify the voltage. However, these transistors have an activation time, slowing down the information transfer.

Another issue is that the *on*/*off* information cannot travel faster than the speed of light $299\,792\,458\,\mathrm{m\,s^{-1}}$, or $30\,\mathrm{cm}$ every nanosecond.

Recent computers work at a maximum of $6\,\mathrm{GHz}$ (with overclocking), which means that information has about $0.166\,\mathrm{ns}$ to perform a computation before the next one comes in. Light can only travel $5\,\mathrm{cm}$ in a cycle, which is smaller than the size of the computer[2]. It is impossible to move information from one side of the circuit to the other in a single clock period, or cycle.

1.1.4 Fanout

In a circuit, the drain (output) of a transistor is connected to multiple different gates (input) of following transistors. If the output is connected to N gate, then this transistor has a fanout of N.

A small current passes through the transistor when it switches, the transistor can fail if there not enough current, as electrons need to move to change the state of the semiconductor. Due to Kirchhoff's current law, current is divided when the wire is used to connect to different transistors in parallel. This means that the fanout cannot be infinite. The maximum fanout of a transistor is called the drive.

If a circuit requires a large fanout, buffer transistors are used to amplify the signal, as illustrated in the next section.

1.1.5 Clock

The transistor circuit is not built to be used for only one execution of a computation. When the input data changes, the signal propagates through the circuit, and after a certain amount of time the result can be retrieved at the output.

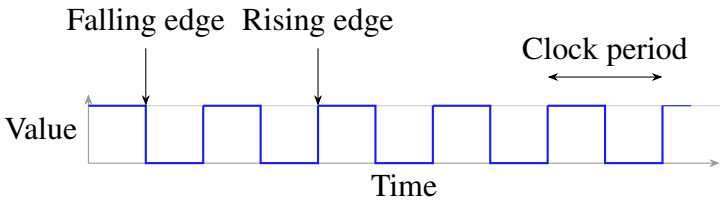


Figure 1.11: Clock signal.

In a synchronous execution model, a ticking clock (Fig. 1.11) keeps the time on the chip. The clock often comes from a quartz oscillator. Every clock period or cycle, new data arrives, and transistors compute using a new value. At the end of the clock cycle, the result is read at the output of the circuit.

[2]CPUs are typically about $1\,\mathrm{cm}$ per $1\,\mathrm{cm}$, but a RAM stick is about $6\,\mathrm{cm}$ long. An external hard drive is often bigger.

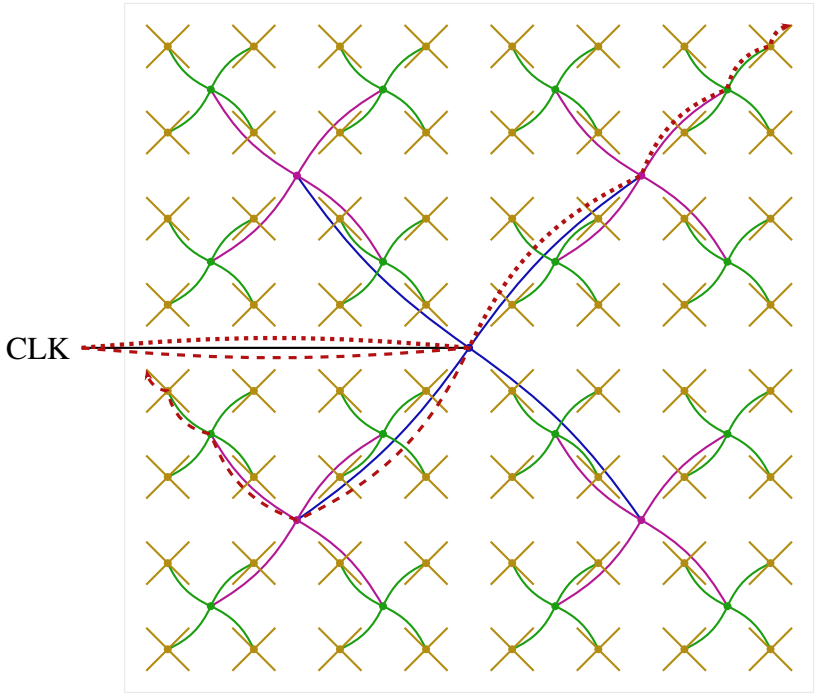


Figure 1.12: Balanced clock tree. The dots are buffers.

The clock, by definition, has a very high fanout, as it connects all over the circuit. To avoid parts of the circuit being desynchronised with others, the clock uses special channels (Fig. 1.12) to get to every part of the chip. The goal is that any component is connected to the clock by approximately the same length of wire, using the same number of buffer transistors along its path. In the example, the leaf of the tree the furthest from the clock (using the dotted path) is as connected the same way as the one nearest from the clock (using the dashed path). Both path are the same length, and are each composed of four buffer transistors.

The clock period is not available in its entirety for computations. Some of it is reserved to absorb uncertainties in the physical implementation of the clock tree.

1.1.6 Moore's Law and Miniaturisation

In 1965, the CEO of Intel, Gordon Moore predicted: *The number of transistors in an integrated circuit will double every year for the next 10 years*. In 1975, this observation was revised to doubling every two years, and became known as *Moore's law*.

This declaration has become a self-fulfilling prophecy, driven by economic mandates demanding exponential growth.

Advances in physics and electrical engineering in the last 60 years have explained the pace described by Moore's law, as transistors are made smaller and smaller. However, miniaturisation comes with problems: the smaller the transistor, the more likely quantum effects, like the tunnel effect, are to occur. By tunnel effect, electrons could jump through the depleted regions of a transistor, which would create a current even when the semiconductor should be isolating.

In recent years, new advancements like 3D construction of transistors have tried

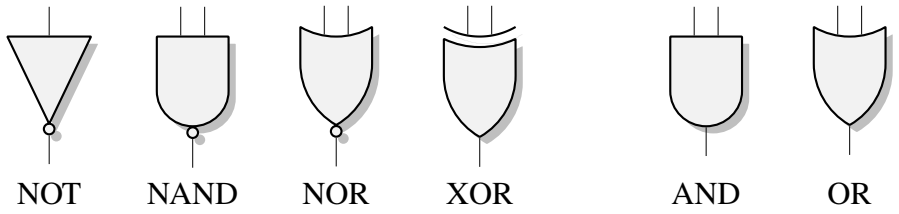
to keep up with the pace. However, the physical limitations have curbed the growth promised by Moore's law, to the point of Hebsen Huang (NVIDIA's CEO) declaring in 2022: *Moore's Law is dead*.

This makes it even more important to design efficient circuits, which is what this thesis is about.

1.2 Circuits

To compute and operate on data, transistors are assembled in large circuits, that make up the computer. Within these circuits, some parts are dedicated to storing data in memory, while others serve as computational units. The computational units operate based on boolean logic.

1.2.1 Logic Gates



(a) CMOS elementary gates. **(b)** Other basic logic gates.

Inputs		NOT	NAND	NOR	XOR	AND	OR
a	b	$\neg a$	$\neg(a \wedge b)$	$\neg(a \vee b)$	$a \oplus b$	$a \wedge b$	$a \vee b$
0	0	1	0	1	0	0	0
0	1	1	1	0	1	0	1
1	0	0	1	0	1	0	1
1	1	0	0	0	0	1	1

(c) Truth table for common logic gates.

Figure 1.13: Basic logic gates and CMOS elementary gates, with their truth table.

Boolean logic is operated on using logic gates (Fig. 1.13). These gates are made of an arrangement of transistors, and used to execute a basic operation on a signal. Any low level computation can be derived from a combination of those gates. Truth tables (Fig. 1.13c) are used to describe the behaviour of a logic gate. Common symbols in boolean logic include $\neg$ for *negation*, $\wedge$ for *and*, $\vee$ for *or*, $\top$ for *true* and $\bot$ for *false*.

As an example, the NAND gate (a NAND $b = \neg(a \wedge b)$) is made with 4 transistors (Fig. 1.14), two p-channel transistors in parallel, and two n-channel transistors connected in series. A NOR gate (Fig. 1.8) is similarly built with 4 transistors.

In CMOS, the NOT, NAND, NOR, XOR gates are the elementary gates, and all the other gates can be derived from them. An AND gate is made by assembling a NAND gate and following that by an inverter.

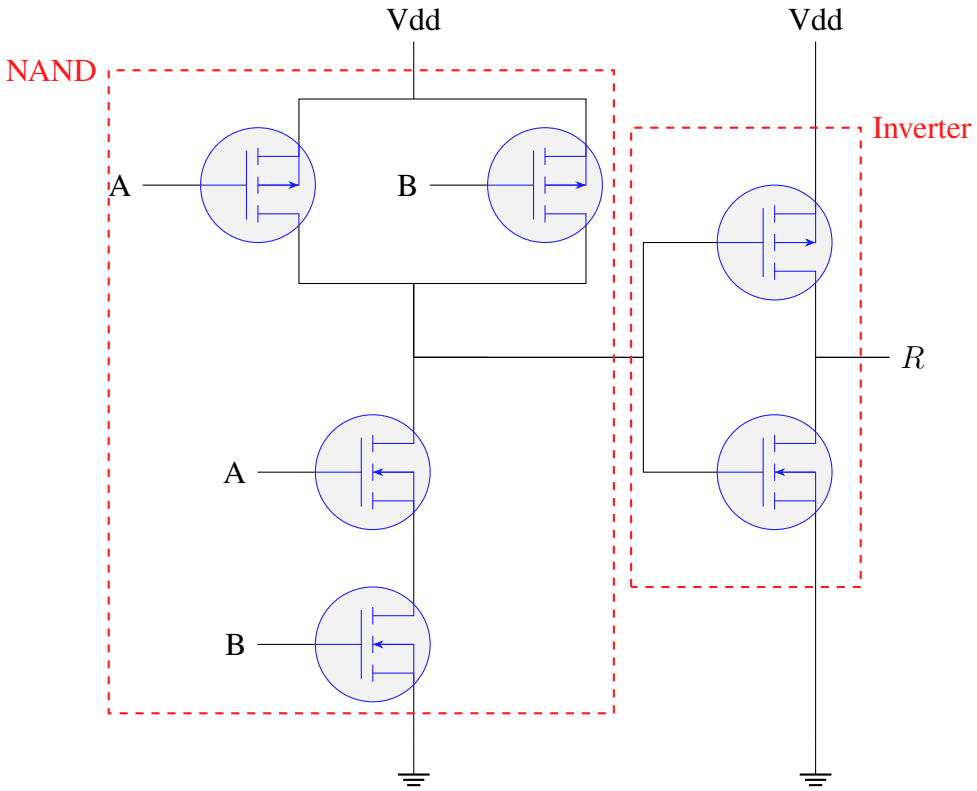


Figure 1.14: An AND gate built with CMOS transistors.

An abstraction that can capture all of the possible gates is the truth table. This can be used to create reconfigurable circuits (see Sec. 1.2.6).

1.2.2 Assembling Gates into Operations

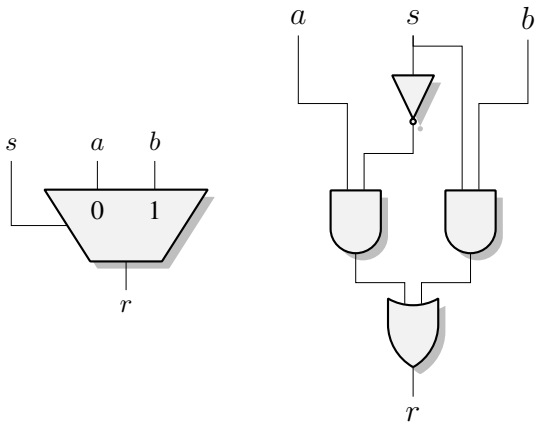


Figure 1.15: Multiplexer and how it is made in gates.

The multiplexer (Fig. 1.15) is a gate that can perform a select operation. If the select signal s is 0, then a is chosen for the result r, otherwise b is chosen. This higher complexity gate can be constructed using the previous low level gates : $r = ((\neg s) \wedge a) \vee (s \wedge b)$

The multiplexer is crucial to construct large operation as it enables to create a branching, similar to software *if-then-else* statements. The inputs a end b of the multiplexer come from two sub-circuits where the formula corresponding to the *then* statement is evaluated and the other for the *else* statement. The select bit s, the result of the *if* condition, chooses the correct result. The implementation of a branch in hardware can be very expensive as both *then* and *else* are pre-computed, and it cannot be optimised the same way software branching is.

1.2.3 Pipelining

For a circuit performing a computation, new data to be processed arrives every cycle. To compute with this new data, transistors must change state, wires must carry information, and this process takes time. When processing data, some parts of the computation are done in parallel, and take more time than others. The longest possible path from the input to the output is called the *critical path*.

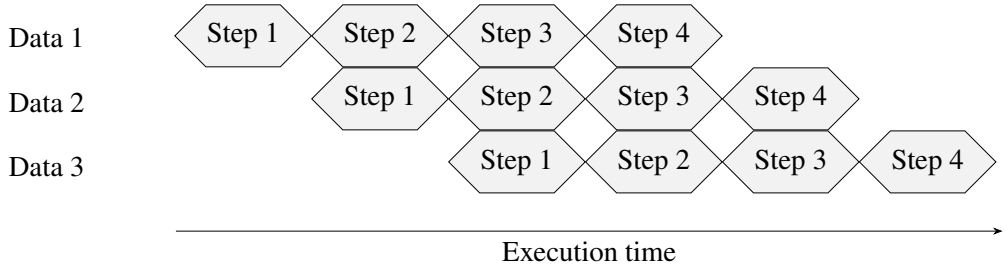


Figure 1.16: Pipelined execution.

If the critical path is larger than the clock period, then the computation must be divided into multiple steps (Fig. 1.16). When the first computational data is finished with the first step, it enters the second step, and the first step is free to accept new data. This is called pipelining, and enables large operations to have a throughput (number of computation carried out) of one computation per cycle, even if the latency (time taken for a computation) of each individual computation is more than a clock period.

1.2.4 Registers and Memories

In order to set up pipelining, it is necessary to save the computation status at the end of every step, to free it for the next computation. Therefore, there is a need for temporary storage between pipeline stages. Registers (Fig. 1.17) are a type of memory that enable to save data for a clock period. Data D is stored at the rising edge of the clock, until the next rising edge, in other words for a full clock period.

Registers are a type of memory called volatile memory, as the information is lost when the circuit is unpowered.

Registers can also be used to store data for more than one cycle, using an *enable* signal e to signal to the register that it should accept the new value d.

Those registers can be used for addressable memories (Fig. 1.18). The address A, made of w_A wires, is used to find which stored data D, made of w_D wires, must

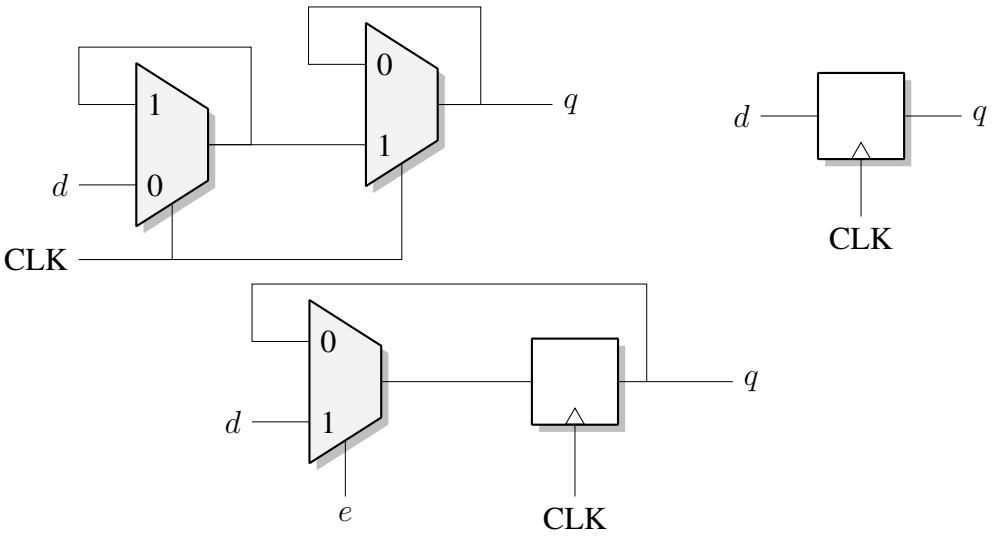


Figure 1.17: Logic of register (left) and its symbol (right). Register with an enable signal for storage (bottom).

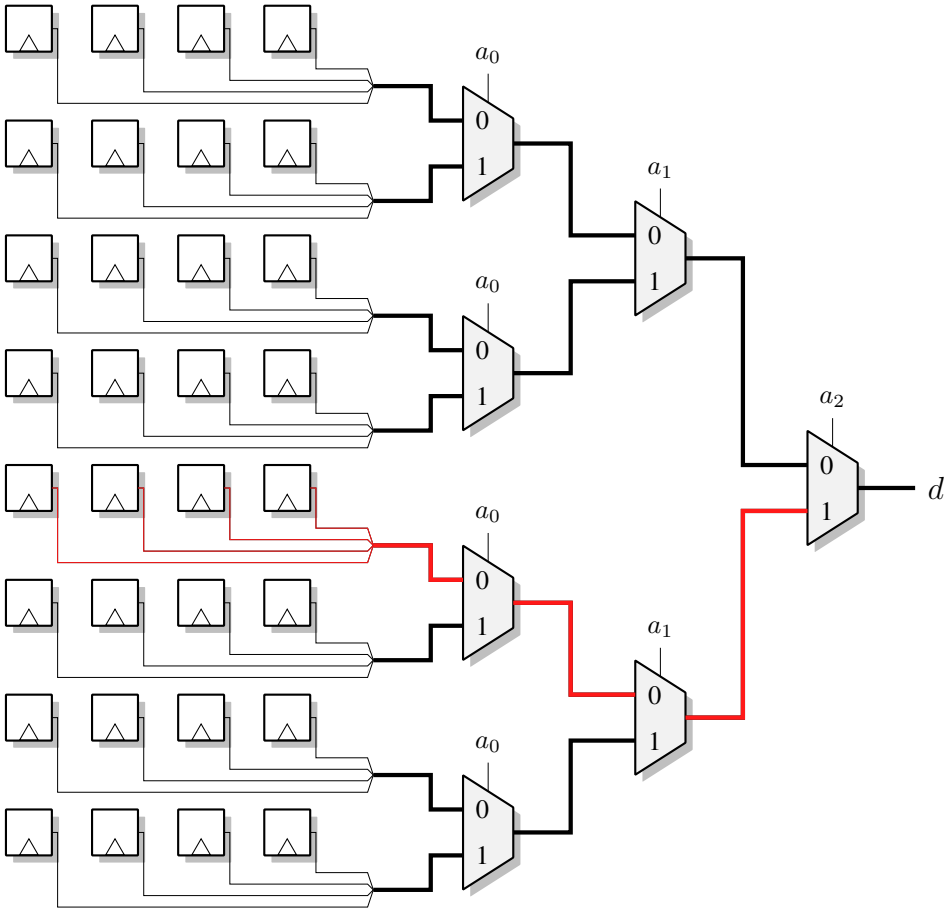


Figure 1.18: Reading an addressable memory, with $w_A = 3$ and $w_D = 4$.

be output. The figure shows an example of value with address 100_2 being chosen. The total number of registers in this addressable memory is $2^{w_A} \times w_D$.

When writing in such memory, the input value is connected to all the stored values. The address is decoded, and computes the enable signals e such that the given address has $e = 1$, and the other ones have $e = 0$, effectively writing only in the registers at the given address.

This type of memory is often used for working memory, for example in *register files*.

1.2.5 Storage

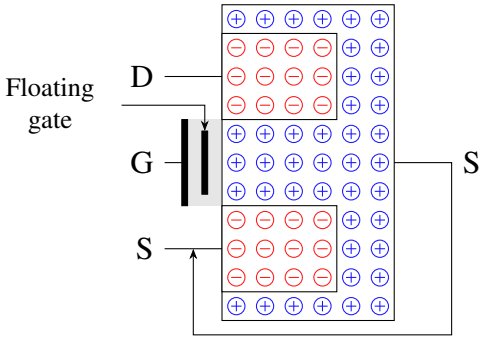


Figure 1.19: Structure of a floating-gate transistor used for non-volatile storage.

Non-volatile memory enables to store data without needing to power the circuit. Flash memory, one of the most common storage for memory cards, USB drives, and SSD disks, are made of a type of non-volatile memory. Data is stored in a special type of transistor called floating-gate transistor (Fig. 1.19), where a second gate is encased in the oxide, between the substrate and the gate. The floating gate can store an electric charge without being powered as it is is isolated by the oxide.

Volatile memory is generally fast but expensive, and non-volatile memory is cheap, but very slow. This is why most operational memory is fast, but cannot store too much data, and large data storage is slow to access. Larger storage, regardless of its type, also take space on the chip, and thus are further away from the computation circuits, taking more time to be accessed. In addition, the larger the storage, the more levels of multiplexers there are to read the data, slowing the access further. Often, data from the storage memory is partially copied into varying degrees of closer volatile memory, in order for it to be used faster for repeated access.

1.2.6 Implementation of circuits: VLSI and FPGA

Transistors are complicated components that can be constructed in many various ways. Assembling them into logic gates and circuits requires a level of care and electrical engineering knowledge that is not expected of hardware designers. When constructing a circuit, a designer has access to a library of logic gates that were constructed and optimised by hand. Various methods are offered to construct a circuit using those optimised gates.

Very-Large-Scale Integration (VLSI)

The VLSI technology for integrated circuits enables companies to design circuits using a large amount of optimised basic gates, and have them manufactured in a specialised factory. Those gates are called *Standard Cells*. The VLSI manufacturer offers all the standard gates AND, OR, NOT, with 2, 3, even 4 inputs, but also more complex ones like an optimised multiplexer cell with 4 inputs and 2 select.

Some gates offer sequential operations, like AND-OR-INVERT (AOI222) that implements: $\neg((A_1 \wedge A_2) \vee (B_1 \wedge B_2) \vee (C_1 \wedge C_2))$.

Any of those complex standard cells could be be broken down into multiple smaller standard cells depending on the situation. Large cells tend to be smaller in area but slower than if the cell was broken down.

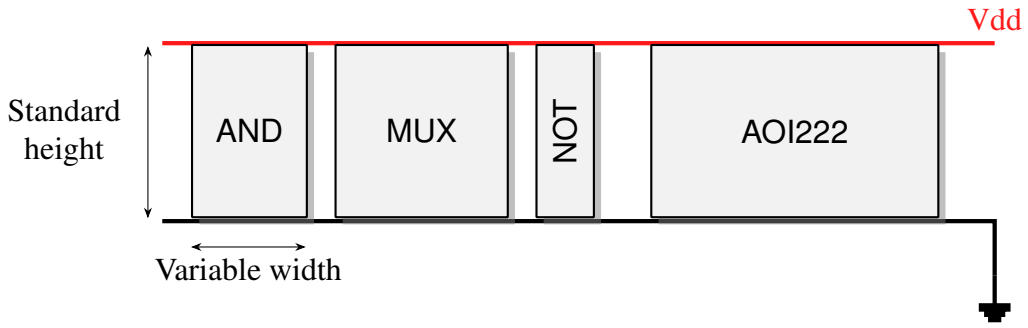


Figure 1.20: Row of standard cells.

Those cells each have a standard height, making them easy to assemble in a line. The ground and the power wire are two parallel wires that are at a fixed position. The routing of computing wires is made on dedicated routing layers, who are above and below the semiconductor layers.

In VLSI, the area metric used to evaluate a circuit refers to the surface used to fit the circuit, including the gaps between rows of standard cells, and the gaps between standards cells of the same row.

Standard cells are generally not completely packed together, filling about 70% of the row. The gaps of inactive semiconductor sitting between the etched standard cells helps heat dissipation.

In some cases, some of those gaps can be used to place the circuit for a rarely used computation. Since it is inactive most of the time, it will dissipate heat for the nearby frequently used circuits. However, the computation will greatly save power when it is used instead of using software techniques, with common computations being a division, or trigonometric function approximations. This paradigm is called *dark silicon*.

In this thesis, the VLSI target include the $16\,\mathrm{nm}$ and the $4\,\mathrm{nm}$ technology node manufactured by Taiwan Semiconductor Manufacturing Company (TSMC).

Field-Programmable Gate Array (FPGA)

Another option is the FPGA (Fig. 1.21), where the basis of the circuit are reprogrammable Look Up Tables (LUTs), the universal basic gate.

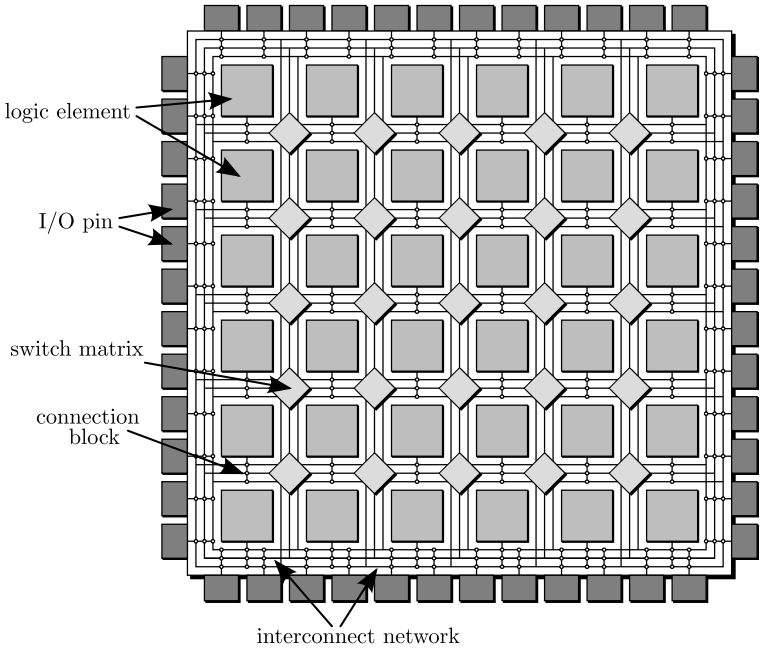


Figure 1.21: Simplified FPGA structure (Figure from [34]).

A Look Up Table has one output and multiple inputs, the current standard being 5 or 6 inputs. The LUTs are implemented with addressable memory, that is filled with the truth table of the desired operation.

A logic gate can be expressed as a LUT using its truth table, and a LUT can be expressed as a boolean formula as well, and simplified using distributivity laws and other common formulas: $\forall a, a \vee a = a \wedge a = a, a \wedge \neg a = \bot, a \vee \neg a = \top, a \vee \bot = a, a \vee \top = \top, a \wedge \bot = \bot, a \wedge \top = a, \ldots$

Routing between the LUTs is made with wires which have reprogrammable intersections, called the switch matrix. All the LUTs have a register just after their output, as a register cannot be made using LUTs.

Modern FPGAs contain acceleration components to help with common operations that would use many LUTs: Block Random Access Memory (BRAM) to store data, and Digital Signal Processing (DSP) blocks that implement a VLSI multiply and add operation. Area for FPGA is counted in number of LUTs, BRAMs and DSPs.

FPGAs are a lower cost alternative to making hardware, as one does not need to prepare the whole manufacturing process of making masks and buying standard cell libraries. If only a couple of those circuits need to be made, it is more cost effective to buy a reprogrammable device. Testing is easier, and any bug with the circuit can be corrected just by reprogramming the FPGA. However, re-programmability comes at the cost of speed, as signals must pass through different switches when routing. The maximal size of the device is also fixed, if the circuit does not fit on the target FPGA, a larger one must be used.

This distinctive architecture comes with various trade-offs compared to VLSI. First, since every LUT is followed by a register, pipelining is a free operation, as no registers must be added or routed to. Then, due to the nature of the universal gate, circuits based on tables are quite cheap compared to other computations. FPGAs

have also been optimised to implement a fast addition operation, as it contains fast carry systems (see Sec. 4.1) between LUTs that do not use the slow reconfigurable routing system.

1.3 Computers

Computers can be implemented either with the VLSI or FPGA technology. The architecture of a computer refers to the way its overall structure is constructed.

1.3.1 Von Neumann Architecture

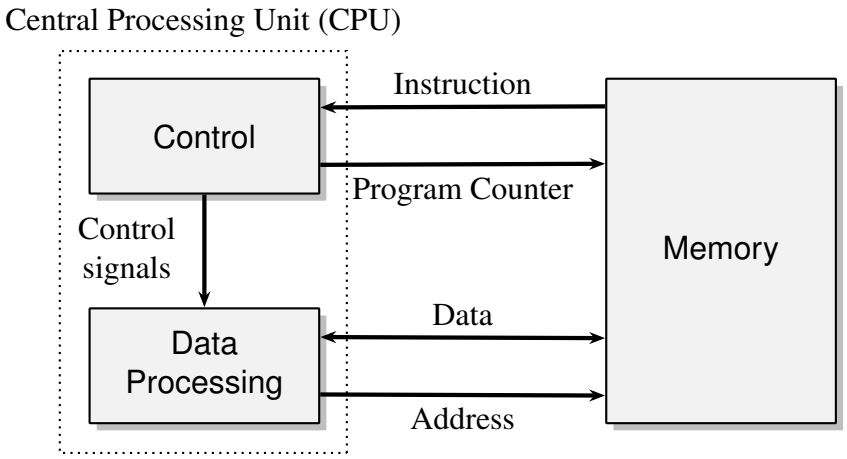


Figure 1.22: Von Neumann architectures for a computer.

Most computers follow the Von Neumann architecture (Fig. 1.22). They are made of multiple components.

First, a memory is needed to store both the program and the data this program operates on.

A control unit is in charge of reading and decoding the program. It keeps track of the address of the program called the Program Counter (PC), and reads in the memory the Instructions (I). Once the instruction is decoded, control signals are sent to the data processing units.

Data processing units contain a little bit of working memory called the *register files*, which are used when breaking down computations in multiple operations. Register files are one type of closer working memory used to reduce the time spent accessing the memory.

A Load and Store Unit (LSU) uses the control signals to load Data (D) into registers from a specific Address (A), or store them into the memory. Caches are often used to further reduce the memory access time, however it is a system that is hidden to the Von Neumann computer. It works by copying storing some often accessed data a bit closer to the processing units, keeping the first access slow but

the subsequent access much faster. If a data was not accessed in a while[3], the data is copied into the far away memory to free the space for new data.

The computations are carried out in the Arithmetic and Logic Unit (ALU), using data from the registers. Often, the ALU contains a specialised Floating-Point Unit (FPU).

Other notable organisations include Graphic Processing Unit (GPU), that allow computations to be carried out in a very parallel way, using the Single Instruction Multiple Data (SIMD) principle.

1.3.2 Kalray MPPA

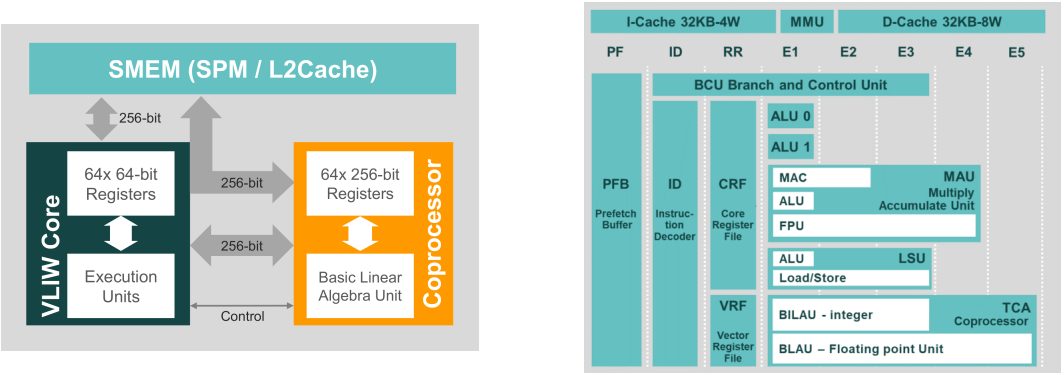


Figure 1.23: Kalray Processing Element (left) and its pipeline detailing the various data processing units (right). Figures from Kalray documentation.

The circuits developed during this thesis are to be included in Kalray's Massively Parallel Processor Array (MPPA), using a VLSI technology. The MPPA's base Processing Element (PE) follows the Von Neumann architecture (Fig. 1.23). The data processing is executed with two ALUs, a LSU, a Multiply Accumulate Unit (MAU) containing also an FPU, and the coprocessor. The Coprocessor is also called a Tightly Coupled Accelerator (TCA), and operates on dedicated register files.

All those units can function at once, and use the Very Long Instruction Word (VLIW) architecture. This architecture puts much of the pressure of parallel execution on the compiler, when the program is converted into instructions. The compiler translate higher level code into instructions that are bundled together if they use different units, allowing parallel execution.

Algorithm 1 Example assembly code for the MPPA.

```
load in r1 from address in r0 //an instruction
; // end of bundle
load in r3 from address in r4 //instruction using LSU
add r1 and r2 in r1 //instruction using ALU
; //end of bundle
```

[3]The definition of "a while" when talking about caches is an active field of research, and out of scope for this thesis.

Pseudocode (Algo. 1) for Kalray's MPPA uses a line skip to separate the different instructions of a single step, and a semicolon to indicate the end of the instruction word.

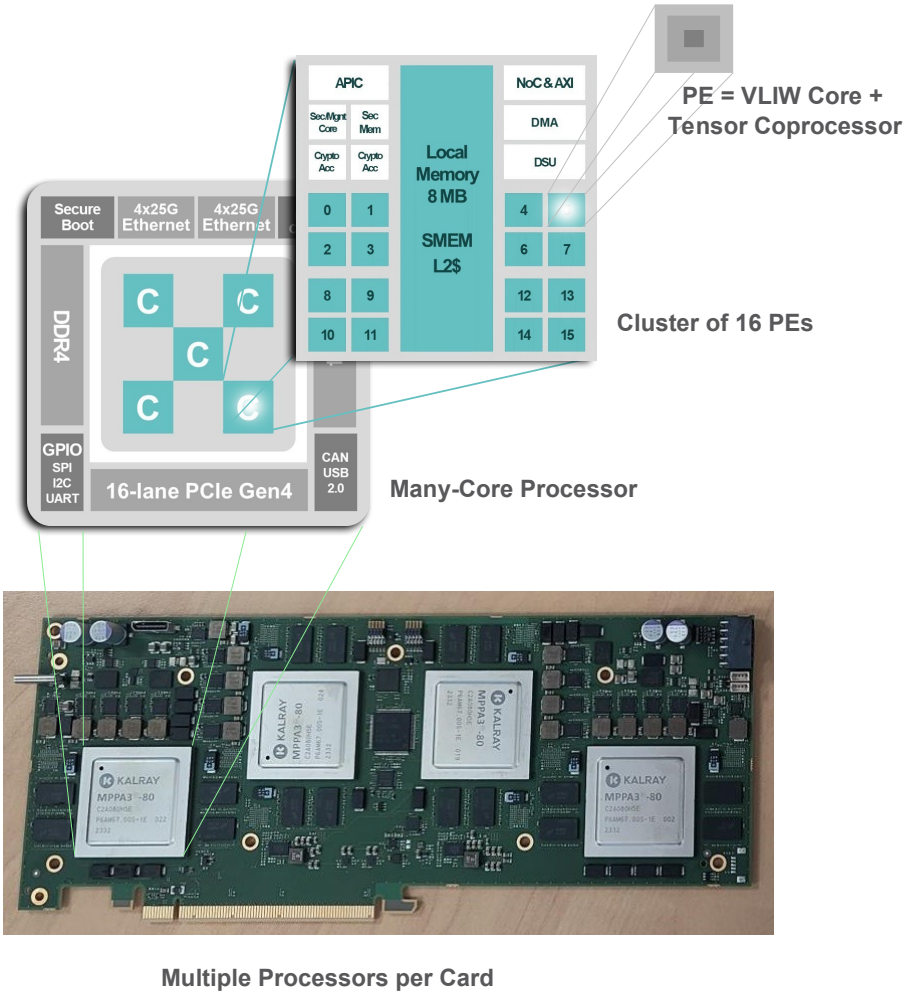


Figure 1.24: Organisation of Kalray's MPPA3 Coolidge Version 2. Figure from Kalray documentation.

While this is already pretty parallel, the Massively Parallel part of the MPPA (Fig. 1.24) comes from how those PEs are assembled. Sixteen PEs are grouped together into a cluster, in which they share a local memory called the SMEM, that functions as a cache. Then, each MPPA contains five of such clusters (so 80 PEs). On top of that, acceleration cards can contain multiple MPPAs, with the pictured card featuring a total of 320 PEs.

1.4 Tools Aiding the Design of Circuits

Multiple tools are used for the steps transforming a description of the architecture to the blueprint of the transistor circuit.

1.4.1 Synthesis place and route

Synthesis is process in which a circuit written with gates is transformed into a hardware blueprint of standard cells (if VLSI is the target) or of LUTs (if FPGA is the target) that has the same logic behaviours as the input circuit.

Once a blueprint of the circuit using the target basic bricks exists, place and route will arrange those bricks in a way that makes it possible to link the cells with the routing wires.

In FPGAs, this means that the synthesis chooses what will be implemented in the truth tables and how they should be linked, and place and route chooses which LUT contains which table, and how to space them to not overwhelm the switch matrices that enable routing.

In VLSI, synthesis chooses the standard cells and how they should be linked, and place and route will pack them into rows of standard cells, placing them to facilitate the routing step, and then using the routing layer to link the standard cells.

In this thesis, the synthesis tool used for VLSI is *Synopsys Design Compiler NXT*, and the tool used for FPGA is *Vivado*.

The level of abstraction of the input code given to the synthesis tool is called register-transfer level (RTL), in which basic gates are used and the pipelining is explicit. Common RTL languages are *VHDL* and *Verilog*.

RTL languages are still pretty low level, describing binary logic. Some tools like High-Level Synthesis (HLS) start from higher level C++ code, however this tends to give less control on how exactly computations are organised and carried out, and heavily rely on proprietary libraries.

Those processes respect the physical limitations of the set target: the period of the cycle, the area dedicated to the circuit, the power it is able to dissipate (by spacing standard cells or LUTs). However, sometimes the designer requests an impossible configuration, and understanding the electronic circuit helps to understand why the synthesis rejects it, and how to solve the issue.

1.4.2 FloPoCo

FloPoCo presents itself[4] as a generator of arithmetic cores (Floating-Point Cores, but not only) for FPGAs (but not only).

FloPoCo is the software used in this thesis used to help write RTL code. It is written in C++ and generates VHDL code.

Operator Generator

There are two levels of FloPoCo usages. The first is as a library for arithmetic operators, that can be fully parametrised to the specific need of the user. In this case, the user can go through the list of available operators, choose the parameters that are needed for their application, both pertaining to the operator and the target FPGA and frequency. The user can generate the fully pipelined operator, as well as a Test Bench checking that it is functional, optimised for their specific target. Generating operators can be done without looking at any of the FloPoCo code, only the documentation.

[4] https://www.flopoco.org/

FloPoCo offers various operators, Floating Point addition, multiplication and division, Floating Point and Fixed Point function approximation with multiple methods, and more. The *DAG Operator*[5] can be used to combine different FloPoCo operators into a larger one by using a configuration file.

This however can be limiting, especially if small transformations need to be applied on intermediate signals.

Custom Operators

When the need for this extra modularity arises, it becomes necessary to create a custom FloPoCo operator. FloPoCo also implements lower level operators to help with a custom operator, like *Shifter*, *LZOC* (Leading Zero and/or One Count), *Sorting Network* and probably the most useful *Bit Heap*. The Bit Heap operator computes the optimal way (either by Integer Linear Programming, or with heuristics if optimality is not necessary) to add multiple integers by compressing the bits before performing the final addition.

Between those existing operators, signals can be declared and operations can be performed on signals using the VHDL syntax. When the signal is are declared, it is necessary to define a few parameters : the name of the signal, if it is a wire or a bus, how wide is the bus, and the delay of the signal. The delay describes the time needed for this signal to be computed when all the operands are ready. This is needed for the automatic pipelining.

FloPoCo supports multiple FPGA targets, which all have different delay parameters. This is why most of the delay time is abstracted into various helper functions, so that the operators are correctly pipelined even when technology changes.

This is how FloPoCo is used in this thesis.

FloPoCo for VLSI

In this thesis, FloPoCo is used for custom operators, and required to generate code for VLSI. Multiple small modifications to the framework have been implemented to add a VLSI target to change the way operators are pipelined, and by adding new naming features (detailed in Sec. A).

[5] https://www.flopoco.org/DAGOperator/

Chapter 2

General Purpose Number Formats

[...] we[1] *share the same feeling that the big improvements brought to numerical computing by the IEEE-754 and 854 standards for floating-point arithmetic are endangered: we must explain to computer architects, compiler designers and numerical application programmers that some features of the standards that sometimes seem arcane or that seem to hinder performance may be crucial when reliability and/or portability are at stake.*

– Jean-Michel Muller

Data in computers is encoded using voltage levels in wires or memory cells. These voltage levels correspond to the binary values of either 0 or 1, which form the building blocks of information representation in digital systems.

However, when dealing with more complex or precise data, binary values alone are insufficient. To expand the quantity of encoded information, *b*inary dig*its* (bits) are grouped together to form a bit vector. A bit vector B, identified by its width w, is an array of bits b_i such that $B = b_{w-1} \dots b_2 b_1 b_0$. The width of a bit vector is often a power of two, the most common being 32 and 64, but sometimes as small as 4 or as large as 256.

[1] Referencing Professor Kahan.

The interpretation of a bit vector, the data it encodes, depends on agreed-upon standards. The encoding of data significantly impacts the ways it can be processed during computations. This chapter describes some of the most common ways of interpreting a bit vector and the data it encodes : natural numbers, integers, fixed-point numbers, and floating-point numbers. It focuses on the number formats, and how to encode real numbers into those formats. The way to operate on those formats is described in later chapters.

2.1 Natural Numbers

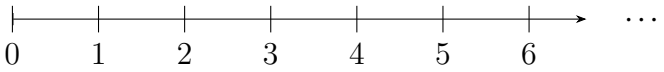


Figure 2.1: Natural number line.

A natural approach is the positional encoding, where the bit vector encodes a number as written binary: the bit vector B of width w is interpreted as $\sum_0^{w-1} 2^i b_i$. Each bit b_i has a position i and a weight 2^i. This enables to encode natural numbers in $\mathbb{N}$ (Fig. 2.1).

(a) Representation of an 8 bit natural number. **(b)** Representation of $146 = 2^7 + 2^4 + 2^1$.

Figure 2.2: Graphical representation of an 8 bit natural number (left) and an example encoding 146 (right).

Sadly, while $\mathbb{N}$ is infinite, the computer is not, and the range of the numbers n that can be encoded is limited by the width of the bit vector used, $n \in [\![0, 2^w - 1]\!]$, (Fig. 2.2). In this example $w = 8$ enables all numbers between 0 and $2^8 - 1 = 255$ to be encoded.

The bit on the far right is the Least Significant Bit (LSB), for natural numbers it is in position 0. The bit on the far left is the Most Significant Bit (MSB), in this example it is in position 7.

When encoding 146, the number is decomposed as a sum of powers of two: $146 = 128 + 16 + 2$. The number is then encoded as bits in binary following this decomposition: $146 = 2^7 + 2^4 + 2^1 = 10010010_2$, where the subscript 2 indicates the number is in binary format.

2.2 Integers

There are various ways to encode negative numbers in binary, i. e. numbers in $\mathbb{Z}$ (Fig. 2.3). The ones referenced in the rest of this work are *sign-magnitude* and *two's complement*.

In the **sign-magnitude** encoding for a bit vector size of w (Fig. 2.4), a bit $s = b_{w-1}$ is used to store the sign of the number, and the rest of the bits are used to

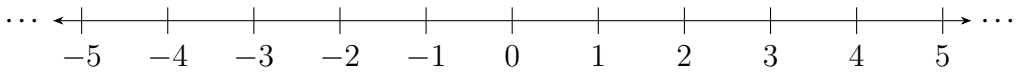
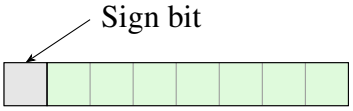


Figure 2.3: Integer number line.



(a) Encoding for an 8-bit integer number.

1	0	0	1	0	0	1	0

(b) Sign-magnitude representation of -18 on 8 bits: encoded as $(-1)^1 \times (2^4 + 2^1)$.

1	1	1	0	1	1	1	0

(c) Two's complement representation of -18 on 8 bits: encoded as $-2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 2^1 = -18$.

1	0	0	0	0	1	0	0	1	0

(d) Sign-magnitude representation of -18 on 10 bits: encoded as $(-1)^1 \times (2^4 + 2^1)$.

1	1	1	1	1	0	1	1	1	0

(e) Two's complement representation of -18 on 10 bits: encoded as $-2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^3 + 2^2 + 2^1 = -18$.

Figure 2.4: Graphical representation of an 8 bit natural number (a) and an example encoding -18 in sign-magnitude (left) and two's complement (right) on 8 and 10 bits.

store the absolute value: $(-1)^s \times \sum_{i=0}^{w-2} b_i \times 2^i$. This method has two representations for 0 (+0 and -0), and can encode numbers in $[\![-(2^{w-1}-1), \ldots, 2^{w-1}-1]\!]$. The same number in a larger representation has a sign bit moved over to the new MSB, and the gaps are filled with zeros.

In the **two's complement** encoding, numbers are considered cyclic as if on $\mathbb{Z}/2^w\mathbb{Z}$. This method has one representation for 0 and can encode numbers in $[\![-2^{w-1}, \ldots, 2^{w-1}-1]\!]$. A positive number $z \in [\![0, \ldots, 2^{w-1}-1]\!]$ is encoded like a natural number. A negative number $z \in [\![-2^{w-1}, \ldots, -1]\!]$ is interpreted such that $z = -2^{w-1} + \sum_{i=0}^{w-2} b_i \times 2^i$. The same number in a larger representation is extended from the most significant bits with copies of the sign bit. This is called *sign-extension* during conversions to a larger format.

In most programming languages, integer formats are: `char` ($w = 8$), `short` ($w = 16$), `int` ($w = 32$) and `long` ($w = 64$). When dealing with natural numbers, programming languages use the `unsigned` keyword in front of the format, for example `unsigned int` for a natural number where $w = 32$.

2.3 Fixed-point Numbers

When more precise fractional numbers are needed, a scaling factor can be added to our integer. The encoding is similar to integers, but the number X is scaled by a fixed power of two, represented as $X \times 2^{\text{LSB}}$ where the Least Significant Bit (LSB) is a parameter of the format. The format is denoted as $\text{uFix}(\text{MSB}, \text{LSB})$ if it is unsigned,

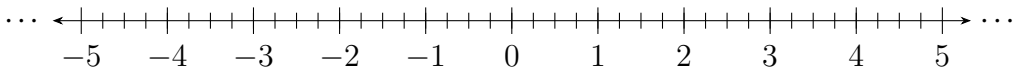
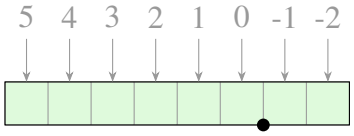
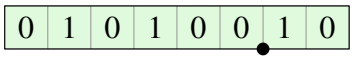


Figure 2.5: Fixed-point format sFix$(5, -2)$ number line.

sFix(MSB, LSB) if it is signed.



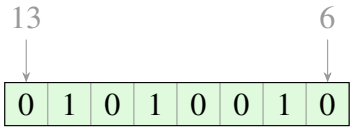
(a) Representation of a fixed-point format uFix$(5, -2)$.



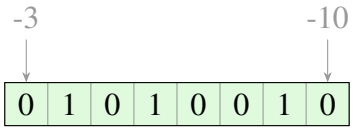
(b) Representation of 20.5 on a fixed-point format uFix$(5, -2)$.

Figure 2.6: Graphical representation of an 8 bit fixed-point number (left) and an example encoding 20.5 (right).

Integers are also a special case of fixed-point format, the examples in the previous sections would be the format uFix$(7, 0)$ and sFix$(7, 0)$.



(a) Representation of 5248 on a fixed-point format uFix$(13, 6)$.



(b) Representation of 0.080078125 on a fixed-point format uFix$(-3, -10)$.

Figure 2.7: Various example of graphical representation of fixed-point numbers where the point is not represented.

Lets take for example the fixed-point format uFix$(5, -2)$ (Fig. 2.5). This format can encode numbers that are an integer multiple of 2^{-2} (Fig. 2.5). When representing fixed-point formats, it is usual to add the point (Fig. 2.6a) to show where the position 0 is. This is useful in the case where the position of the bits are not shown (Fig. 2.7). The fixed-point format can also be used to encode very big (Fig. 2.7a) or very small numbers (Fig. 2.7b), in which case the point is not always represented.

Most programming languages do not have a native type for fixed-point numbers. Often, dedicated libraries are used, in which the integer and fractional part of the fixed-point number are encoded separately as two integers.

2.4 Floating-point Numbers

While fixed-point numbers allow a fixed scaling for a given format, a dynamic scaling gives more flexibility to the number format (Fig 2.8). This will enable the approximation of real numbers in $\mathbb{R}$.

This encoding is similar to the scientific representation (Fig. 2.9). It follows the structure: Sign $\times$ Significand $\times$ Base$^{\text{Exponent}}$. While base 10 can exist in computers

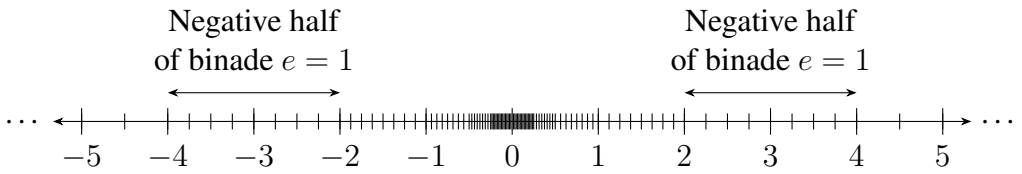


Figure 2.8: Floating-point $\mathbb{F}(4, 3)$ number line.

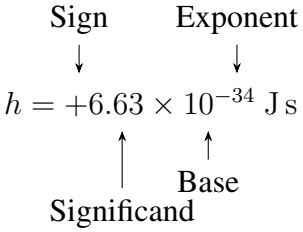


Figure 2.9: Scientific notation of the Planck constant.

for banking software, base 2 is the most commonly used in digital systems. This encoding has been standardised in 1985 [66] as the IEEE 754 Standard.

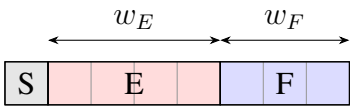


Figure 2.10: Representation of floating-point $\mathbb{F}(4, 3)$.

A floating-point format (Fig. 2.10) is made of three fields S, E, F. $\mathbb{F}(w_E, w_F)$ denotes the set of floating-point numbers parametrised by the width w_E of the exponent E, and the width w_F of the fraction F. $\mathbb{F}(w_E, w_F)$ is also used, by abuse of notation, to describe the floating-point format. All the examples in this section use the format $\mathbb{F}(4, 3)$ to illustrate.

For unity of representation, the significand is constrained to have exactly one non-zero digit in its integer part. This ensures that the exponent describes the order of magnitude of the number. The significand always start with 1, so this bit can be omitted, and only the bits after the point, the fraction $F \in \text{uFix}(-1, -w_F)$, are encoded. The significand is reconstructed as $1.F = 1 + F$. The size of the fraction field determines the precision of the format.

The exponent is an integer, and should be able to have both positive and negative value, as to encode both very large and very small numbers. The choice taken by the standard is not to use a two's complement or sign magnitude for the exponent, but a bias approached. E, the biased exponent, is an unsigned integer which represents the signed number $e = E - b, b = 2^{w_E - 1} - 1$, the unbiased exponent. This choice enables two floating-point numbers to be compared like two sign-magnitude integers. The size of the exponent field determines the range of the format.

The sign, as with sign-magnitude methods, is encoded with a unique bit. An encoding $(S, E, F) \in \mathbb{F}(w_E, w_F)$ represents the rational:

$$x = (-1)^S \times 2^{E-b} \times (1 + F) \quad . \tag{2.1}$$

The numbers who have the same exponent are called a *binade*.

2.4.1 Special Values

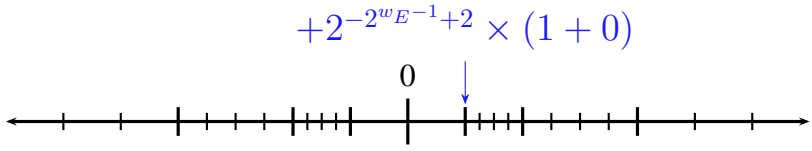
Some encoding space in the floating-point format is reserved for special values.

Infinity is encoded with the fraction equal to 0 and the exponent equal to $2^{w_E} - 1$, the maximum exponent. Both $+\infty$ and $-\infty$ are encoded depending on the sign.

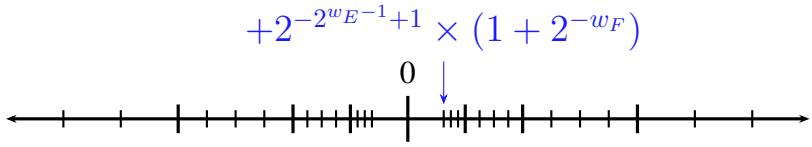
Not a Number (or NaN) encode an error. Any number with an exponent equal to $E = 2^{w_E} - 1$, a non-zero fraction and any sign encodes an NaN. Historically, the large number of NaN values ($2^{w_F+1} - 2$) were reserved to be able to encode error messages into the numbers. In practice[2], only two NaN classes are really used, either a signalling NaN, or a quiet NaN. If the most significant bit (MSB) of the fraction is 1, then the NaN is quiet, otherwise it is signalling.

The whole binade with the maximum exponent $2^{w_E} - 1$ is occupied by infinity and NaN encoding. The maximum representable number has an exponent of $E = 2^{w_E} - 2$, which encodes $e_{\max} = 2^{w_E - 1} - 1$. The value of the maximum positive representable number is $2^{2^{w_E}-2} \times (1 + \frac{2^{w_F}-1}{2^{w_F}})$ (Fig. 2.12a).

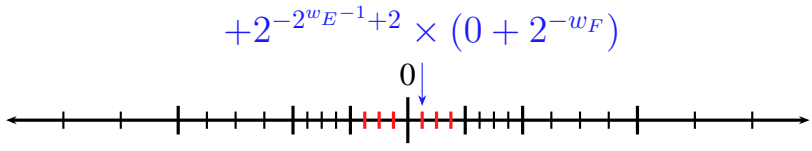
Zero is a special value that cannot be obtained with Eq. 2.1. It is encoded with the exponent and fraction equal to 0. There are two encodings of 0, noted $+0$ and -0.



(a) Zero exponent encodes zero.



(b) Use 0 binade like normal numbers (except for 0 encoding).



(c) Have subnormals.

Figure 2.11: Various ways to use the encoding for $E = 0$.

[2]Except in Javascript, where the large number of NaN is used to encode integers.

Subnormals: The binade of numbers whose exponent is 0 is already limited in size as two encoding are already used to encode ± 0. There are multiple options in how to use the rest of the binade.

The first option is to use the whole binade with 0 exponent as extra encoding for 0 (Fig. 2.11a). This is generally considered a waste of encoding, but is sometime chosen as a cheap solution.

The encoding can also be used following the normal formula (Fig. 2.11b). This is not a common option, as it makes the binade incomplete, and a large gap still remains around 0.

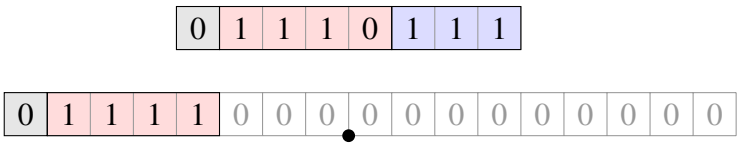
The last option is the use of subnormals (Fig. 2.11c). Subnormals are a special type of floating-point numbers that are not normalised, that is whose significand does not start by 1 (Eq. 2.2).

$$(-1)^S \times 2^{0-b+1} \times (0+F) \quad \text{when } E = 0. \tag{2.2}$$

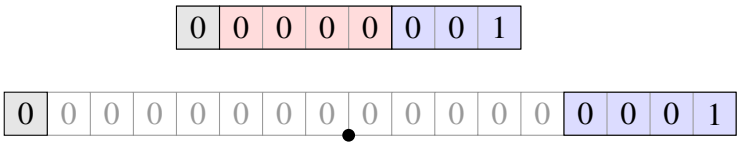
With subnormals, there is the same interval between two floating-point numbers in the binade with exponent 0 and exponent 1, allowing a regularly spaced graduation of numbers in $\left[-\times 2^{-2^{w_E-1}+2}, +\times 2^{-2^{w_E-1}+2}\right]$. This is called gradual underflow. The smallest normal number has an exponent that is encoded with $E = 1$, representing $e_{\text{min\_normal}} = -2^{w_E-1} + 2$.

The largest advantage of this choice is its impact on properties verified by floating-point numbers. The most important one for computer scientists [52] is $x - y \neq 0 \Leftrightarrow x \neq y$, for x, y floating-point numbers. For mathematicians, this would be the Sterbenz Lemma [96]: if $\frac{y}{2} \leq x \leq 2y$, then $x - y$ is also a floating-point number, thus the subtraction is computed exactly.

The smallest representable subnormal number (Fig. 2.12b) is: $+2^{-2^{w_E-1}+2-w_F}$. The weight of this number is $e_{\min} = -2^{w_E-1} + 2 - w_F$.



(a) Largest positive floating-point number in $\mathbb{F}(4,3)$: 120.



(b) Smallest positive floating-point number in $\mathbb{F}(4,3)$: 0.001953125.

Figure 2.12: Representation of the smallest and largest representable number in $\mathbb{F}(4,3)$ and their expansion in the fixed-point encoding $\text{sFix}(8,-9)$.

2.4.2 Expansion to a Fixed-point Encoding

A floating-point format has a finite maximum and minimum exponent. There exists a fixed-point format (very large), in which we can encode all the numbers in $\mathbb{F}(w_E, w_F)$

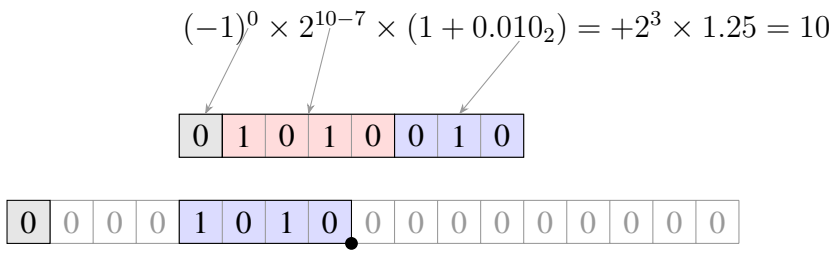


Figure 2.13: Representation of 10 on a floating-point $\mathbb{F}(4, 3)$ and its expansion in the fixed-point encoding sFix$(8, -9)$.

(Fig. 2.13).

This format is sFix$(e_{\max}+1, e_{\min})$, with $e_{\max} = 2^{w_E-1}-1$ and $e_{\min} = -2^{w_E-1}+2-w_F$.

2.4.3 Common Floating-point Formats

The original IEEE 754 standard from 1985 [66] only described two binary floating-point formats, on 32 and 64 bits. A revision in 2008 [67] added three formats, on 16, 128 and 256 bits.

- binary16, FP16 or `half` in most programming languages, is the smallest floating-point format described by the standard, with $w_E = 5, w_F = 10$. It is often used for graphical computing or other computationally intensive applications that do not require much precision and range.
- binary32, or FP32, with $w_E = 8, w_F = 23$, the most versatile and general format. It is called `float` in most programming languages.
- binary64, FP64 or `double`, is a floating-point format commonly used for scientific computing, with $w_E = 11, w_F = 52$. In most computers it is the largest format supported in hardware.
- binary128 or `quad`: $w_E = 15, w_F = 112$, is arguably larger and more precise than what anyone would ever need. The fraction of binary128 ($2^{113} \approx 10^{34}$) is large enough to express the diameter of the observable universe ($\approx 8.8 \times 10^{26}$m) with the precision of about the size of a virus (10^{-7}m). To this, add an exponent in $[10^{-4965}, 10^{4931}]$.
- binary256 or `octuple`: $w_E = 19, w_F = 236$. In case someone wants bigger.

2.5 Computing Uses

When using a computer, most people are not designing code for any of the aforementioned formats, but with real numbers in mind.

First, real numbers must be converted into these formats. Then, those numbers are used in computations and further transformed.

2.5.1 Rounding

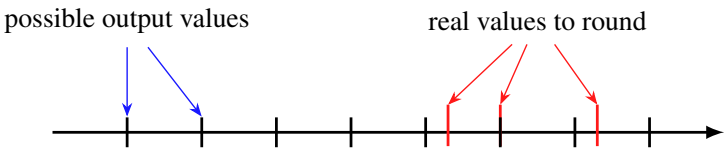


Figure 2.14: Representable number line and real values.

If the real number $x \in \mathbb{R}$ is in the range of the format, then it is either exactly a number the format can represent, or is somewhere between two representable numbers (Fig. 2.14). The rounding operation is often denoted as $\circ(\cdot)$, and $X = \circ(x)$ is the rounded value of x.

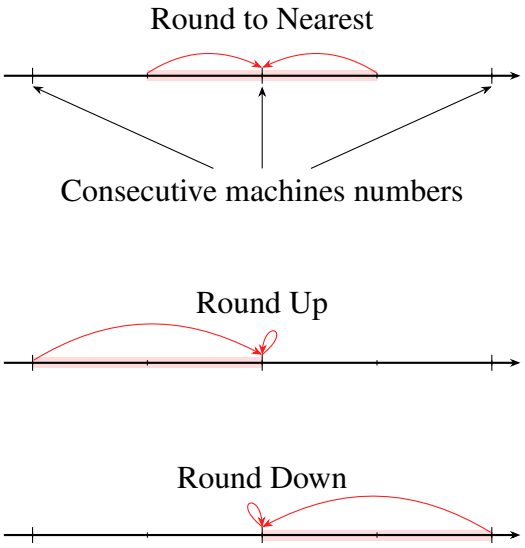


Figure 2.15: Various rounding options for a real number.

There are multiple rounding modes defined in the IEEE 754 standard. The main ones are Round to Nearest (RN), Round Up (RU), Round Down (RD) and Round to Zero (RZ) (Fig. 2.15). In Round Up mode, the number is rounded to X^+, the smallest floating-point larger than x. In Round Down mode, the number is rounded to X^-, the largest floating-point number smaller than x. Round to Zero is equivalent to Round Up when x is negative, and to Round Down when it is positive. x verifies $x \in [X^+, X^-]$, with X^+, X^- two consecutive representable numbers.

Round to Nearest rounds to the nearest floating-point number. In some cases, x is exactly between two floating-point numbers. This situation is called a *tie*, and the number a *mid-point*. The standard defines two different ways to break the tie. The first one, Round to Nearest Even (RNE), rounds x to the number (X^+ or X^-) whose encoding finishes with a 0. It is the most commonly used for binary floating-point. The other tie-breaking rule is Round to Nearest ties Away from zero (RNA). This rounding is used for decimal floating-point numbers, following the common tie rule where x is rounded to the number of larger magnitude, that is $\circ(1.5) = 2$.

A useful rounding mode that is not in the standard is Round to Odd (RO) introduced [5] to avoid issues when numbers are rounded twice into formats with progressively less precision. In Round to Odd, x is rounded to the number X^+ or X^- which encoding finishes with a 1, except if it is exactly representable as an even representation (in which case $X = X^+ = X^-$ is even).

Faithful Rounding

Sometimes, rounding is relaxed, allowing for either X^+ or X^- be used as rounding option for x. This is called Faithful rounding (Fig. 2.16). In this context, Round to Nearest is sometimes also called Correct Rounding.

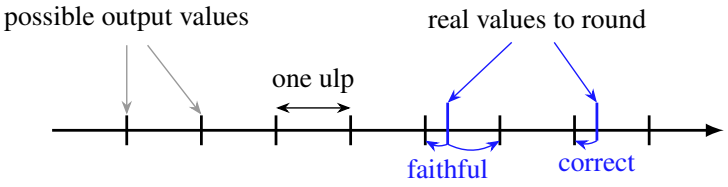


Figure 2.16: Illustration of Round to Nearest and Faithful rounding.

Ulp

An ulp (Unit in the Last Place) can be defined as the distance between two consecutive representable numbers (Fig 2.16). In the case of natural numbers or integers, the ulp is equal to one. In the case of fixed-point numbers, the ulp is 2^{LSB}.

In the case of floating-point numbers, the ulp is used in relation to a specific number, as the ulp is different depending on the exponent. The ulp of a floating-point number is $\text{ulp}(X) = 2^{E-b-w_F}$. The definition of an ulp of a real number, when rounding to a floating-point number, has subtleties when nearing an exponent change. This is discussed in details in [95], with multiple definition options.

In this work, ulp is only used for fixed-point numbers, for which it is well-defined. The ulp function is useful for error evaluation when certifying some rounding properties. For correct rounding, $|\circ(x) - x| \leq \frac{1}{2}\text{ulp}(x)$. For faithful rounding, $|\circ(x) - x| < \text{ulp}(x)$.

2.5.2 Computational Hazards

When executing any operation $\diamond \in \{+, -, \times, \div, \ldots\}$ on floating-point numbers $x, y \in \mathbb{F}(w_E, w_F)$, the IEEE operation is defined as $\text{IEEE}_\diamond = \circ(x \diamond y)$. The result must be as if the operation was computed exactly like a real and then rounded.

If the intermediate result $x \diamond y \notin \mathbb{F}(w_E, w_F)$, then there is a loss of precision during the rounding, which results in the *Inexact* flag being raised.

When a division by 0 is detected, the IEEE 754 *Division by 0* flag is raised, and the computation results in either $+\infty, -\infty$ or NaN (in the case of $\frac{0}{0}$).

Some flags are raised because the number is out of the range of the format. If the result is too big to fit in the format, this is called an *Overflow* of the format,

which raises the corresponding IEEE 754 flag. An *Underflow* is when the number is too small to fit in the format. The IEEE 754 flag is raised when a number looses precision to the range of the format. A number that can be exactly represented as a subnormal does not raise any flags. If a number is rounded to a subnormal with loss of precision, then the inexact and the underflow flags are raised.

The last IEEE 754 flag is the *Invalid* flag. It is raised when an invalid operation is executed: $\frac{0}{0}, +\infty - \infty, \infty \times 0, \sqrt{-2}, \ldots$ In most of those cases, a NaN is returned.

Any operation whose inputs contain a signalling NaN will raise the invalid flag. It will also return a quiet NaN, except in the Max or Min functions who have special functioning (as NaN is neither bigger nor smaller than any other number).

Chapter 3

Accelerating Machine Learning

> Ἐπιστήμη γὰρ οἶμαι δεῖ κρίνεσθαι ἀλλ᾽ οὐ πλήθει τὸ μέλλον καλῶς κριθήσεσθαι.
>
> – Πλάτων

Machine Learning (ML) first gained popularity for computer vision, using Convolutional Neural Networks (CNN). The main applications of the automatic analysing of pictures or video are self-driving cars, video surveillance, waste sorting, and medical imagery. The new revolution in the ML world are Large Language Models (LLM), whose use is now widespread for many applications: translation, chatbots, code generation, ...

We trust that ML researchers know what they are doing, and aim to provide them the tools to power their models.

3.1 How are Large Language Models Powered ?

The effectiveness of Large Language Models (LLMs) has increased significantly in recent years. The primary driver of the technological advancement of LLMs was brought by Transformers [121]. A transformer is a specific type of ML model that is able to generate text and make links between subjects. Its structure is described in Fig. 3.1.

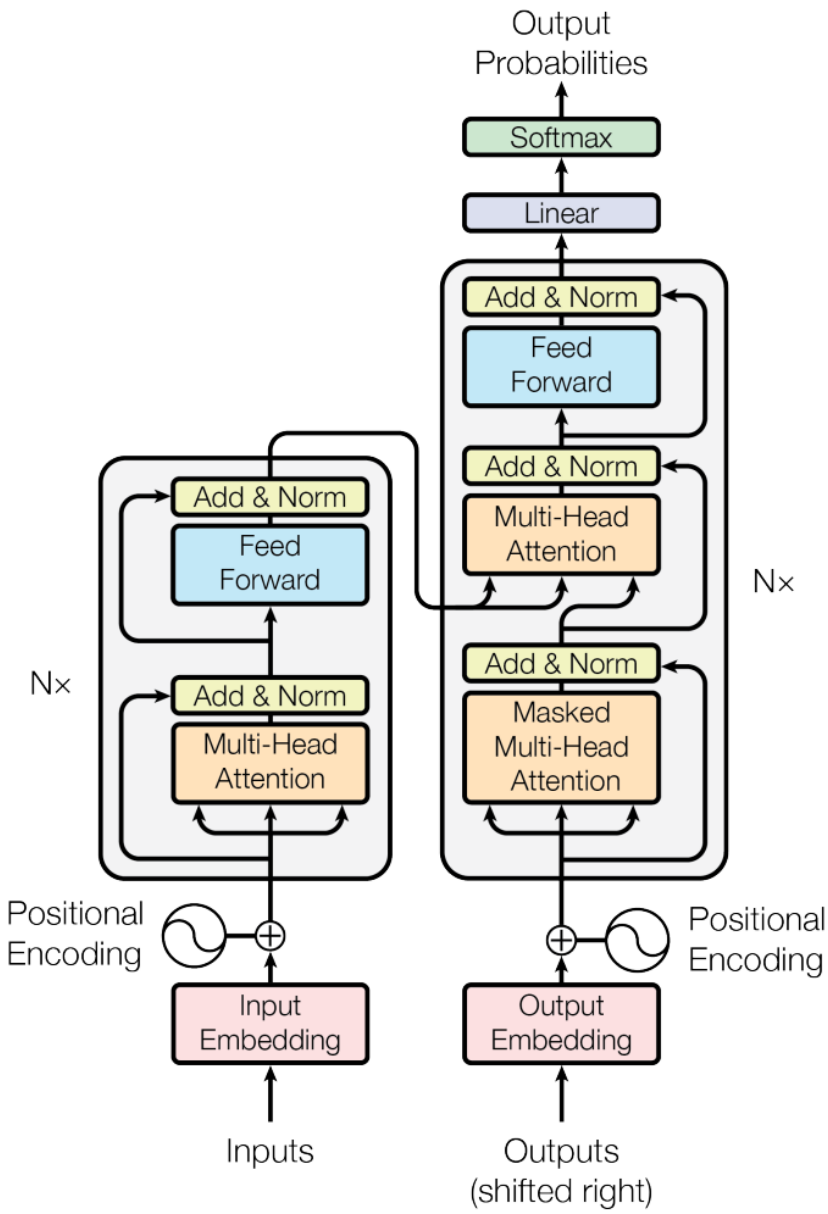


Figure 3.1: Architecture of a transformer network (Figure from [121]).

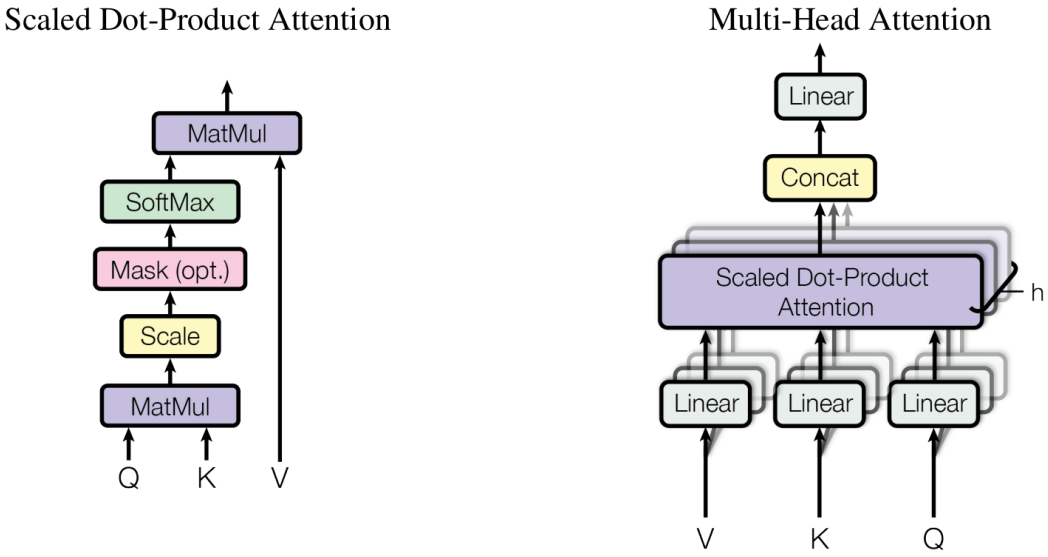


Figure 3.2: Architecture of an attention layer (Figure from [121]).

A large part of machine learning revolves around efficiently moving data between storage and arithmetic units. The computational core of the transformer network is made of multiple attention layers (Fig. 3.2). Three steps in attention mechanism are arithmetic operations: matrix multiplication (described in section 3.1.2), the softmax normalisation layer, and the linear layer that is composed of a ReLU (Rectified Linear Unit) activation function and a linear scaling (described in section 3.1.1).

There are two different phases in the life of a machine learning model, training and inference. First is the training. During this phase the model is used for a specific task, and the quality of the given result compared to the expected result enables to tune its parameters. Once the results are satisfactory, the model can be used for its task, for the phase called inference.

This thesis focuses more on inference.

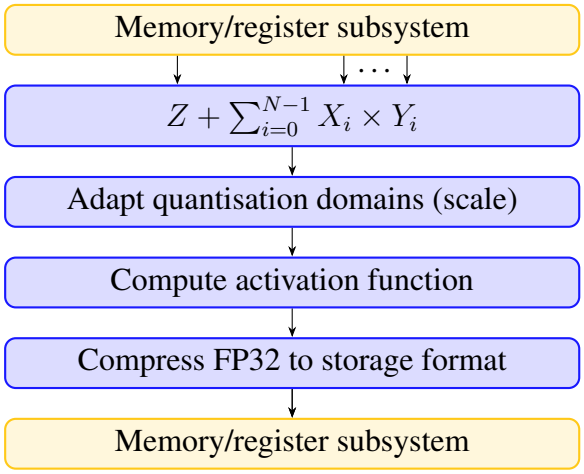


Figure 3.3: Architecture of steps for ML.

Without loss of generality, the different steps in network inference are described in Figure 3.3.

3.1.1 Functions

The softmax function operates on a vector of real numbers $\vec{x} = (x_i)_{0 \leq i < n} \in \mathbb{R}^n$ and outputs a vector $\sigma(\vec{x}) = \vec{y}$ such that:

$$y_i = \frac{e^{x_i}}{\sum_{j=0}^{n-1} e^{x_j}}$$

In order to avoid overflows when implementing this function, it is often rewritten as

$$y_i = \frac{e^{x_{\max} - x_i}}{\sum_{j=0}^{n-1} e^{x_{\max} - x_j}} \quad ,$$

where $x_{\max} = |\vec{x}|_\infty$.

The softmax function is heavily used in transformers, and thus requires a strong hardware support in the ML accelerator.

The reciprocal square root function is also used to accelerate training with the batch normalisation technique, common in Convolutional Neural Networks [70]. It can also be used for transformers [126]. Different steps of output vectors are grouped together in a *batch* of size m, often smaller than n. Batch normalisation operates on the same vector coordinate for different steps, while the softmax function that operates on the coordinates of one step only (Fig. 3.4).

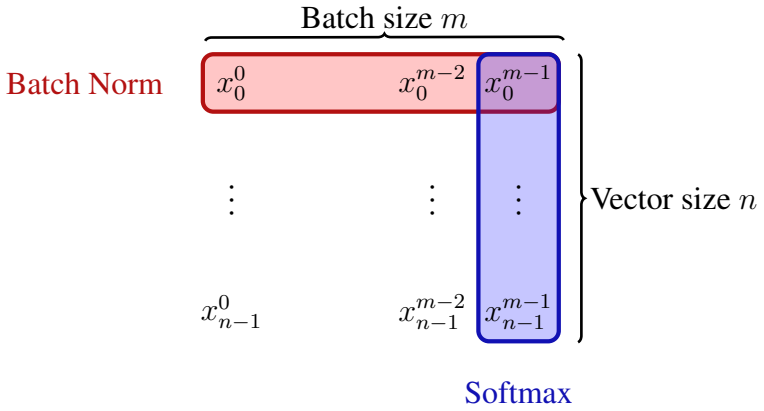


Figure 3.4: Batch normalisation inputs compared to softmax.

It useful to have a hardware support of the inverse square root $\frac{1}{\sqrt{x}}$ for acceleration purposes.

$$\text{RMSNorm}(\vec{x})_i = \frac{x_i}{\text{RMS}(\vec{x})} \text{ where } \text{RMS}(\vec{x}) = \sqrt{\frac{1}{n} \sum_{j=0}^{n-1} x_j^2}$$

3.1.2 Matrix Multiplication

The most computational intensive part of the activation mechanism is the matrix multiplication. This operation needs a huge amount of input data, and also generates

a lot of it. The "Memory Wall", term coined by Wulf and McKee [124], describe how the processing speed increases much faster than the memory speed.

Because of this, the way data is sent between the memory and the computing unit shapes the way the computation is conducted. The bus between the memory and the computation units is very expensive, and making it larger is often not a viable option. The goal is to do as much computation as possible on the data that was brought back.

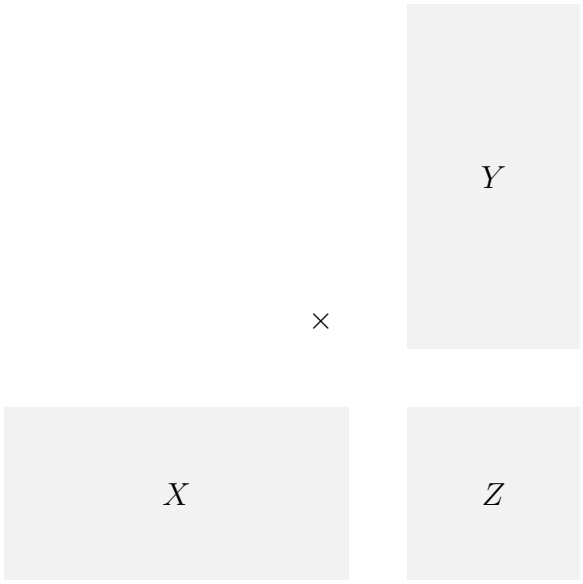


Figure 3.5: Representation of a rectangular matrix multiplication and add: $Z = Z + X \times Y$.

Matrix multiplication (Fig. 3.5) is good to optimise this, as the quantity of input data is in n^2 and the quantity of computations in n^3, where n is the size of the matrix. The matrix multiplication is a very large operation, and is often broken down into smaller unit.

One way is to break it down in outer products $U \otimes V = Z$. The inputs of the operator are two vectors $U, V \in \mathcal{V}_n$, and the output is $Z \in \mathcal{M}_{(n,n)}$, such that $Z_{(i,j)} = U_i \times V_j$. However, this creates a lot of output data, that could be difficult to move out of the computation unit. A solution to mitigate this is to keep the output data in place, add multiple outer products inside it, and only retrieve the result once the computation is finished. This choice is present in Arm's Scalar Matrix Expansion (SME) [2] and IBM POWER 10 matrix-multiply assist (MMA) facility [11].

Another popular choice is the systolic array [82]. It enables multiple computation units (CU) to share data (Fig. 3.7). Instead of keeping data in a centralised place, where each CU can directly load it from memory, only one CU loads parts of the data, and shares it with the next CU that will need it.

For example, value $Y_{0,1}$ is delayed in cycle 1, used to compute $Z_{0,1}$ in cycle 2, $Z_{1,1}$ in cycle 3 and $Z_{2,1}$ in cycle 4.

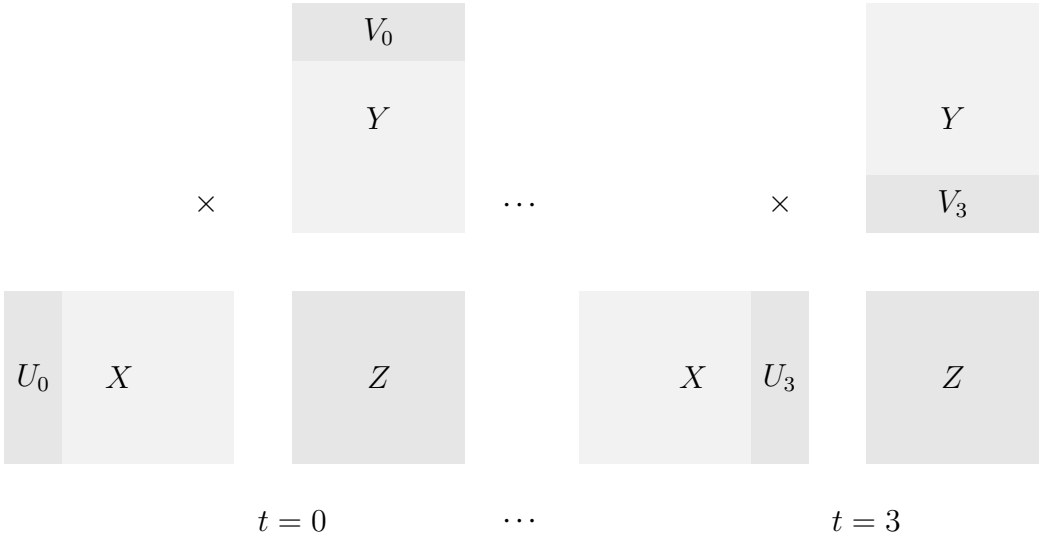


Figure 3.6: Matrix multiplication broken down into multiple outer products. At every step $t = i$, $Z_{i+1} = Z_i + U_i \otimes V_i$.

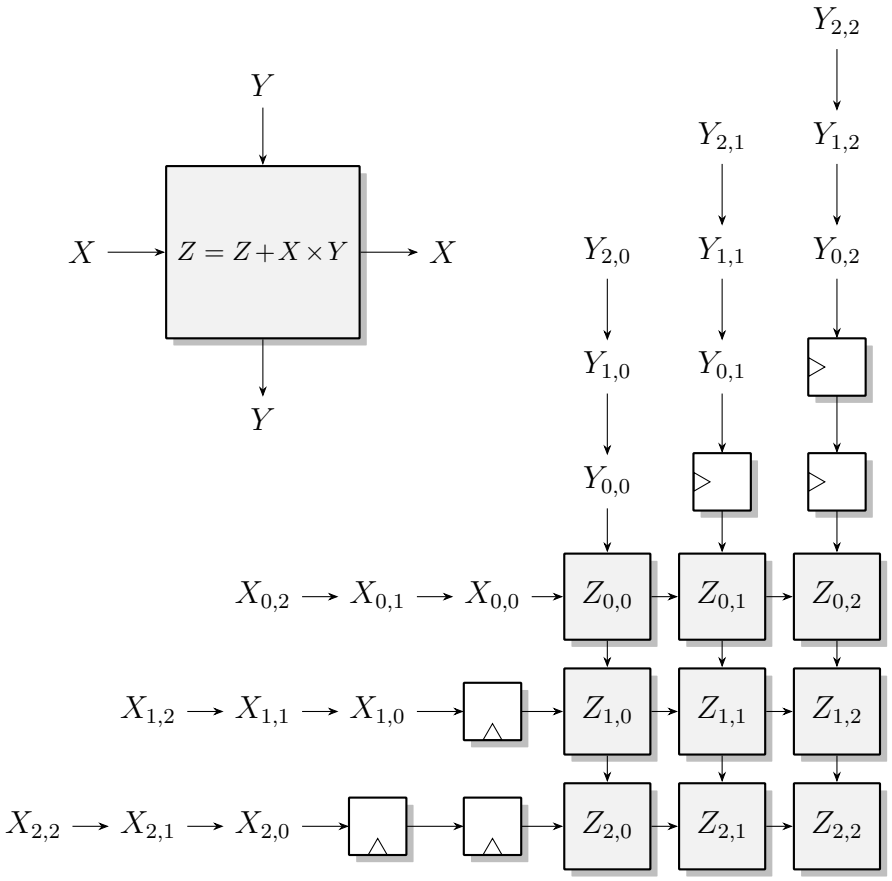


Figure 3.7: A computation unit (top) and its use in a systolic array (bottom) to compute a matrix multiplication.

3.2 Kalray's Scheme

The matrix multiply operation in the Kalray MPPA3 CV2 uses a similar idea as the systolic array.

3.2.1 PE-to-PE

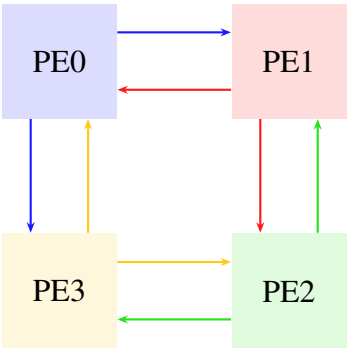


Figure 3.8: Send-receive channels between PEs. The colours are coherent with the PE's number.

The protocol used to organise matrix multiplication is called PE-to-PE [41]. It enables 4 Processing Elements (PE) comprised of a Core and a Tightly Coupled Accelerator (TCA), to load independently part of a large matrix, and share data through dedicated channels (Fig. 3.8).

$\mathcal{M}_{(n,m)}(\mathbb{F})$ refers to the set of matrices of n rows and m columns, filled with numbers, often floating-point of format $\mathbb{F}$.

Each PE (Fig. 3.9) has a hardware matrix-multiplication operator that can do the following operation: $X \times Y + Z$, where $X \in \mathcal{M}_{(4,8)}(\text{FP16}), Y \in \mathcal{M}_{(8,4)}(\text{FP16}), Z \in \mathcal{M}_{(4,4)}(\text{FP32})$. This operator is pipelined in 4 cycles, and contains a bypass such that the output Z can loop into the input without needing to use register space.

The accelerator contains an independent register bank containing 64 registers of 256 bits.

Every cycle, PE0 can load 256 bits of data from memory, and send/receive 256 bits from another PE (either PE1 or PE3). In the matrix-multiply computation, X, Y are both 512 bits inputs, which means that $X_{0,0}$ (Fig. 3.9) is loaded in two halves of 256 bits: $X_{0,0,h}, X_{0,0,l}$. This scheme is software pipelined (Algo. 2).

The global PE-to-PE scheme (Fig. 3.10) works with 4 PEs working together to compute: $Z = X \times Y + Z$, where $X \in \mathcal{M}_{(16,32)}(\text{FP16})$, that is a matrix with 16 rows and 32 columns, filled with FP16 numbers, $Y \in \mathcal{M}_{(32,16)}(\text{FP16}), Z \in \mathcal{M}_{(16,16)}(\text{FP32})$. Each PE loads a fourth of X and Y, while Z is kept in each PE's TCA register. It then receives from two other PE a missing data, enabling each PE to compute a fourth of the result.

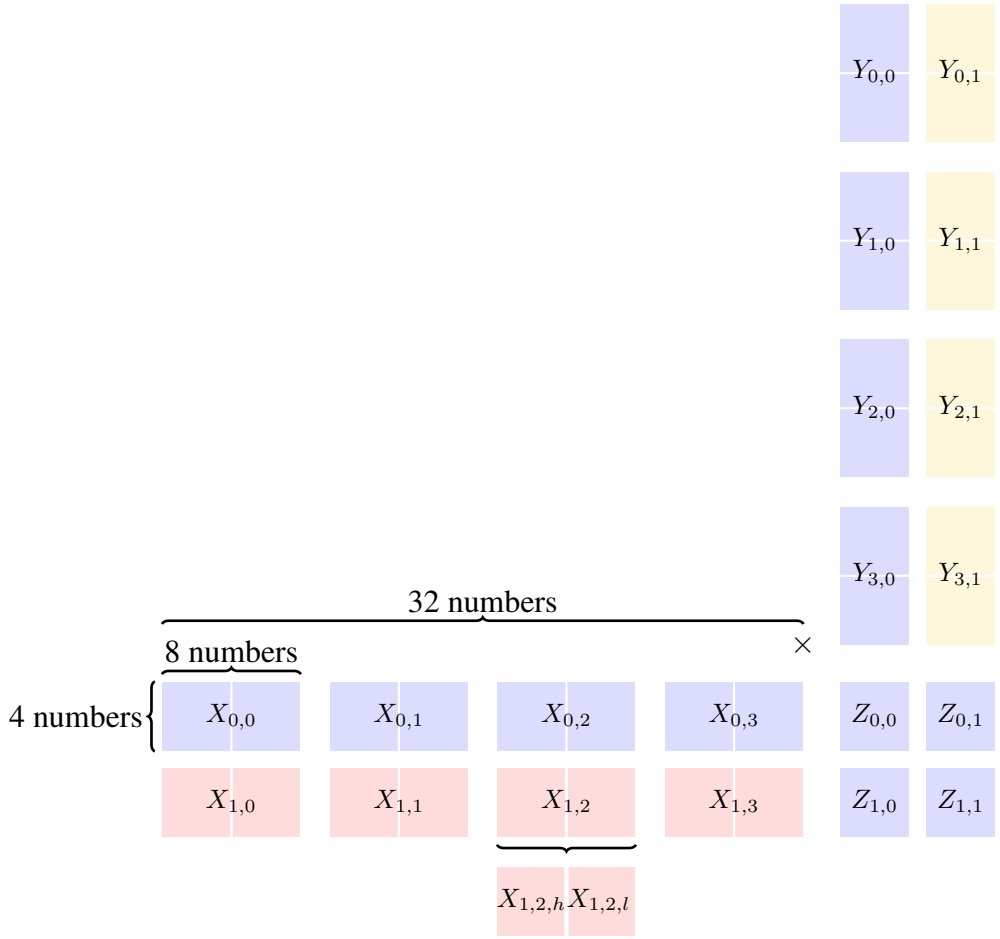


Figure 3.9: Computations executed by PE0, using data loaded by PE0, and shared from PE1 and PE3.

Algorithm 2 Pipelined code of matrix-multiplication for PE0. In black, the first set of matrix multiplication and the loading of the values. In blue, the second set, and in red the third one.

```
 1 load X_{0,0,l}
 2 ; //end of bundle
 3 load Y_{0,0,l} //from SMEM 256b
 4 send X_{0,0,l} to PE1 //256b
 5 receive X_{1,0,l} from PE1 //256b
 6 ;
 7 load X_{0,0,h}
 8 send Y_{0,0,l} to PE3
 9 receive Y_{0,1,l} from PE3
10 ;
11 load Y_{0,0,h}
12 send X_{0,0,h} to PE1
13 receive X_{1,0,h} from PE1
14 ;
15 load X_{0,1,l}
16 send Y_{0,0,h} to PE3
17 receive Y_{0,1,h} from PE3
18 compute X_{0,0} × Y_{0,0} + Z_{0,0}
19 ;
20 load Y_{1,0,l}
21 send X_{0,1,l} to PE1
22 receive X_{1,1,l} from PE1
23 compute X_{0,0} × Y_{0,1} + Z_{0,1}
24 ;
25 load X_{0,1,h}
26 send Y_{1,0,l} to PE3
27 receive Y_{1,1,l} from PE3
28 compute X_{1,0} × Y_{0,0} + Z_{1,0}
29 ;
29 load Y_{1,0,h}
31 send X_{0,1,h} to PE1
32 receive X_{1,1,h} from PE1
33 compute X_{1,1} × Y_{1,1} + Z_{1,1}
34 ;
35 load X_{0,2,l}
36 send Y_{1,0,h} to PE3
37 receive Y_{1,1,h} from PE3
38 compute X_{0,1} × Y_{1,0} + Z_{0,0}
39 ;
```

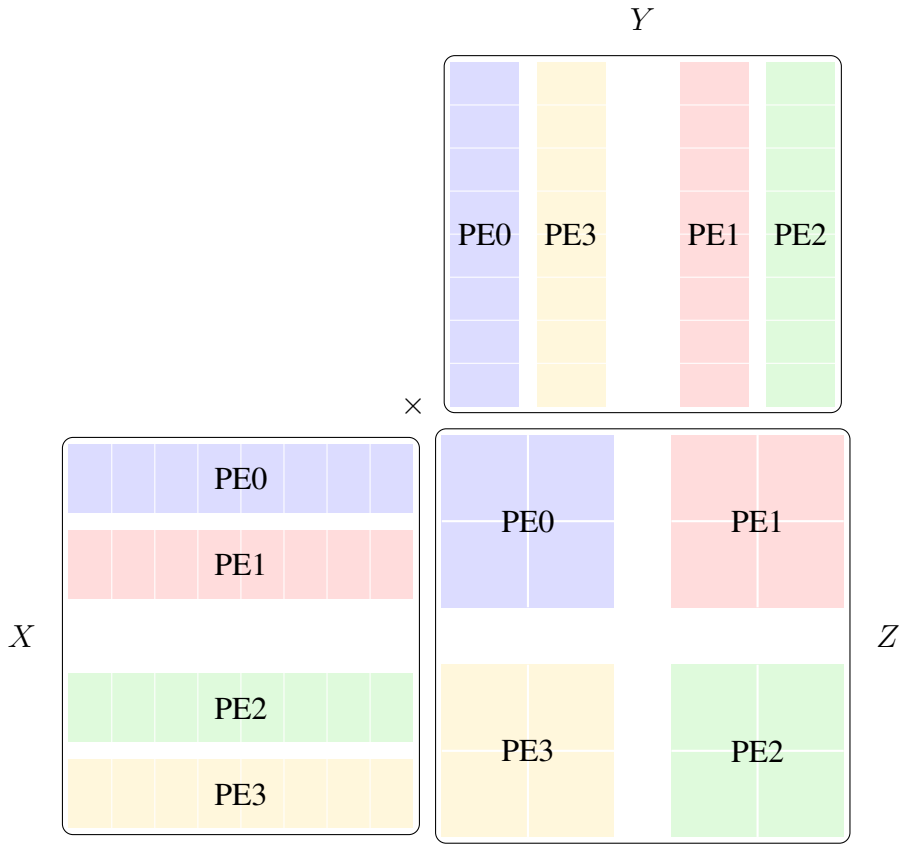


Figure 3.10: PE-to-PE scheme in the MPPA3 CV2, showing how the load of the input matrices X, Y, Z are broken down into the 4 PEs. The matrix multiplication is done in small chunks by the PE that loaded that chunk of Z.

3.2.2 Matrix Accelerators

For the new generation of Kalray processors MPPA4, an alternative route was taken. Instead of distributing the computation in the PE, the matrix-multiply unit is separated in a dedicated accelerator, nicknamed the Loosely Coupled Accelerator (LCA). This dedicated accelerator is better suited to keeping the matrix Z in the operator, which leads to various design choices. The LCA is controlled and fed data by four PE, similarly to the PE-to-PE protocol, and it can compute: $Z = X \times Y + Z$, where $X \in \mathcal{M}_{(32,16)}(\text{FP16}), Y \in \mathcal{M}_{(16,32)}(\text{FP16}), Z \in \mathcal{M}_{(32,32)}(\text{FP32})$.

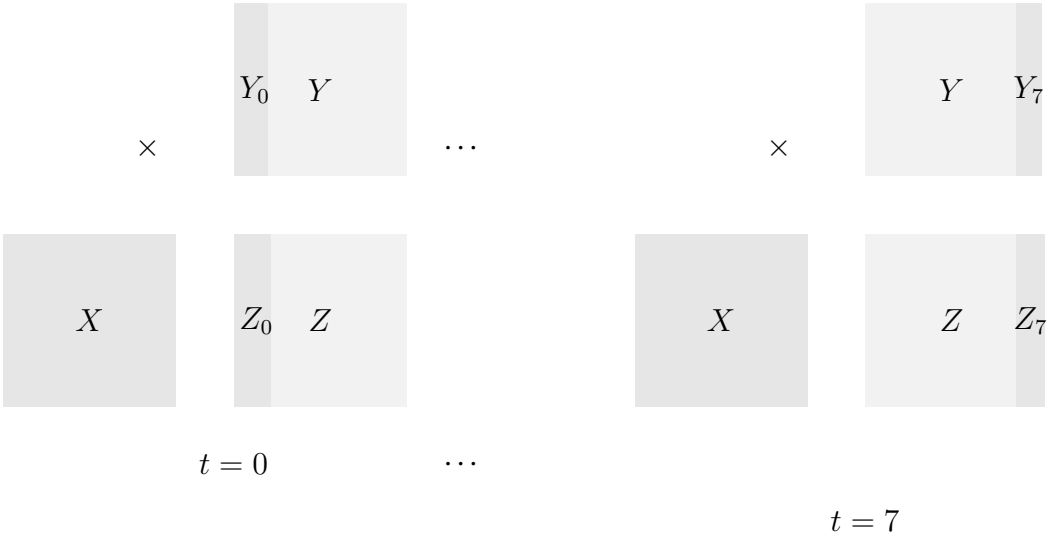


Figure 3.11: Representation of the execution in 8 step, pipelined operation.

This operation is separated into 8 pipelined operations (Fig. 3.11), where the hardware computation is: $Z = X \times Y + Z$, where $X \in \mathcal{M}_{(32,16)}(\text{FP16}), Y \in \mathcal{M}_{(16,4)}(\text{FP16}), Z \in \mathcal{M}_{(32,4)}(\text{FP32})$. The LCA can compute 4096 FLOPs (**F**loating-**P**oint **OP**eration**s**) per cycle, while in the PE-to-PE scheme the 4 PE can only compute 1024.

The organisation of the accelerator plays a great role in its efficiency, and guides the operator design. However, it should also be chosen with the arithmetic in mind, as it has an impact on the computation. For example, an outer product will be more sensitive to cancellation issues than a dot product. In this thesis, the organisation is considered a fixed constraint, and the arithmetic component is designed around it.

3.3 Specific Number Formats for Machine Learning

In an effort to increase the amount of data transported from the memory to the matrix-multiply accelerator, is possible to reduce the precision of the numbers. This enables more numbers to be transported in a bus that has the same width in bits.

Those advancements were made possible by the introduction of quantisation, where a neural network trained with a precise format can have its parameters compressed for inference.

Earlier neural networks used IEEE floats (FP32) for both training and inference. A truncated FP32 was then used, BF16 [69], with the same range as an FP32, but a third of the precision. Half-floats (FP16) are another option that was considered. Then a truncated FP16, called E5M2, or another 8 bit format called E4M3 (with 4 bits of exponent, 3 bits of fraction) were introduced [112], further reducing the size of floating-point numbers used for inference.

Other more exotic formats can be used, for example Posits [54, 10, 87, 98] (more information in section 3.3.2), or the Logarithmic Number System [77, 74, 13].

Fixed-point formats were considered, but were not that effective compared to floating-point formats of the same size. The benefits of FP8 compared to INT8 quantisation are explained by the non-linear sampling of FP8 values [91], which is a better match for Gaussian-like distributions that have more density around zero [83]. In the setting of inference with Post-Training Quantisation (PTQ), [83] found that many computer vision networks benefit from FP8 formats.

Large Language Models are very sensitive to range, requiring scaling steps to achieve satisfactory training [48]. In the current LLM giants, Deepseek [86] was trained using mixed-precision that included FP8, BF16 and FP32, resulting in one of the less expensive LLM training yet.

3.3.1 FP8 - The Struggle for Standardisation

E5M2: The E5M2 format is originally created as a truncated half-float, and is successfully used in Deep Neural Networks for computer vision in [123]. It follows specifications of the IEEE-754 standard [66] without being an official format from the standard.

E4M3: The E4M3 format is a custom format introduced in [112]. E4M3 was first adopted by NVIDIA, ARM and Intel, huge industrial actors, in [91], then by Meta, Google and AMD with the Open Compute Project specification [103].

The E4M3 format follows the philosophy that for small formats, every represented number counts. The IEEE-754 standard contains different encodings for the same (or nearly the same) information. While everyone agrees that some encodings can be used to represent numbers, there are more issues determining which encodings to remove.

If E4M3 followed the IEEE-754 specifications, it would:

- Contain 2 infinities : $+\infty = 0.1111.000$ and $-\infty = 1.1111.000$.
- Contain 2 zeros : $+0 = 0.0000.000$ and $-0 = 1.0000.000$
- Contain 14 NaN : 8 quiet ($x.1111.1xx$) and 6 signalling ($x.1111.0xx$ that are not ∞).

In the spirit of not wasting encoding space, all the standards agree that it is necessary to use subnormals and not flush them to zero.

The original E4M3 [123] removed the encoding of infinity, and kept only 2 NaN values, encoded as $s.1111.111$. The two zeros are kept in the standard. All the values encoded as $s.1111.xxx$ that are not NaN are used to represent numbers with

an unbiased exponent of 8 (where 7 used to be the maximum). Any overflow in computations would return a NaN. This is also the standard agreed on by [103].

A later version [99] by Graphcore proposed to remove the -0 and use the space to encode an unique NaN. This way, all the values encoded as $s.1111.xxx$ are used to represent numbers. Like the previous version, an overflow in computations would return a NaN.

While this could be considered counter-intuitive, it is not a problem to use NaNs in case of infinities. The results are only checked at the end of the network, and an infinity in a matrix multiplication has high chances of generating a NaN, either with addition $\infty - \infty$ or multiplication $\infty \times 0$. Just knowing that an error was made is enough for neural network applications.

Bias value Since FP8 formats have little range, some propositions [83] introduced a format with variable bias value. This allows the bias to be adapted per-channel (weights) or per-tensor (activations), as well a be fixed for a whole network.

This idea is further developed in the OCP Microscaling standard [104], and applied to even smaller formats, like FP6 (E2M3 or E3M2), FP4 (E2M1) or INT4.

IEEE's attempt to standardise: The IEEE working group P3109 is, at the time of writing this thesis, working on a standardisation of the 8 bit floating-point. This would result in a new standard, and not an addendum to IEEE-754. An interim report published in September 2023 [53] gives more information on what this standard could look like.

The first surprising part is that all sizes of exponent and fraction size have been described, from E7M0 to E1M6, and where E1M6 is integer in sign-magnitude representation.

Another is the change to the E5M2 format, that had been until then kept as a truncated FP16. In the IEEE report, the infinity and NaN binade would also be used for numbers, and the bias of the format would be changed from 15 to 16. The reason for this bias change is to have more symmetry in the format, with the smallest exponent being -15, and the largest 15 instead of -14 and 16 respectively. However, this makes some FP16 numbers overflow the E5M2 format, and the conversion between FP16 and E5M2 becomes more expensive.

The working group offers two versions for each format. Using E4M3 as an example: The first version contains infinities, encoded as $s.1111.111$, and a NaN is encoded where -0 would be $1.0000.000$. The second one does not contain infinities, and encodes everything like the Graphcore format. However, in the case of overflow, this format will saturate the number to the largest number for the format $s.1111.111$, instead of a NaN.

Kalray's choice: In order to be compatible with existing hardware and software implementations, Kalray decided to support the OCP FP8 in its accelerator. However, during the early stages of this thesis, the company had a preference for the Graphcore format because it could encode more numbers.

3.3.2 A More Exotic Format: the Posit

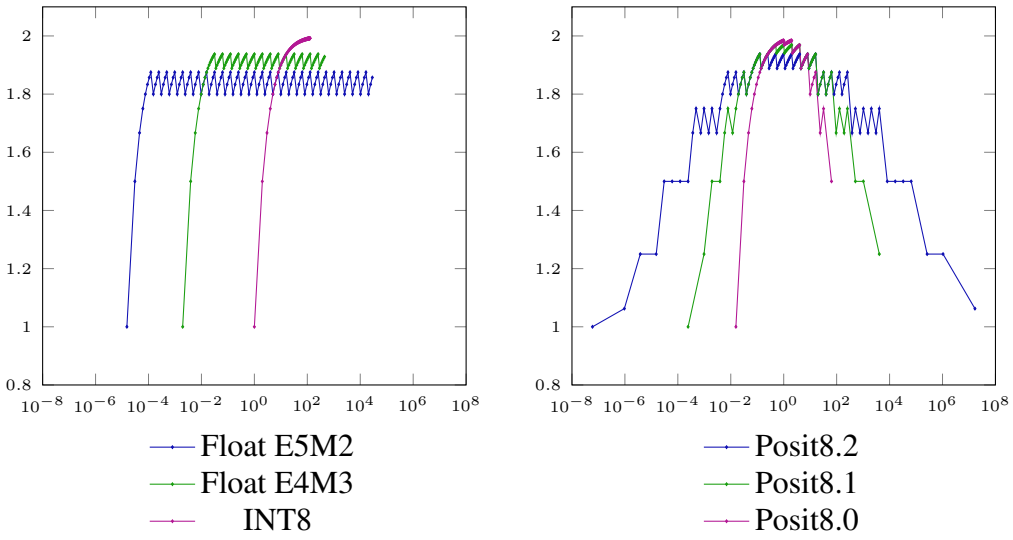


Figure 3.12: Precision of Posit8 ($es \in \{0, 1, 2\}$), FP8 (E4M3, E5M2) and INT8 formats. It is computed as the absolute difference between two successive values divided by the one of the largest magnitude.

Posit Formats The posit representation [55, 54] encodes a floating-point number x with a sign $S \in \{0, 1\}$, a regime value R, an exponent value $E \in [0, 2^{es} - 1]$ and a fraction value $F \in [0, 1)$ as:

$$x = \begin{cases} 0 & \text{if } S = 0 \text{ and all other bits are } 0 \\ \textbf{NaR} & \text{if } S = 1 \text{ and all other bits are } 0 \\ ((1 - 3S) + F) \times 2^{(1-2S)\times(2^{es}\times R+E+S)} & \\ & \text{otherwise} \end{cases}$$

The regime value R is encoded in a bit-field of $k > 0$ identical bits and a bit whose value differs from the previous one (i. e. with an unary representation). If the first regime bit is zero then $R = -k$ else $R = k - 1$. There is only one exception value called NaR (Not a Real). Given that the minimum number of regime bits is two, while the exponent field takes *es* bits before any fraction bits are encoded, the maximum number of fraction bits for the Posit8.*es* representations is $5 - es$.

The posit encoding is easier to understand by first considering case $S = 0$. For the bit patterns different from 0 and from **NaR**, then $x = useed^R \times 2^E \times (1 + F)$, where $useed = 2^{2^{es}}$, so the exponent is actually $2^{es} \times R + E$. In case $S = 1$, one may take the two's complement of the posit bit pattern, decode it as in case $S = 0$, then negate the resulting value. This procedure is typically implemented in hardware to produce the internal floating-point representation of a posit number [10].

Suitability of Posits for Deep Learning The standard Posit8 representation [54] (with *es* = 2) appears especially interesting [87], as it combines a dynamic range larger than the E5M3 representation with a precision comparable to the E4M3 representation around zero (Fig. 3.12). Previous work also confirmed that Posit8.2

performs significantly better than E5M2, Posit8.0 and Posit8.1 on detection and classification networks, while Posit8.3 performed similarly to Posit8.2 [30]. However, Posit8.0 and Posit8.1 arithmetic is less expensive to implement than Posit8.2 arithmetic, so these formats are also included in the scope of our exploration.

For the applications of posit arithmetic to deep learning, the first comparisons [98, 15] of the Posit8 format for training rely on an early version of the posit standard where the *es* parameter is set respectively to 0, 1, 2 for the posit bit sizes of 8, 16, 32. Training with the Posit8 and the Posit16 formats using $es = 1$ [87] explores less standard Posit formats.

Posit8.0 arithmetic exposes satisfactory inference results on smaller networks [15] with Quantisation Aware Training, comparing various small floating-point formats for multiple network models and datasets, including LeNet-5 on the MNIST dataset. Using the much cheaper INT8 arithmetic for this last example results in very similar network accuracy [105].

Chapter 4

Implementing Fixed-Point Arithmetic in Hardware

— $1 + 1 = ?$
— 2 *!*
— *Non.* 1*. On parle une... ou un quand on est ensemble... mais dans notre monde à nous* $1 + 1 = 2$*,* $2 + 2 = 4$ *comme ça on devient selfish, on prend du pognon et on partage pas...*
Mais si $1 + 1 = 1$*... ou peut-être que* $1 + 1 = 11$*, et ça c'est beau !*

– Jean-Claude Van Damme

Floating-point numbers, with their ability to closely mimic the properties and behaviour of real numbers, are a preferred choice for mathematical and scientific computations. Although integers and fixed-point numbers offer fine-grain control and efficiency, the preference for floating-point numbers is rooted in the comfort of their usage.

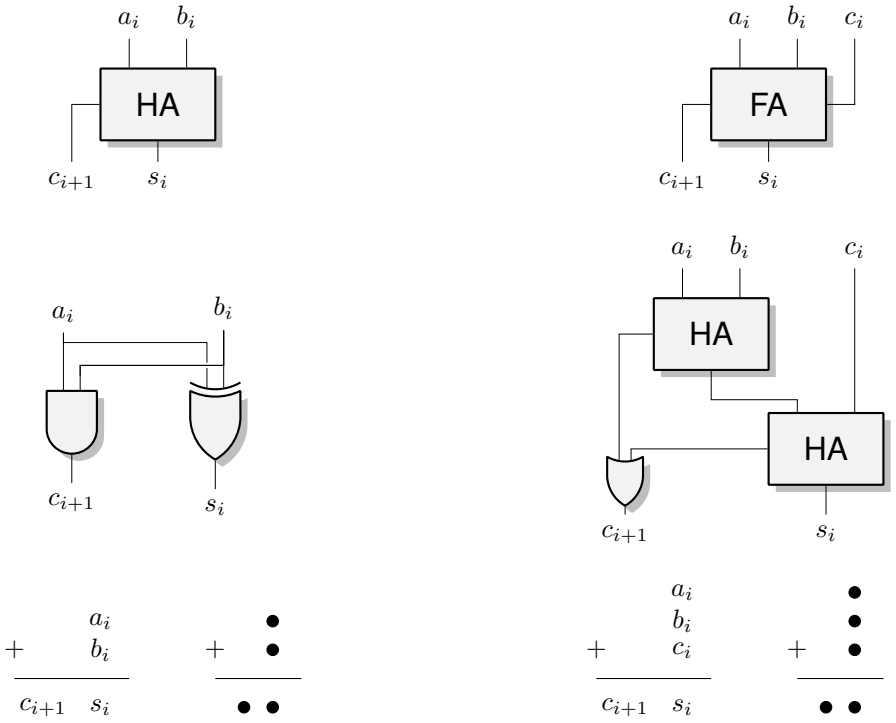
When implementing operations on floating-point numbers in digital systems, it is essential to remember they consist of three parts: an exponent, which is an integer; a significand, which is a fixed-point number; and a sign, which is a binary value. When

designing operators for floating-point numbers, it is inevitable to manipulate the underlying integers and fixed-point numbers, which are implicitly scaled integers.

4.1 Integer Addition

The simplest way to add two integers is to add them bit by bit. This naive operator is called the *Ripple-Carry Adder*. For each bit position, it adds the two bits of the numbers a_i, b_i, as well as the carry c_i arriving from bits of lower weight into the result of the sum for this position s_i and the carry required for the following computations c_{i+1}.

4.1.1 Basic block



(a) Half-Adder **(b)** Full-Adder

Figure 4.1: Operation executed by Half- and Full-adders, as well as their representation.

Half- and Full-Adders (Fig. 4.1) are the basis of the addition in hardware. A Half-Adder (HA) into two bits a_i, b_i that have the same weight, and adds them into a 2-bit positional encoding $c_{i+1}s_i$, where s_i has the same position i as a_i and b_i, and the carry c_{i+1} has the position $i + 1$. The formula executed by a HA is : $s_i = a_i \oplus b_i, c_{i+1} = a_i \wedge b_i$. A Full-Adder (FA) takes a_i, b_i, c_i, and outputs the sum of those three signals as two bits called the sum bit s_i and the carry out c_{i+1}. The operation can also be represented using a dot diagram.

The Full-Adder may computed with two Half-Adders (Fig. 4.1).

4.1.2 Ripple-Carry Adder

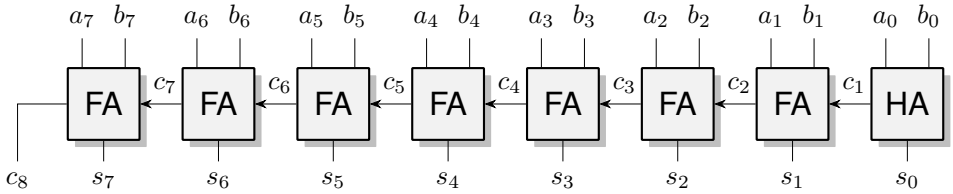


Figure 4.2: Sum of two 8-bit natural numbers with a Ripple-Carry Adder

Those components chained together make a Ripple-Carry Adder (Fig. 4.2). During this computation, s_0 is computed very fast, while the full adder computing s_7 had to wait for $c_1, c_2, c_3, c_4, c_5, c_6$ to be computed. The result of a ripple-carry adder does not have all the bits finishing the computation at the same time. This latency property must be taken into account when designing larger circuits.

The result of the addition of two natural numbers on n bits is a natural number on $n+1$ bits. When adding signed numbers encoded in twos complement, the carry output is not used, as the modular representation on $\mathbb{Z}/2^w\mathbb{Z}$ enables the result to change sign. If there is a risk of overflow and the extra bit is needed, it is simpler to sign-extend the input numbers (see Sec. 2.2) before the addition. In this case, $a_8 = a_7, b_8 = b_7$. After the addition, the sign of the result is in s_9, and c_9 can be ignored.

4.1.3 Integer Subtraction

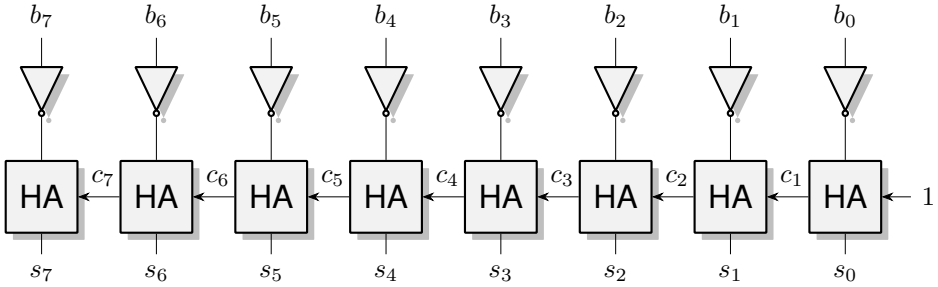


Figure 4.3: Computing $-b$.

If a subtraction $a - b$ is needed, the naive way is to break down the operation into $a + (-b)$. The representation for $-b$ in twos complement can be computed by inverting all the bits of b, and adding 1 in the LSB position (Fig. 4.3). An incrementer is cheaper than an adder as it uses half-adders instead of full-adders.

The computation of $a - b$ and the subsequent addition can be combined into one computation (Fig. 4.4). This way, the carry propagation is only carried out once, reducing the total latency compared to computing $a + (-b)$ by assembling two separate operators $-b$ and $a + x$.

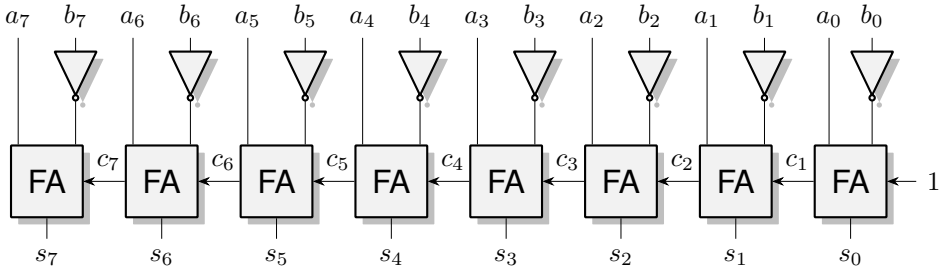


Figure 4.4: Computing $a - b$.

4.1.4 Carry-save

Algorithm 3 Adding 4 numbers with 3 Ripple-Carry Additions

$\texttt{tmp}_0 \Leftarrow X + Y$
$\texttt{tmp}_1 \Leftarrow Z + T$
$R \Leftarrow \texttt{tmp}_0 + \texttt{tmp}_1$

When more than one number must be added, it is also interesting to reduce the number full carry propagations. If 4 numbers X, Y, Z, T are to be added, the naive way is to do 3 additions (Algo. 3). However, this can be reduced with the carry save representation, in which the result is kept as an unevaluated sum of two numbers. The carry output of each adder block is saved as the second number, instead of being propagated.

A Full Adder (Fig. 4.1) can be used to compress three bits into two. A large addition of multiple numbers can be carried out in two phases, compression then final addition. First, the larger numbers are compressed, that is partially added, into a carry-save format. This might be done into multiple steps, but each step is parallel, avoiding high latency carry propagation. The carry-save format is then converted into an integer with a final ripple-carry adder.

Instead of the previous 3 adders, adding 4 numbers can be done with two steps of parallel 3:2 compression, and then the final addition (Fig. 4.5). The length of the critical path is reduced.

In this scheme, the two successive stages of compression are equivalent to the construction of a 4:2 compressor. The use of the 4:2 compressor as a block can help the synthesis tools, and is more comfortable when generating compression trees as 4 is a more hardware friendly number than 3. While it might look like a carry propagation, the carries from the first layer of FA go into the second layer of FA. This chain of 4:2 compressors called a *row compressor* compresses the bits in a parallel way.

In the aforementioned example, all the numbers to be added are on the same number of bits, creating a rectangle of bits to be compressed, called a *bit array* [18, 114, 114].

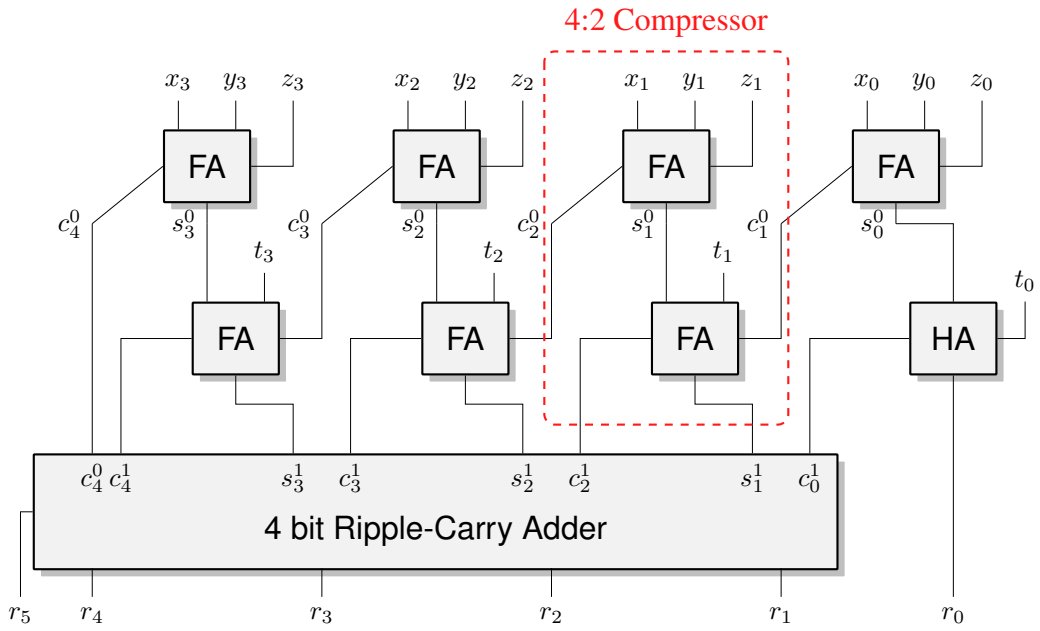


Figure 4.5: Sum of 4-bit numbers $R = X + Y + Z + T$ using full adders (3:2 compressors).

4.1.5 Bit heap

In some cases, the numbers are not all of the same size, and row compressors cannot exactly capture the shape of the bits added (Fig. 4.6). This less regular shape is called a *bit heap* [34]. Extra care must be taken to correctly tile the bit heap with compressors to reduce the latency and area of the sum [81].

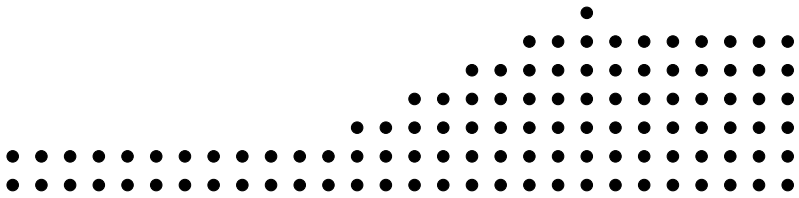


Figure 4.6: Representation of a non-rectangular bit heap. Each dot represents a bit to be added. Figure generated by FloPoCo for the function evaluation of $\sin(\frac{\pi}{4}x)$ on $[0, 1[$: `flopoco generateFigures=1 FixFunctionByMultipartiteTable f="sin(x*pi/4)" lsbIn=-16 lsbOut=-16`.

When the inputs of the sum are signed but the MSB of each number are not the same, sign extension must be performed. Since those bits can be either 0 or 1, this would add many bits to the bit heap, increasing the cost of the addition.

However, a trick [34] can be used to perform a cheaper sign extension in the bit heap scenario. Since all the bits are replicated, the real information only holds on 1 bit. If sign extending with s on 4 bits, instead of adding $ssss$ on the bit heap, one could add $1111_2 + \neg s$. If $s = 0$, $1111_2 + \neg 0 = 1111_2 + 1 = 10000_2$ where the most significant bit is then cut off. If $s = 1$, $1111_2 + \neg 1 = 1111_1 + 0 = 1111_2$, performing the correct sign extension.

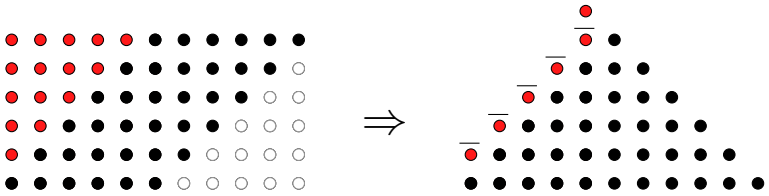


Figure 4.7: Bit heap where sign extension bits (in red) are replaced by a constant and the negation of the sign bit.

Since 1111_2 is a constant, it can be pre-added for all the inputs (Fig. 4.7).

4.2 Rounding

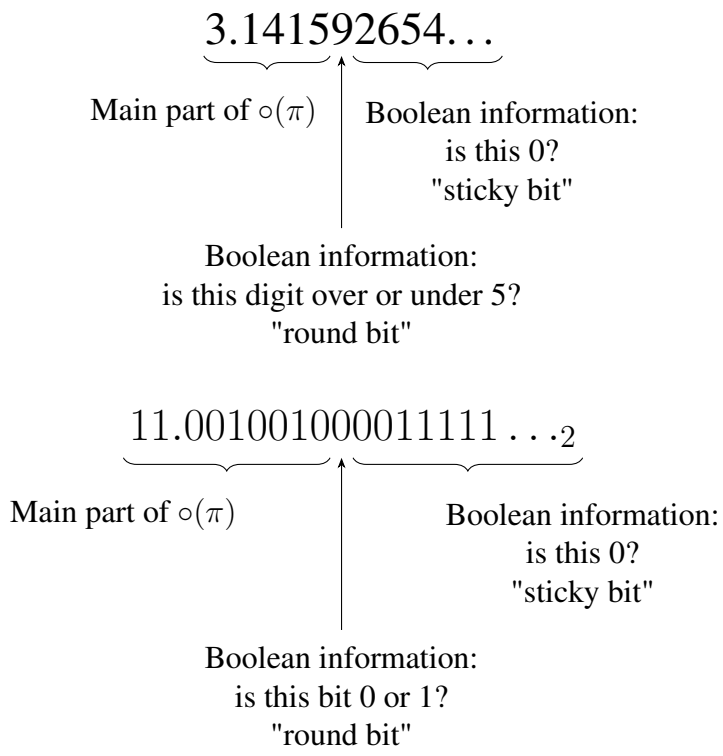


Figure 4.8: Rounding of π on 5 digits (top) or 9 bits (bottom).

To round a number x (Fig. 4.8) to a fixed-point format, multiple pieces of information must be computed. First, the value is truncated to the target precision, in this example this value is $F_{10} = 3.1415$ or $F_2 = 11.0010010_2$. If x is positive, $x \in [F, F + 1\text{ulp}[$, if it negative, $x \in]F - 1\text{ulp}, F]$, where ulp is short for Unit in the Last Place (See. 2.5.1). $F + 1$ulp is the next representable value in the fixed-point format, and $F - 1$ulp is the previous representable value.

Let us assume x is positive, as the negative case is symmetric. The rounded value $\circ(x)$ is either F or $F + 1$ulp. In this example, $F_{10} + 1\text{ulp} = 3.1416$ and $F_2 + 1\text{ulp} = 11.0010011_2$.

When rounding in decimal, one must figure out if x is closer to F, or $F + 1$ulp.

This is done by checking if the first digit truncated out is over or under 5. If it is under 5, then $\circ(x) = F_{10}$, if it is over 5, then $\circ(x) = F_{10} + 1\text{ulp}$. If it is exactly 5, then more information is needed. If any of the subsequent digit are not 0, then x is closer to $F_{10} + 1\text{ulp}$. Otherwise, x is exactly as close to F_{10} and $F_{10} + 1\text{ulp}$, which is called a tie. In decimal, the common tie breaking rule is Away from 0, where $F_{10} + 1\text{ulp}$ is always chosen. In the example, $\circ(x) = 3.1416$ as the round digit is 9.

Rounding in binary follows a similar structure. The first bit truncated out is called the round bit. If it is 0, then x is closer to F_2, otherwise there is a possible tie. If any of the bits of lower magnitude is equal to 1, then x is closer to $F_2 + 1\text{ulp}$. Otherwise all the bits are 0, and x is exactly as close to F_2 and $F_2 + 1\text{ulp}$. The information "is any of the lower-magnitude bits equal to 1" is a boolean value called the *sticky bit*. It can be computed as the OR if those bits.

If the round bit and the sticky bit are both 0, then $x = F$ exactly.

When computing the rounding of a value x, the two options for $\circ(x)$ are computed, F by truncation, and $F + 1\text{ulp}$ using an incrementer (or $F - 1\text{ulp}$ if the sign is negative).

In two's complement, $F - 1\text{ulp}$ must be computed with a subtraction.

In the sign-magnitude representation, $|F| + 1\text{ulp}$ can both computes $F + 1\text{ulp}$ in the positive case, and $F - 1\text{ulp}$ in the negative case.

With the two values computed, a multiplexer chooses the right result depending on a control bit c:

$$\circ(x) = \begin{cases} F \text{ if } c = 0, \\ F + 1\text{ulp} \text{ if } c = 1 \end{cases} .$$

This bit is computed depending on multiple values: the rounding mode (see Sec. 2.5.1), the sign, the round bit, the sticky bit and the value of the Least Significant Bit (LSB) of F.

For the most common rounding modes, and in the sign-magnitude representation, c is computed as follows:

- Round to Zero (RZ): Always use the truncated value F, $c = 0$.
- Round Up (RU): If the sign $s = 0$, the number is negative, $\circ(x) = F$. If $s = 1$, and x is exactly F, then $\circ(x) = F$, otherwise $\circ(x) = F + 1\text{ulp}$. $c = s \wedge (\text{sticky} \vee \text{round})$
- Round Down (RD): If $s = 1$, then $\circ(x) = F$. If $s = 0$, and x is exactly F, then $\circ(x) = F$, otherwise $\circ(x) = F - 1\text{ulp}$. $c = (\neg s) \wedge (\text{sticky} \vee \text{round})$
- Round to Nearest ties to Even (RNE): If the round bit is 0, $\circ(x) = F$. If it is 1 and the sticky bit is also 1, $\circ(x) = F + 1\text{ulp}$. In the case of a tie, when round is 1 and sticky is 0, then the result is either number, between F and $F + 1\text{ulp}$ (or $F - 1\text{ulp}$) whose least significant bit is 0. $c = \text{round} \wedge (\text{sticky} \vee \text{LSB})$
- Round to Nearest ties Away from zero (RNA): Similar to RNE, but in case of a tie the number is never rounded to F. $c = \text{round}$

Round to Zero is a very cheap rounding mode as there is no need to compute $F + 1\text{ulp}$, explaining why some GPUs choose to only support RZ in their accelerator, like the NVIDIA Tensor Core [90].

Round to Nearest ties Away from zero can equivalently be computed by adding $\frac{1}{2}$ulp before truncating. This is also a cheap scenario, as it does not require the separate incrementer and multiplexer, and the $\frac{1}{2}$ulp can often be added during the computation of x for cheap, for instance if it is included into a bit heap. This extra bit for rounding is the single bit on top of the bit heap shown in Fig. 4.6.

Faithful rounding leaves even more freedom in rounding, as either F and $F+1$ulp are acceptable values. Less information on x must be known to compute its faithful rounding than its correct rounding.

4.3 Integer Multiplication

						0	1	1	0	1	1
					×	1	1	0	1	0	1
						0	1	1	0	1	1
+					0	0	0	0	0	0	0
+				0	1	1	0	1	1	0	0
+			0	0	0	0	0	0	0	0	0
+		0	1	1	0	1	1	0	0	0	0
+	0	1	1	0	1	1	0	0	0	0	0
0	1	0	1	1	0	0	1	0	1	1	1

Figure 4.9: Binary multiplication of 27 and 53.

Hardware multiplication of X and Y is carried out the same way as the paper-and-pencil multiplication (Fig. 4.9).

$$X \times Y = \sum_{j=0}^{w-1} (y_j \times X \times 2^j) \quad .$$

The first number X is multiplied by each digit of the second number Y, and shifted accordingly. The terms $y_i \times X \times 2^j$ are called the partial products. It is even simpler in binary than in decimal to compute the partial products, as every bit y_i of Y is either 0 or 1, that is $y_j \times X$ is either X or 0.

This multiplication can also be seen as a bit heap (Fig. 4.10):

$$X \times Y = \sum_{i=0}^{w-1} \sum_{j=0}^{w-1} (x_i \wedge y_j) \times 2^{i+j} \quad .$$

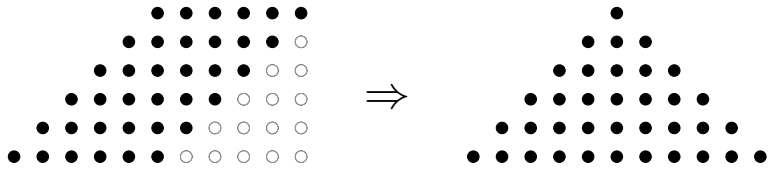


Figure 4.10: Bit heap representation of the 6×6 multiplier of the previous example.

The construction of the bit heap is not very expensive, using only AND gates, and the placement of the bit at the correct power of two is pre-computed. The bulk of the hardware cost is in the compression of the bit heap, which has been the subject of much research [18, 114, 114, 81]. A trick called Booth encoding [6] halves the height of the bit heap.

4.4 Bit Vector Manipulations

While manipulations of the bit vectors are not exactly unique to fixed-point, some of those operations are useful basic components for fixed-point and floating-point operators, in particular, the Shifter and the Leading Digit Counter.

All the figures from this section come from [34].

4.4.1 Shifter

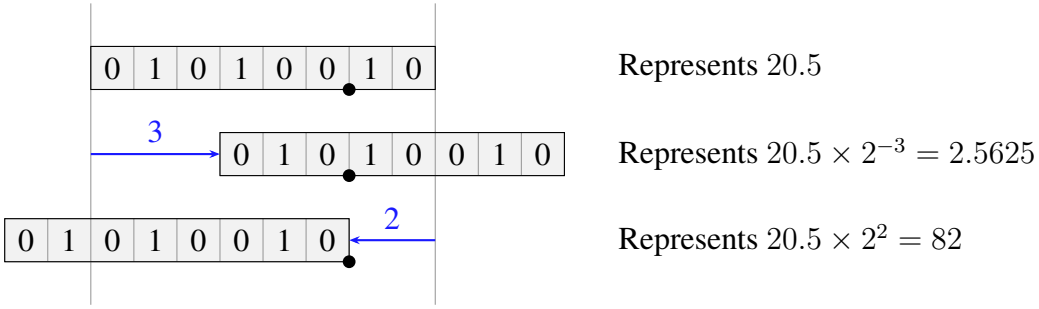


Figure 4.11: Example of a right and left shift.

In decimal, shifting the decimal point corresponds to multiplying by a power of 10. In binary, it is equivalent to multiplying by a power of 2 (Fig. 4.11). In this example, the fixed-point numbers are shifted by a constant, are still encoded on 8 bits, and only the position of the relative point changed. Constant shifts, which are just a change of implicit scaling, do not require any logic to be carried out.

Shifting by a variable integer requires a shifter operator. It inputs a fixed-point number X and a natural number S, and shifts the fixed-point by S positions. In the case of a left shifter, the output represents $X \times 2^S$, and in the case of a right shifter, the output represents $X \times 2^{-S}$.

S can be signed and the resulting shifter is bidirectional. Most of the shifters needed for the arithmetic operations on fixed-point and floating-point numbers can be made unidirectional by either shifting X left starting by the right-most position, or shifting X right starting by the left-most position. This transformation only requires

adding a constant to S, which can often be slipped into the hardware component that is computing the shift value S.

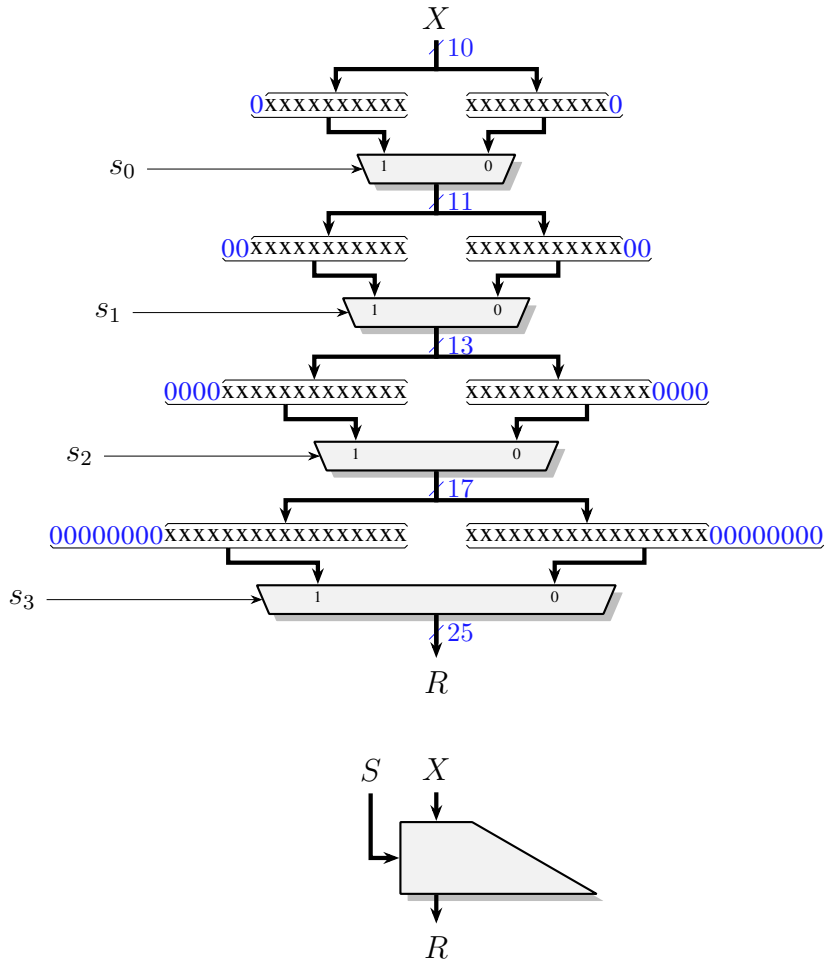
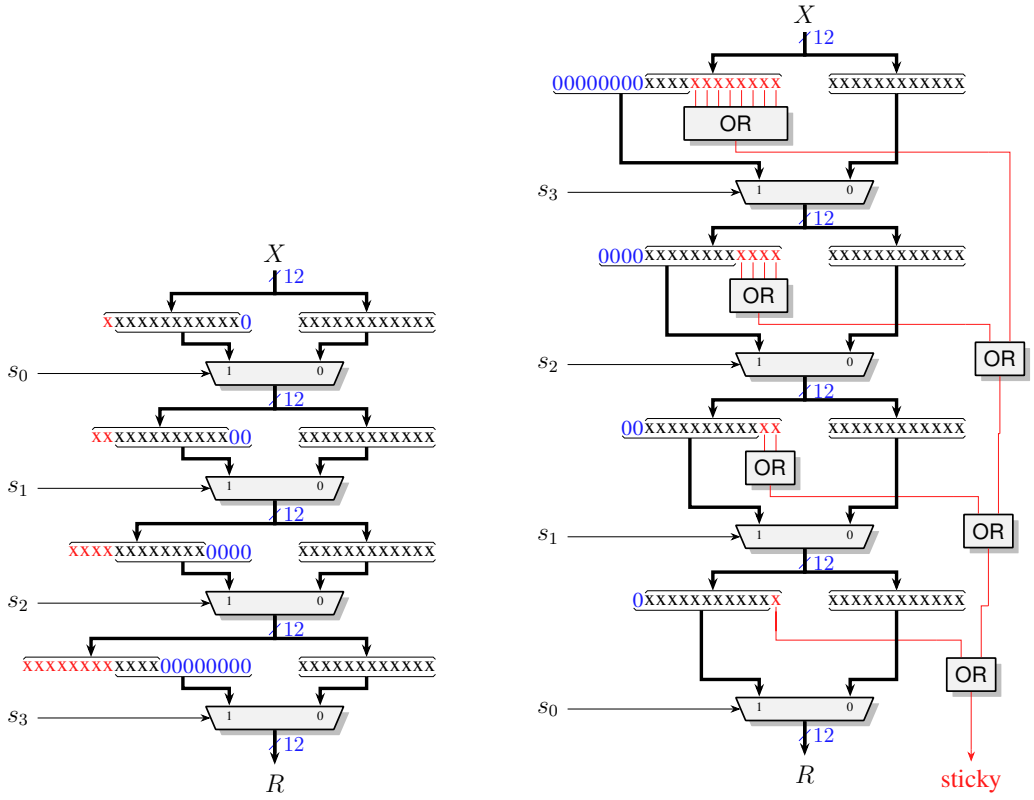


Figure 4.12: Architecture of a full right shifter for $w_X = 10$ and *max_shift* $= 15$. The s_i are the bits of S. For a left shifter, just exchange the two inputs to each multiplexer, or (equivalently) complement all the s_i. Figure from [34]. Below, the common representation of a full right shifter.

The full shifter operator (Fig. 4.12) is made of successive multiplexers (Fig. 1.15). In this example, X of size $w_X = 10$, can be shifted right by $S = \sum_{i=0}^{3} s_i \times 2^i$ positions, that is a maximum of 15 positions. The shifter output R is of size $w_R = 10 + 15 = 25$. In the first step, X is either not shifted or shifted by 1, depending on the value of s_0. In the i^{th} step, X is either not shifted or shifted by 2^i, depending on the value of s_i.

In this example, both sides of the number are padded with bits equal to 0. For integers encoded in twos complement, it is desirable to pad on the left with the sign bit of X instead of 0, which performs the sign extension.

In some cases, one might want a shifter where $w_R < w_X +$ *max_shift*, which leads to a possible loss of information. If $w_R = w_X$, then the shifter is called a barrel shifter. The bits that are shifted out can either be thrown away (Fig. 4.13a), or ORed together as a way to detect if only 0s were lost, which is no loss of information, or if



(a) Architecture of a 12-bit *left* barrel shifter. The red bits are discarded.

(b) Architecture of a 12-bit *right* barrel shifter with early sticky bit computation.

Figure 4.13: Architecture of two shifters with loss of information. Figures from [34].

there was any 1 (Fig. 4.13b).

4.4.2 Leading Digit Counter

The Leading Zero Count (LZC) counts the number of most significant zeros to the left of the first one. The Leading One Count (LZC) counts the number of most significant ones to the left of the first zero. The Leading Digit Count is either a LZC if the value of the MSB is 0, or a LOC. The following explanations will use the Leading Zero Counter as an example.

The construction of the LZC operator (Fig. 4.14) uses a similar idea as the shifter, but functions the other way around. The size of the count C is determined by the maximum number of zeros that can be counted in the input X: $w_C = \lceil \log_2(w_X) \rceil$. Let $w_X = 14$ for the example, then $w_C = 4$. If the first $2^{w_C-1} = 8$ bits are 0, then $c_{w_C-1} = 1$, and those first 8 zeros are removed from X. Otherwise, $c_{w_C-1} = 0$, and the 8 last bits of X are removed. This repeats until X was completely counted.

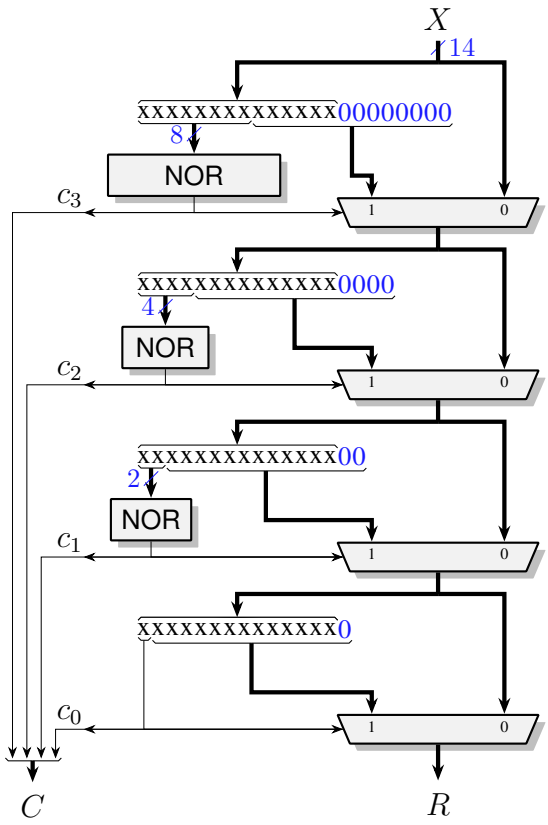


Figure 4.14: Architecture of a 14-bit combined leading zero counter and left shifter. Figure from [34].

Often when dealing with floating-point numbers, the input X is shifted left by the result of C after the LZC. Since the LZC already contains partial shifting of X for the count, the two operators (LZC and Shift) can be combined into a Normaliser. A Normaliser takes the input X, and returns C the count of leading zeros in X, as well as $R = X \times 2^C$. This component is crucial to construct the normalised significands in floating-point operators.

Now that all the important subcomponents are described, we can dive into the construction of floating-point operators.

Part II

Matrix Multiplication

Chapter 5

Exact Dot Product for Small Precisions

This chapter compares the implementation cost of a dot product operator depending on which 8-bit format is used. It aims to help machine learning researchers make informed decisions when choosing formats for their networks. In machine learning, the metric often used to compare the efficiency of formats is the network accuracy compared to the number of multiply and add operations performed. This metric tends to ignore that operating on some formats is much more expensive than others.

The considered 8-bit formats are int8, Posit8 with $es \in \{0, 1, 2, 3\}$, and the two FP8 formats: E5M2 and E4M3 (see Sec. 3.3). Int8 is a common format used for very cheap deep neural networks computation, however its lack of range make it

less effective than floating-point formats. The wider dynamic range of 8-bit floating-point formats allows to directly quantise a pre-trained model down to 8 bits without losing accuracy by fine-tuning batch normalisation statistics [112]. For the sake of comparison, FP16 is also included as it is a standard format for machine learning.

5.1 Floating-point Hardware Basic Blocks

The operators presented for this comparison are constructed using multiple blocks that are at the basis of most floating-point operators.

5.1.1 Internal Format and Floating-point Pack and Unpack

As a reminder (see Sec. 2.4), normal floating-point numbers are encoded as a sign S, a biased exponent E, and a fraction F that is interpreted as $(-1)^S \times 2^{E-b} \times (1+F)$.

Subnormal floating-point numbers have an exponent equal to 0, a sign S, and a fraction F that is interpreted as $(-1)^S \times 2^{0-b+1} \times (0+F)$.

The floating-point format is a format where some information is compressed, in particular the encoding of the special values, and the implicit bit of the significand. In order to compute with floating-point numbers, one must first retrieve the compressed information, that is unpack the floating-point in an internal format. This internal format has an unique representation for both normal and subnormal numbers.

Internal Format and Floating-point Unpack

This internal format (Fig. 5.1) contains its sign X_{sign}, its biased exponent X_{exp} with an unified exponent for subnormals, its significand X_{sig}, and a vector of flags X_{flags} (isnormal, isinf, isnan, issignan, iszero). Unpacking enables to have a single representation for both subnormal and normal numbers: $(-1)^{X_{\text{sign}}} \times 2^{X_{\text{exp}}-b} \times X_{\text{sig}}$. Here, X_{sig} is in the format uFix$(0, -w_F)$, while F was in the format uFix$(-1, -w_F)$.

The implicit bit is made explicit in the significand, so there is no need to normalise subnormal inputs, they will just have a non-normalised significand. In other words, X_{sig} can be any value of the fixed-point format uFix$(0, -w_F)$. The subsequent additions or multiplication of X_{sig} will use fixed-point hardware and work all the same.

Floating-point Rounding and Packing

When rounding a number x into floating-point number of format $\mathbb{F}(w_E, w_F)$, one must fill the three fields of the number.

The sign X_{sign} is easy to fill. The rest of the computation is then done on $|x|$, except for the rounding of the fraction.

If x is a special value: exact zero, NaN or $\pm\infty$, then it can be encoded directly into the floating-point format.

For the exponent, one must compute the weight of the most significant 1 of x. When converting a fixed-point number into floating-point, a Leading Zero Counter

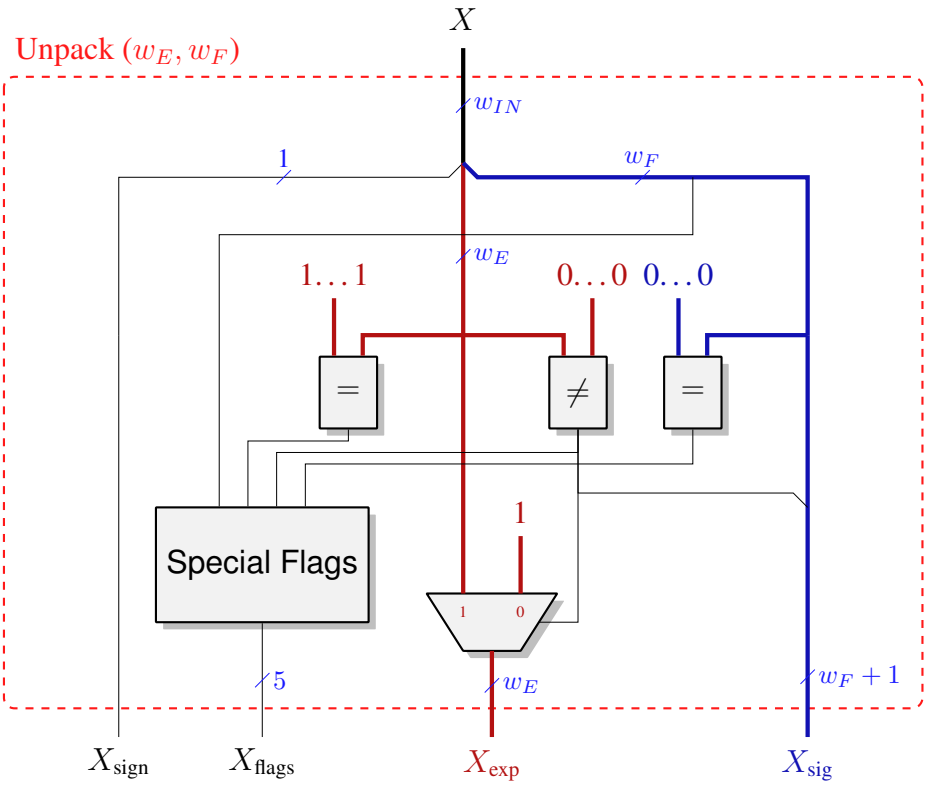


Figure 5.1: Unpacking a floating-point number with exponent size w_E and fraction size w_F.

(LZC) can be used. The unbiased exponent $e = E - b$ is computed relative to the fixed MSB of the fixed-point format. If the result of the LZC is l, then $e = \text{MSB} - l$.

If $e > e_{\max}$, then x is too big (in magnitude) to be representable in the chosen format, $\circ(x) = \pm\infty$. If $e < e_{\min}$, then x is too small to be representable, and x is rounded to zero: $\circ(x) = \pm 0$. If $e_{\min} < e < e_{\min\_\text{normal}}$, then x is a subnormal and its exponent is $e' = e_{\min\_\text{normal}}$. Otherwise, $e_{\min\_\text{normal}} < e < e_{\max}$, then x is a normal number and its exponent is $e' = e$. This classification is not final as the rounding of F can modify it, however it changes how F is computed.

The rounded significand of the result is computed as the fixed-point rounding of $|x| \times 2^{-e'}$ in $\text{uFix}(0, -w_F)$, following the rounding described in Sec. 4.2. The incrementation when rounding gives a temporary post-rounding significand $M_{\text{tmp}} \in \text{uFix}(1, -w_F)$[1]: the bits of M_{tmp} are written $m_1 m_0 \dots m_{-w_F}$.

The final exponent e_{final} is either e' or $e' + 1$ depending on if carries during the incrementation changed the position of the first significant one. The biased exponent is $E = e_{\text{final}} + b$ for normal numbers, or $E = 0$ for subnormal numbers.

If the number is normal, and if the carries did not change the position of the first significant one ($m_1 = 0, m_0 = 1$), then $F = m_{-1} m_{-2} \dots m_{-w_F}$ and $e_{\text{final}} = e'$. If the number is normal, and the carries changed the position of the first significant one ($m_1 = 1$), then $M_{\text{tmp}} = 100\dots0_2$ and $F = 0$ and $e_{\text{final}} = e' + 1$, potentially resulting in an overflow result.

If the number is subnormal, and rounding made it not subnormal ($m_1 = 0, m_0 = 1$), then $F = m_{-1} m_{-2} \dots m_{-w_F}$ and $e_{\text{final}} = e_{\min\_\text{normal}}$. If the number is subnormal, and rounding did not change that ($m_1 = 0, m_0 = 0$), then $F = m_{-1} m_{-2} \dots m_{-w_F}$ and $E = 0$.

As the bias b is a constant, its value can be included into other additions on the exponent (like the subtraction by the leading zero count), and it is mostly ignored in the literature when describing floating-point operators.

Moreover, since $e_{\text{final}} = e' + 1$ only when a carry comes out of the incrementer, incrementing the concatenation of the biased exponent and the truncated fraction EF_{trunc} will result in the correct packing, taking care of all the issues with subnormals and $\pm\infty$ encoding.

5.1.2 Posit Pack and Unpack to Internal Format

As a reminder, posits are encoded such that:

$$x = \begin{cases} 0 & \text{if } S = 0 \text{ and all other bits are } 0 \\ \textbf{NaR} & \text{if } S = 1 \text{ and all other bits are } 0 \\ ((1 - 3S) + F) \times 2^{(1-2S)\times(2^{es}\times R + E + S)} & \\ & \text{otherwise} \quad . \end{cases}$$

The regime R is encoded in unary in the format, while the exponent E and the fraction F are encoded in binary, like floating-point formats.

[1] The term *mantissa* is generally avoided when talking about floating-point numbers as it is unclear if it refers to the fraction or the significand. However, since S is already used for the sign, the shorthand M is occasionally used for significands in this thesis.

Posit Unpack

Unpacking posits is less straightforward than floating-point numbers due to the variable field width of the regime R.

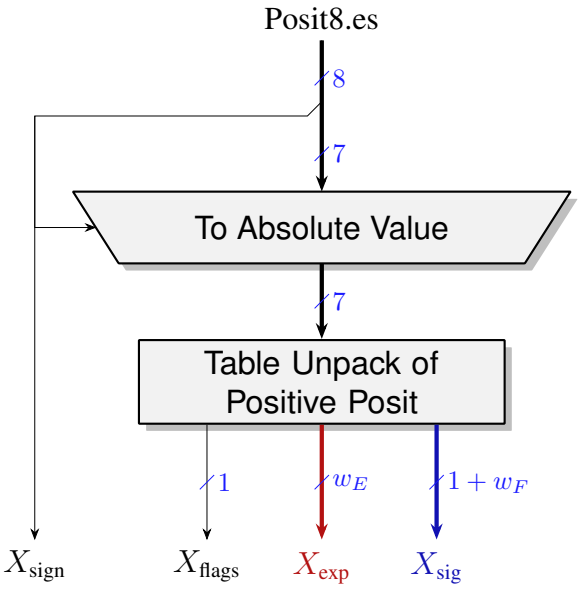


Figure 5.2: Architecture of a table-based posit unpacking.

Various architectures can be used to unpack posits into an internal format, from architectures specific to unpacking [119] to adapted conversion from posit to floating-point architectures [87, 97, 30].

Exploration of various conversion methods in [30] suggests that a factored table implementation (Fig. 5.2) is cheaper than a logic implementation when converting Posit8 to FP16. For unpacking, the difference should be even starker as the output of an unpacked Posit8 is smaller than an FP16. An internal format that can accommodate the Posit8.es input format has $w_E = 5 - es$ exponent bits, $w_F = 4 + es$ fraction bits, a sign bit and an exception bit to encode the NaR value. The size of the table output in the decomposition is $1 + w_E + (1 + w_F) = 11$ bits.

Posit Rounding and Packing

Rounding a posit is similar to rounding a floating-point number, except that the fraction field does not have a fixed size.

The es least significant bits of the exponent of a posit are encoded in binary in the exponent field, and the most significant bits of the exponent are encoded in the regime field in unary.

If the regime field is k zeros followed by a one then $R = -k$, while k ones followed by a zero then $R = k + 1$

Once the size of the regime field is known, the exponent and the fraction are rounded and packed similarly to floating-point numbers.

5.1.3 Multiplication on the Internal Format

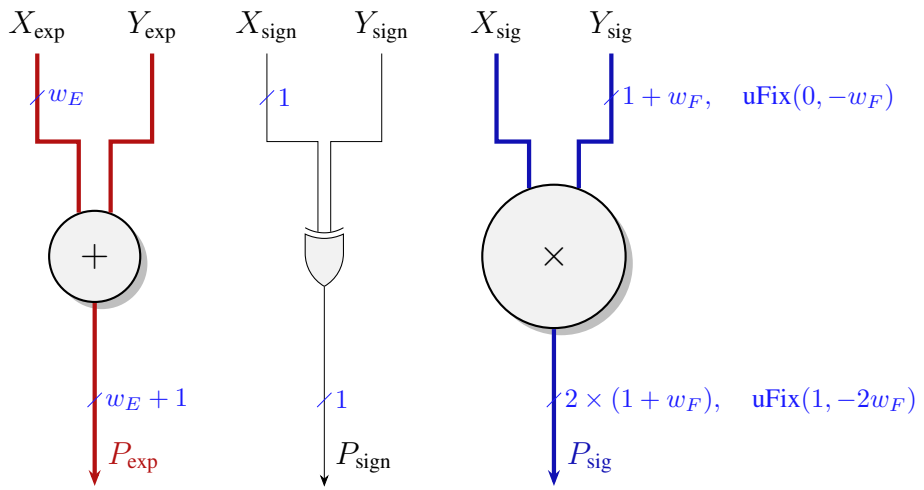


Figure 5.3: Core computation of the floating-point multiplication. It is preceded by floating-point unpacking, and followed by floating-point repacking

Floating-point numbers are simpler to multiply than add (Fig. 5.3). Let X and Y be two unpacked floating-point numbers.

The formulas for multiplying X by Y, using notations from Fig. 5.3, are:

$$\begin{aligned} X \times Y &= ((-1)^{X_{\text{sign}}} \times 2^{X_{\text{exp}}-b} \times X_{\text{sig}}) \times ((-1)^{Y_{\text{sign}}} \times 2^{Y_{\text{exp}}-b} \times Y_{\text{sig}}) \\ &= (-1)^{X_{\text{sign}} \oplus Y_{\text{sign}}} \times 2^{X_{\text{exp}}-b+Y_{\text{exp}}-b} \times (X_{\text{sig}} \times Y_{\text{sig}}) \\ &= (-1)^{X_{\text{sign}} \oplus Y_{\text{sign}}} \times 2^{(X_{\text{exp}}+Y_{\text{exp}})-2b} \times X_{\text{sig}} \times Y_{\text{sig}} \\ &= (-1)^{P_{\text{sign}}} \times 2^{P_{\text{exp}}-2b} \times P_{\text{sig}} \quad . \end{aligned}$$

This component computes the exact result of the product. It also works if the inputs were not normalised numbers, and does not require normalising input numbers.

If the result need to be rounded to an IEEE number, a rounding component can follow the product (see Sec. 5.1.1). Beware that the exponent P_{exp} here has a bias of $2b$, and that P_{sig} is in the format of uFix$(1, -2 \times w_F)$, which means it has two bits to the left of the fixed point.

5.1.4 Addition

The difficulty in floating-point addition is that the fractions cannot be added as is, except if they have the exact same exponent. An alignment phase must be executed beforehand, so that each bit of one number is added to the bit of the same weight of the second number.

For any given floating-point format $\mathbb{F}(w_E, w_F)$, a associated fixed-point format sFix$(2^{w_E-1}, -2^{w_E-1} + 2 - w_F)$ can represent every floating-point number without loss of precision. The alignment phase needed for floating-point addition (Fig. 5.4) can be done by converting the input floating-point into this fixed-point format, and

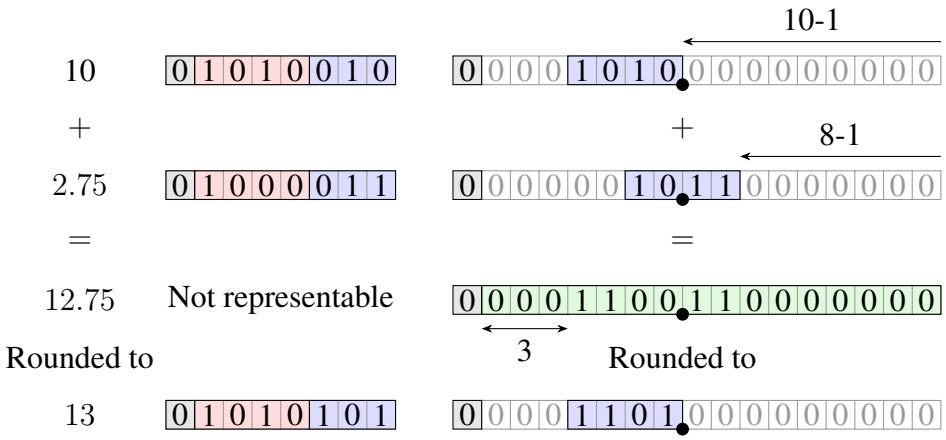


Figure 5.4: Example of the sum of floating-point numbers in $\mathbb{F}(4,3)$ (left) using its expansion in the fixed-point encoding $\text{sFix}(8,-9)$ (right).

then adding the fixed-point numbers together. The fixed-point number is finally converted into floating-point.

In hardware (Fig. 5.5), transforming an input X into its fixed-point representation can be done by shifting left the significand X_{sig} by the biased exponent $X_{\text{exp}} - 1$. Once the sum is carried out in fixed-point, a Leading Zero Count enables to compute the exponent of the result. The fixed-point number is then normalised and rounded (see Sec. 5.1.1).

This is obviously a large adder for very few non-zero bits. It makes sense when adding many terms of a format with little range, which is the object of this chapter. Using a full-precision fixed-point accumulator (often referred to as a Kulisch accumulator) was proposed for a sum of many terms as a way to avoid loosing accuracy [78, 80].

Early Long Accumulators were iterative, accumulating one product per instruction [78, 80]. The corresponding large fixed-point addition can be sped up thanks to parallel execution [79]. A two's complement, high-radix carry-save representation of the accumulator allows for high frequency operation at low hardware cost [37].

The classic method for floating-point addition reduces the size of the adder, by making the alignment a little more complicated. This method is described in the next chapter (Chap. 6).

5.1.5 Fused Multiply-Add

The Fused Multiply Add (FMA) is an IEEE operation computing $\circ(x \times y + z)$ with one rounding. It enables to speed up computations, as one instruction computes two operations, and is also more accurate since there is only one rounding step.

The full-precision fixed-point alignment method of the previous section can be used for the FMA. A product of floating-point numbers also has an equivalent fixed-point representation, which be deduced from the parameters w_E, w_F of the floating-point format. For products, $\text{LSB} = -2 \times (2^{w_E-1} - 2 + w_F)$ and $\text{MSB} = 2 \times (2^{w_E-1} - 1)$. This defines the full-size in bits w_p of the fixed-point representation of the product. Values for the general purpose floating-points formats are summarised in Table 5.1. A more specific table for different 8-bit formats is given later (Tab. 5.2).

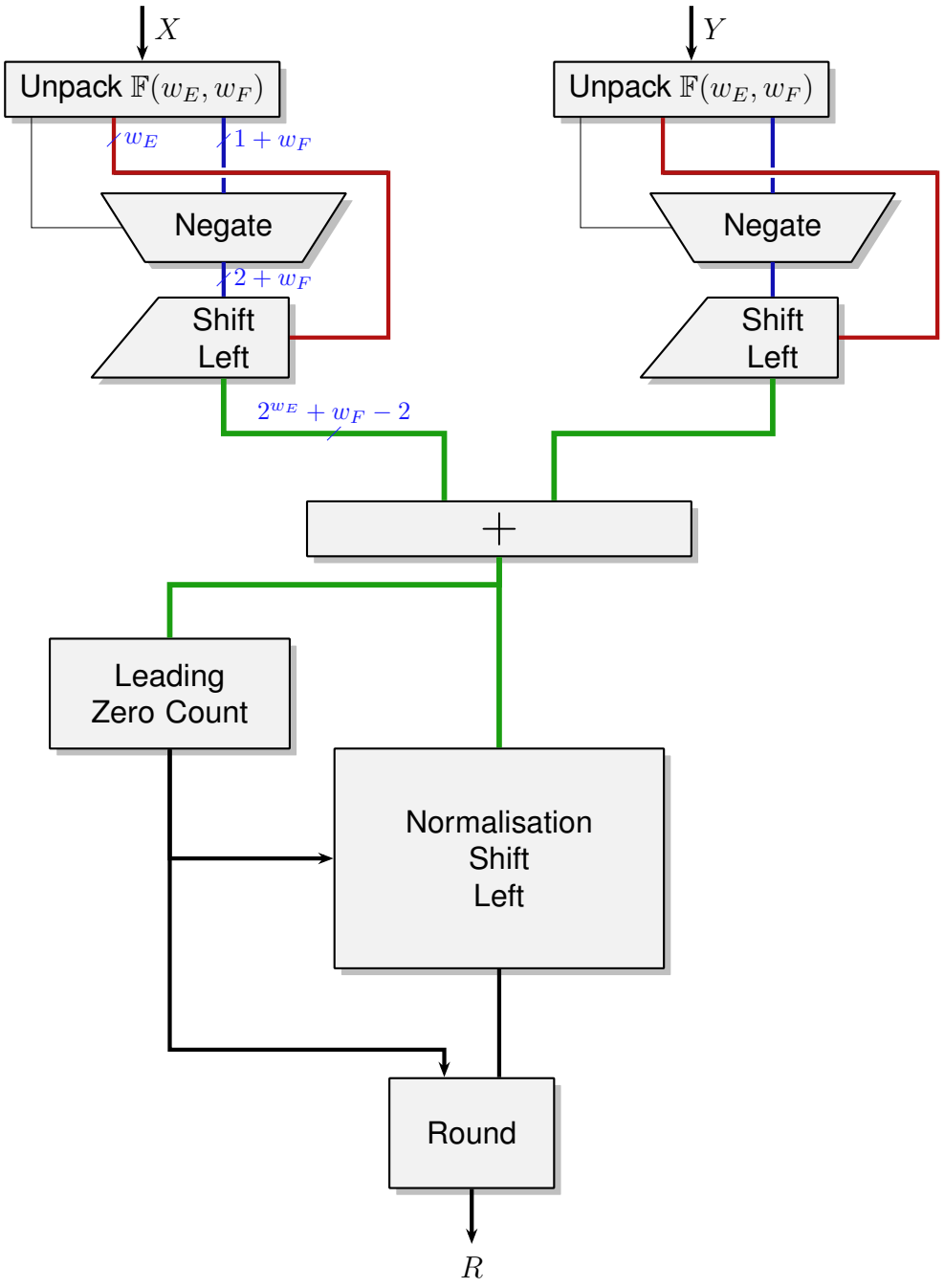


Figure 5.5: Architecture summing two floating-point numbers in $\mathbb{F}(w_E, w_F)$ using its expansion in the fixed-point encoding $\mathrm{sFix}(2^{w_E-1}, -2^{w_E-1} + 2 - w_F)$.

Format	Product			
	mult. size	LSB	MSB	Full-Size w_p (bits)
FP16	11×11	-48	30	80
BF16	8×8	-266	254	522
FP32	24×24	-298	254	554
FP64	53×53	-2148	2046	4196

Table 5.1: Size of fixed-point formats for a product.

This conversion to a common fixed-point format (or alignment) requires shifting left P_{sig} by $P_{\text{exp}} - P_{\text{exp,min}} = P_{\text{exp}} - 2$ bits, since the smallest biased exponent is 1 since subnormals ($E = 0$) behave as if $E = 1$. Alternatively, this is equivalent to shifting right P_{sig} by $w_p - P_{\text{exp}}$ bits. In both of those cases, the shift also performs a sign extension. Once converted to fixed-point, the terms can be summed using integer addition into a full-size accumulator (Fig. 5.6). Multiplications, conversions and additions are all exact. The result may then be normalised and rounded (only once) to the output format.

This approach is quite efficient for formats with small full-size accumulators, so has been used in particular for FP16 FMA [7]. It can also be extended to add more than one FP16 product [8]. As the sum of products is exact, it is associative and can be parallelised without any consideration of the exponent values.

5.2 Dot Product Operators

Deep Neural Networks can be broken down into multiple layers, which have different computational steps (Fig. 5.7).

In this chapter, the dot product computation is broken down into three steps, where the inputs X_i, Y_i are multiplied and added to a full-precision fixed-point accumulator Z, that is then saved in memory. This allows multiple exact accumulations to be carried out. Full precision accumulation is also required by the posit standard to compute dot products [54].

Once accumulation is complete, the fixed-point accumulators need to be converted to FP32 for use by non-trivial activation functions. Then, the results of the FP32 activation functions may be compressed back to a 8-bit floating-point representation.

5.2.1 Dot-Product-and-Add

Unpacking

The first step is to unpack the multiplicands into the internal format, using either the floating-point or posit unpack depending on the input format.

For posits, the range of exponent of the intermediate representation is much larger than the actual range of the posit format. For example the smallest Posit8.2 product is 2^{-48}, while the smallest exponent of the internal format is 2^{-62}. The unpacking step adjusts the bias of the internal format for posits such that the exponent of the smallest posit is encoded with the exponent 0.

NaN and ∞ Values

A full-precision dot product cannot generate ∞ as the format is by definition large enough so that no overflow is possible. NaN can only be generated by operating on input ∞ values ($\infty - \infty$ or $\infty \times 0$). In the operator output, any ∞ or NaN flags an invalid result.

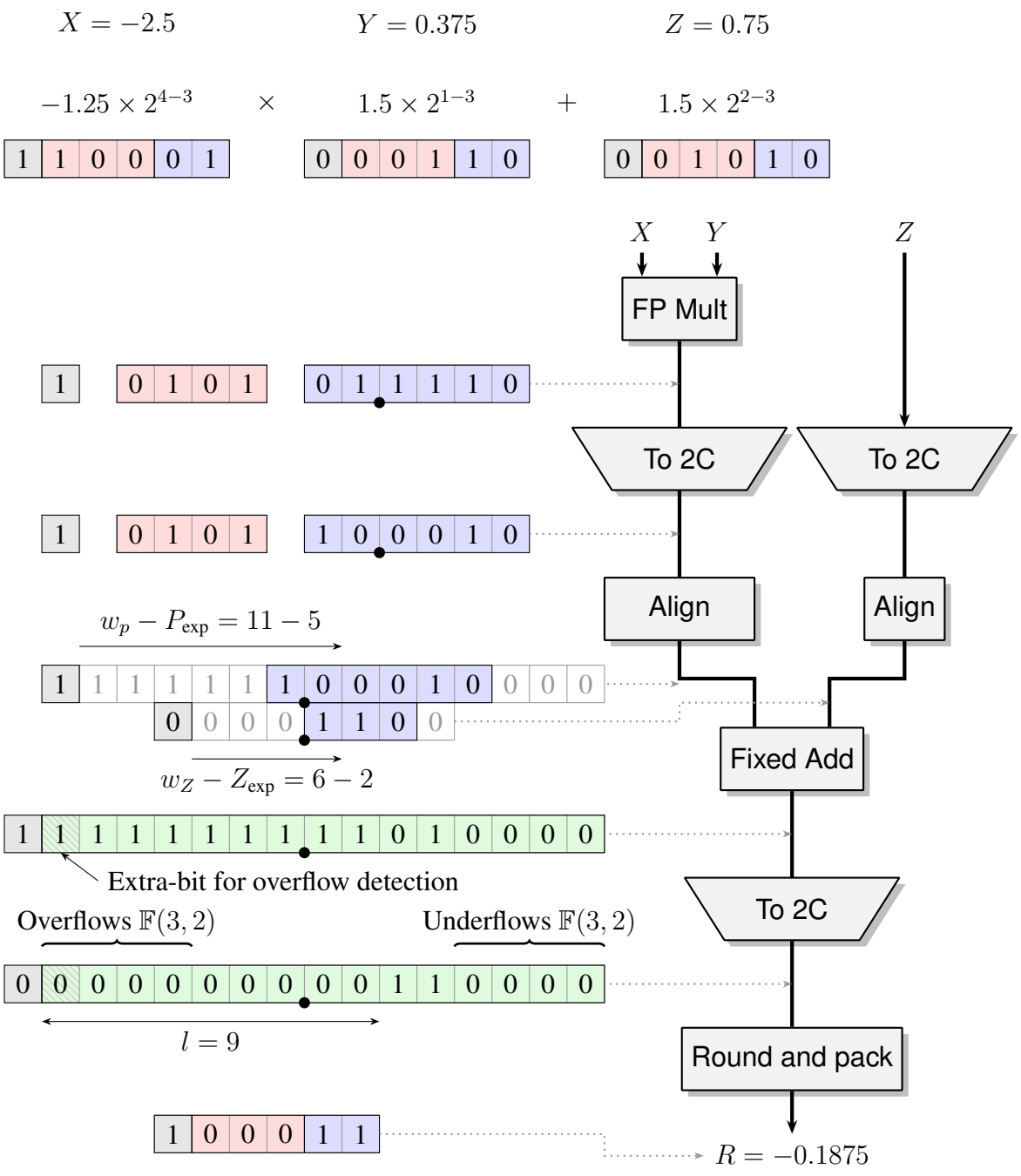


Figure 5.6: Example of alignment for the FMA using a full-size fixed-point format. To 2C refers to converting the sign-magnitude significand into two's complement.

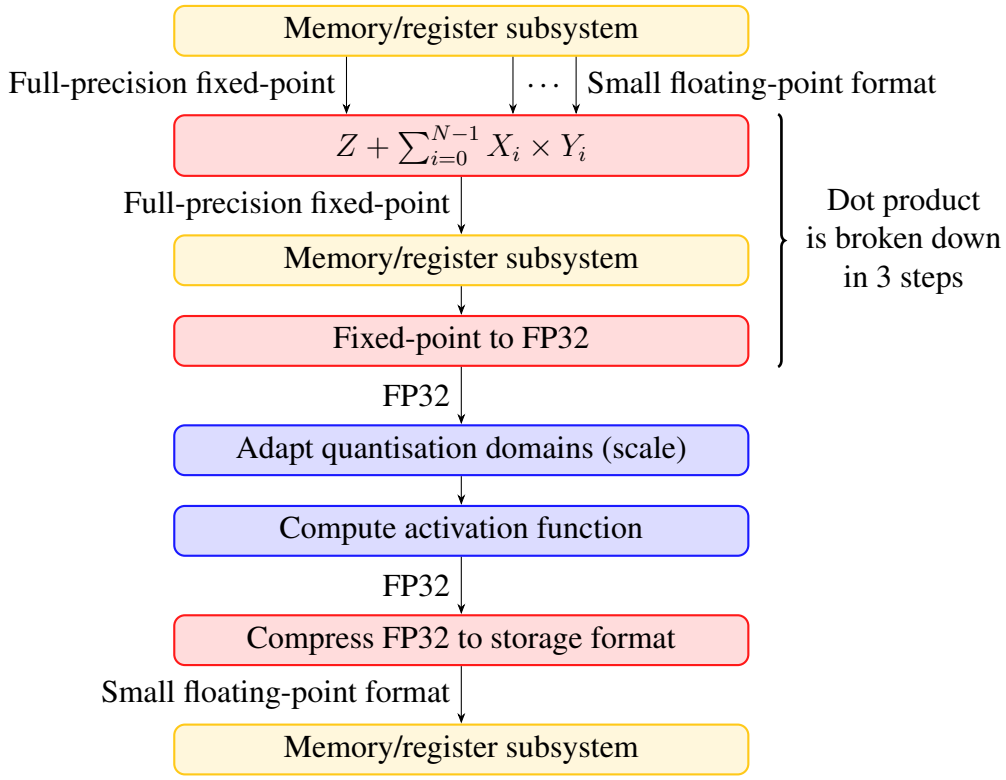


Figure 5.7: Architecture of steps for ML. In red the operators addressed in this chapter.

In the specific case of Machine Learning applications, distinguishing between ∞ and NaN is useless. This concurs with the fact that the E4M3 from the OCP standard [103] does not encode ∞. Accordingly, when unpacking input, the ∞ and NaN flags are converted to a single error bit which has the same meaning as the posit NaR [54].

A flag bit is used to encode an error in the accumulator result (and input Z). It is raised if the error flag for the input accumulator Z is raised, or if any of the multiplicand was ∞ or NaN. It is also raised if the sum overflows the accumulator, which may happen when the operator is used sequentially more times that it was designed to be used. Here, there is no risk of overflowing the accumulator when less than $D = 4096$ products are added, which corresponds to using the operator $\frac{D}{N}$ times sequentially.

Product alignment

Multiplication

The multiplication of the significands and their alignment into a w_p-bit fixed-point data path is carried out in the same way for all the floating-point formats. The only difference are the size of the different bit vectors, depending on the format (Tab. 5.2). Each format has a different size of significand, leading to a different multiplier size.

The multiplication is carried out using the multiplier component previously described, that computes the product without loosing accuracy.

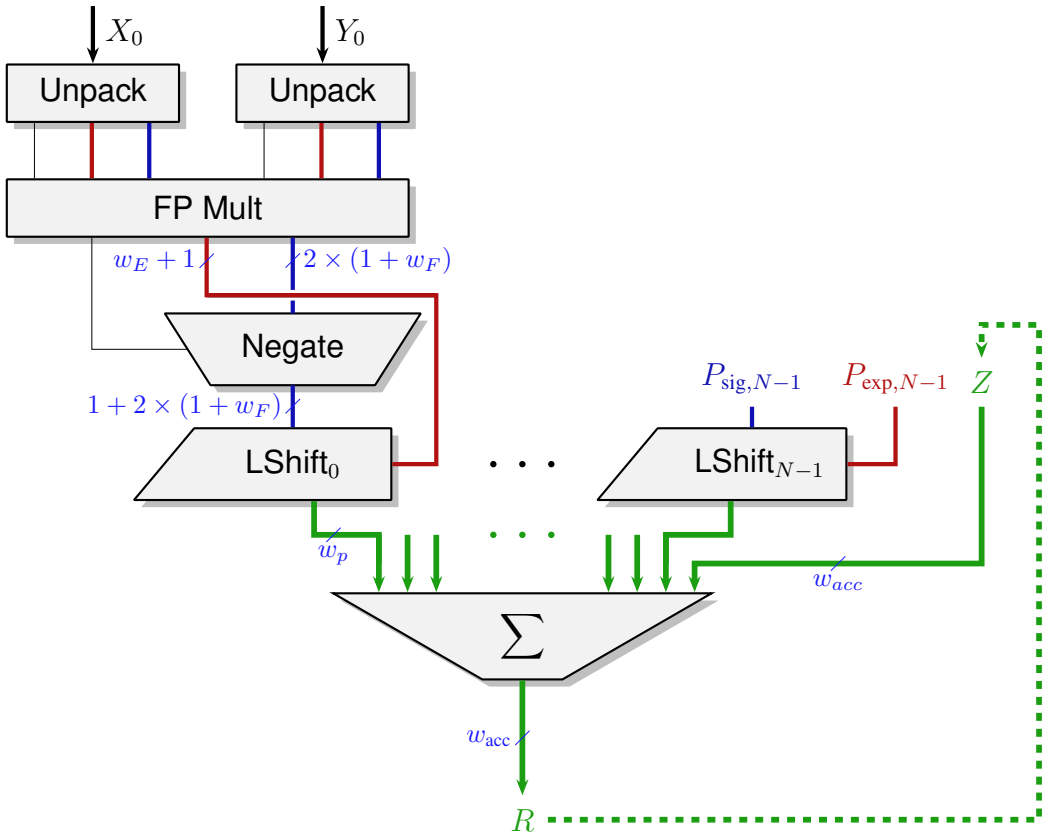


Figure 5.8: Architecture summing N floating-point products to a full-precision fixed-point accumulator Z.

Format	Product				Accumulator	
	Mult. size	LSB	MSB	w_p (in bits)	MSB	w_{acc} (in bits)
int8	8×8	0	15	16	31	32
E4M3	4×4	-18	16	36	44	63+1
E5M2	3×3	-32	30	64	94	127+1
Posit8.0	6×6	-6	6	26	50	63+1
Posit8.1	5×5	-24	24	50	38	63+1
Posit8.2	4×4	-48	48	98	78	127+1
Posit8.3	3×3	-96	96	194	158	255+1
FP16	11×11	-48	30	80	78	127+1

Table 5.2: Multiplier sizes, product sizes and accumulator sizes for the exact dot product accumulate operator.

Alignment

Each format has a different range, leading to a different size of full-precision fixed-point format:

- For the IEEE-754 floating-point formats: sFix$(2 \times (2^{w_E-1} - 1) + 1, 2 \times (2^{w_E-1} - 2 + w_F))$.
- For the E4M3, which encodes normal values on the largest binade: sFix$(2 \times 2^{w_E-1} + 1, 2 \times (2^{w_E-1} - 2 + w_F))$
- For the Posit formats, the format is computed with minPosit and maxPosit described in the standard [54].
- For integers, no alignment is needed, the product is on sFix$(15, 0)$.

The accumulator Z is of size $w_{\text{acc}} > w_p$. The exact size depends on the number of products D added, as the fixed-point format is extended with $\lceil \log_2(D) \rceil$ extra bits on the left to absorb the possible overflows. For ML applications, at least 12 bits were added to accommodate the $D = 4096$ products. This number is then rounded up to the next power of two to match standard data type sizes. Programming from high-level languages requires that the accumulator be loaded to and stored from memory, likely using the wide memory access instructions which are usually provided for the SIMD data types; this motivates that the accumulator size be a power of 2. Loading and storing the accumulator also enables to parallelise the dot product computation across several processing cores while ensuring that a final reduction across cores yields the exact result.

The posit quire [54] encodes NaR (the error value) as the sign bit set and all other bits cleared. This makes the decoding of an error value pretty expensive, as a logical OR must be carried out on nearly the whole accumulator size. In order to save logic when converting the accumulator contents back to FP32, all architectures presented in this chapter use a separate error flag, that is concatenated to the accumulator when saving them to memory.

Accommodating the Posit Format

The smallest Posit number has no fraction bits, as all the encoding space is used by the regime. When aligning the posit (Fig. 5.9), the product is shifted by the value of the exponent, and the $2 \times w_F$ least significant bits are removed from the aligned product, as they cannot be 1.

Sum

The fixed-point aligned products are summed with a bit heap compressor tree, using a FloPoCo implementation [81]. The resulting fixed-point accumulator R can then be saved into the memory subsystem, either to be reused as Z in further dot products or converted to FP32.

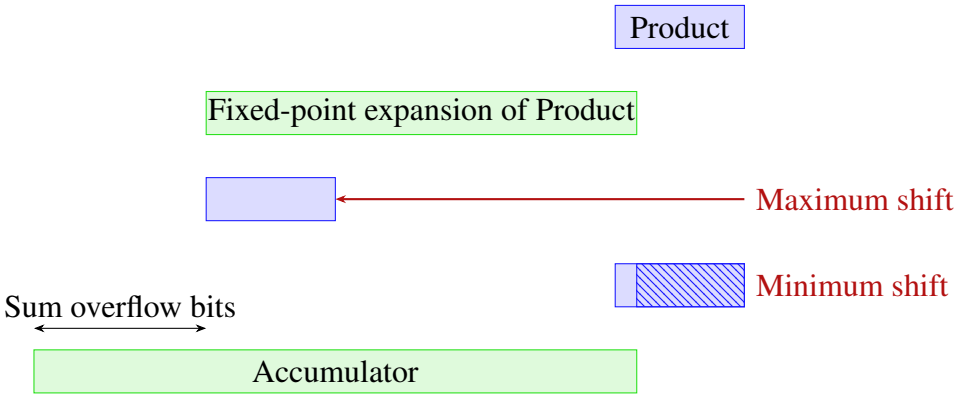


Figure 5.9: Alignment of the posit product.

5.2.2 Full-precision Fixed-point to FP32 Conversion

The FP32 format is useful for the subsequent scaling and activation function computations. It is easier to move around than the larger fixed-point accumulators. The computations can also be carried out using the FP32 operators of the general purpose core.

The largest of the proposed accumulator is sFix$(158, -96)$, while the FP32 range is uFix$(127, -149)$. Therefore this conversion cannot underflow FP32, and does not need to be able to create subnormal results. However, it could overflow the format.

This operator is a floating-point round and pack basic operator. The main modification is that the error flag is used to detect and output a NaN value. Most of the logic in this operator is dedicated to the Normaliser (Fig. 4.14).

5.2.3 Quantisation: FP32 to 8-bit Format Conversion

This operator is used after the activation function.

When converting to a floating-point or posit formats, it is composed of an unpacking, followed by a round and pack operator.

There is no need for a normaliser since the range of FP32 is so much larger than the range of the 8-bit formats. FP32 subnormals will always be rounded to 0 for floating-point, or minPosit for posits.

When converting to an INT8 format, the input is unpacked, shifted with an 8-bit shifter, and rounded as a fixed-point.

5.3 Synthesis results

The organisation of Kalray's accelerator (see Sec. 3.2) allows 256 bits for each vector $(X_i), (Y_i)$. This memory limit fixes $N = 32$ for 8-bit formats, and $N = 16$ for 16-bit formats

5.3.1 Dot-Product-and-Add

Table 5.3 shows synthesis results for the exact dot-product-and-add operator. Each operator is synthesised with the Synopsys Design Compiler NXT for the TSMC

Format	# of products	Area (μm^2)	Power (mW)	OPs/W ratio
FP16	16	8040	4.12	0.4
INT8	32	4107	1.67	1.0
FP16	32	15482	7.84	0.21
E4M3	32	4896	2.73	0.61
E5M2	32	8266	4.67	0.36
Posit8.0	32	7188	3.93	0.42
Posit8.1	32	9222	5.19	0.32
Posit8.2	32	17821	9.85	0.17
Posit8.3	32	26217	17.23	0.1

Table 5.3: Synthesis results of the exact dot product accumulate operator, with a target frequency of $250\,\mathrm{MHz}$.

16FFC node with a target frequency of $250\,\mathrm{MHz}$, to simulate 5 cycles at $1.25\,\mathrm{GHz}$. Operations per watt figures are normalised relative to the int8 format as it is the most energy-efficient representation.

The operator for FP16 with $N = 32$ is added for comparison purposes, but its memory usage is too high for its integration in the accelerator.

The main takeaway from Table 5.3 is that the largest FP8 format E5M2 has twice the performance compared to the baseline FP16 operator, for a 3% increase in area. It is this more energy efficient to implement separate FP16 and FP8 operators.

Another observation is that the operator with Posit8.2 multiplicands is not better than the one with FP16 multiplicands, whether on dynamic power or on area. This can be explained by the fact that the wider range of the Posit8.2 values compared to FP16 leads to a larger fixed-point representation of shifted products (96 bits versus 80 bits from Table 5.2), which in turn implies wider shifters and compression tree. Effects of these increases in the fixed-point data path appear comparable to those of the decrease of the multiplier sizes.

The cost of the decompression from the Posit format to a sign-exponent-mantissa format also appears significant, having arguably a bigger impact on the hardware complexity than expanding to fixed-point representation. In particular, the Posit8.0 and Posit8.1 have a fixed-point representation of the same size as the E4M3 floating point, but have an extra cost of 47% for Posit8.0 and 88% for Posit8.1. The cost of Posit8.1 is even larger than E5M2 by 12%, even though E5M2 has an accumulator twice the size. Posit8.3 requires the widest accumulator, which leads to its poor energy efficiency.

5.3.2 Full-precision Fixed-point to FP32 Conversion

The accumulator to FP32 conversion operators are synthesised with Synopsys Design Compiler NXT for the TSMC 16FFC node with a target frequency of $1.25\,\mathrm{GHz}$, and were fully pipelined in 2 cycles (Tab. 5.4). The resulting power and area figures are $5\times$ to $10\times$ lower than the corresponding dot product operators. In a matrix-multiply accumulate, conversion operators are only used once per matrix element, compared to $\frac{D}{N}$ usages of the dot-product-and-add, further lowering their relative contribution to total energy consumption.

Formats	w_{acc} (bits)	Area (μm^2)	Power (mW)
Posit8.3	255	2488	3.81
FP16, E5M2, Posit8.2	127	1542	2.41
E4M3, Posit8.1, Posit8.0	63	743	1.22
INT8	32	427	0.74

Table 5.4: Synthesis results of the accumulator to FP32 conversion operator, pipelined in 2 cycles at $1.25\,\text{GHz}$.

5.3.3 Quantisation: FP32 to 8-bit Format Conversion

Format	Area (μm^2)	Power (mW)
FP16	117	0.22
INT8	99	0.18
E4M3	66	0.17
E5M2	65	0.16
Posit8.0	103	0.20
Posit8.1	122	0.21
Posit8.2	207	0.29
Posit8.3	110	0.21

Table 5.5: Synthesis results of the FP32 to low bit-width floating-point conversion operators, with a single-cycle latency (no pipelining) at $1.25\,\text{GHz}$.

The conversion from FP32 to the 8-bit format are synthesised with a target frequency of $1.25\,\text{GHz}$ (Tab. 5.5). None of the FP32 to low bit-width conversion operator need pipelining, while their power consumption and area appear one order of magnitude smaller than those of the accumulator to FP32 conversion operators.

5.4 Integration into Kalray products

This exploration further cemented the interest in a dedicated dot product operator for 8-bit formats instead of only decompressing the 8-bit storage format into FP16. Some details were still to be determined: which FP8 format to use ? Should the accumulator be exposed in the ISA ? This section addresses these questions.

5.4.1 Combined E4M3 and E5M2 operator

As research on FP8 machine learning progressed, it became apparent that both formats E4M3 and E5M2 were required for ML acceleration [112]. A common E5M3 format can be used to represent both E4M3 and E5M2 formats, and the same template of architecture can be used.

The cost of this operator is barely higher than an E5M2 operator.

5.4.2 Storage of the Fixed-Point Accumulator

Instead of storing the accumulator in the memory subsystem, that is accelerator registers, it can be stored in dedicated registers inside the accelerator circuit. Since those accumulators are not accessed by the user or moved around, it is not needed for the fixed-point accumulator to be a power of two. To accommodate the larger dot product size of transformer networks, 16 bits instead of 12 are added to the fixed-point accumulator, enabling to add around 65k products.

The error flags are also separated from the accumulator, as they are not computed exactly at the same pipeline stage.

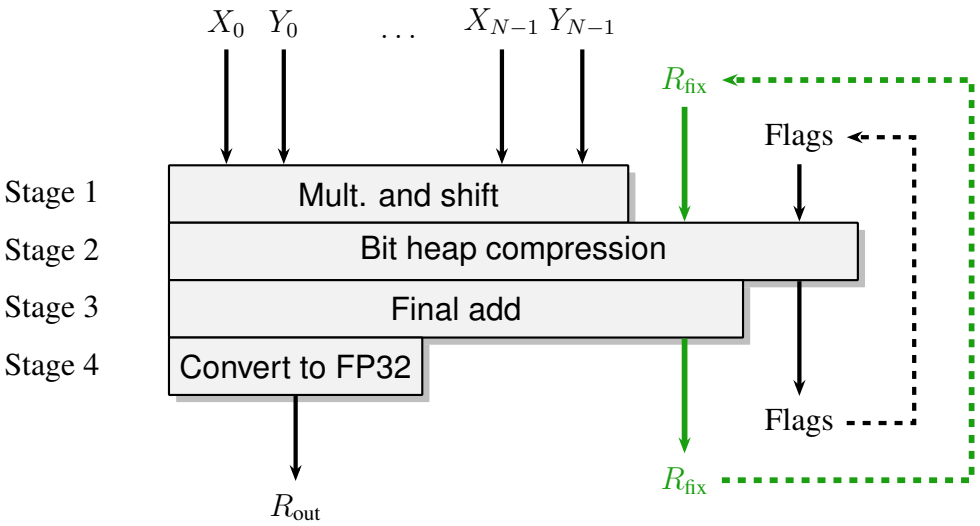


Figure 5.10: Pipeline stages of the FP8 dot product operator designed for Kalray.

The resulting dot product operator is pipelined in 4 cycles at $1.56\,\text{GHz}$ on the TSMC $4\,\text{nm}$ node. Instead of separating the dot product and the conversion to FP32, this operator links them together (Fig. 5.10), and enables early exit of the fixed-point accumulator R_{fix}. The last stage being only needed for the conversion, it can be clock-gated to save power when the dot product is still looping, and reactivated when the result must be converted to FP32.

Synthesis results are presented in the next chapter (Chap. 6), where they are compared to other similar dot product operators synthesised on the same node.

5.5 Conclusions

This chapter compared of the implementation cost of using various 8-bit number format for operators that exactly compute a 32-term dot product for 8-bit floating-point vectors and accumulate them in full precision into a large fixed-point accumulator. These dot product operators are complemented by two families of conversion operators, one for the conversion of the wide accumulators to FP32, and the other for the conversion of the FP32 values to the 8-bit floating-point formats.

Synthesis for the TSMC 16FFC node and a target frequency of 1.25 GHz exposes the significant advantages of using FP8 formats in terms of dynamic power and

implementation area.

Supporting Posit formats is expensive, and they are a very commonly used format, which was detrimental when choosing which formats Kalray will implement. The industry moving towards the support of both FP8 formats was also a decisive factor when shaping the final operator for Kalray's accelerator (Fig. 5.10).

Chapter 6

Relaxing Exactness for Dot Product on Larger Precisions

The dot product operators (Fig. 6.1) presented in this chapter have a similar structure to the ones described in the previous Chapter (Chap. 5).

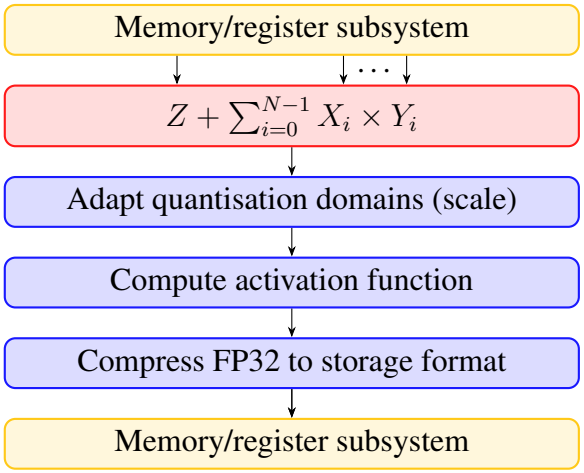


Figure 6.1: Architecture of steps for ML. In red, the operators addressed in this chapter.

The only difference is that the accumulator Z and the result R are floating-point numbers, which introduces logic to unpack Z and pack R (Fig. 6.2). If the operator is used multiple times sequentially, the conversion to FP32 at every step will worsen accuracy compared to architectures from the previous chapter.

This chapter will first present various architectures for the FPΣ component (see Sec. 6.1)

It then expands (see Sec. 6.2) on these designs to enable the reuse of architectures designed for machine learning formats (FP16, BF16) to compute dot product operations with FP32 multiplicands. The core idea is to adapt the principles of double-word arithmetic [96] to linear algebra. This method is commonly used in software [46] when the available hardware is not precise enough for a specific application.

6.1 State-of-the-Art Dot Product Operators

The full-precision fixed-point accumulation presented in Chapter 5 can be used to implement FPΣ. It needs to be modified by adding unpack and pack components such that Z and R are floating-point numbers.

This section presents other methods of implementing FPΣ base on classic floating-point addition.

6.1.1 Floating-point Addition: Relative Alignment

Two terms

The classic way to compute a floating-point addition of two numbers is to align the significands on the one with the largest magnitude [34] (Fig. 6.3). The size of the

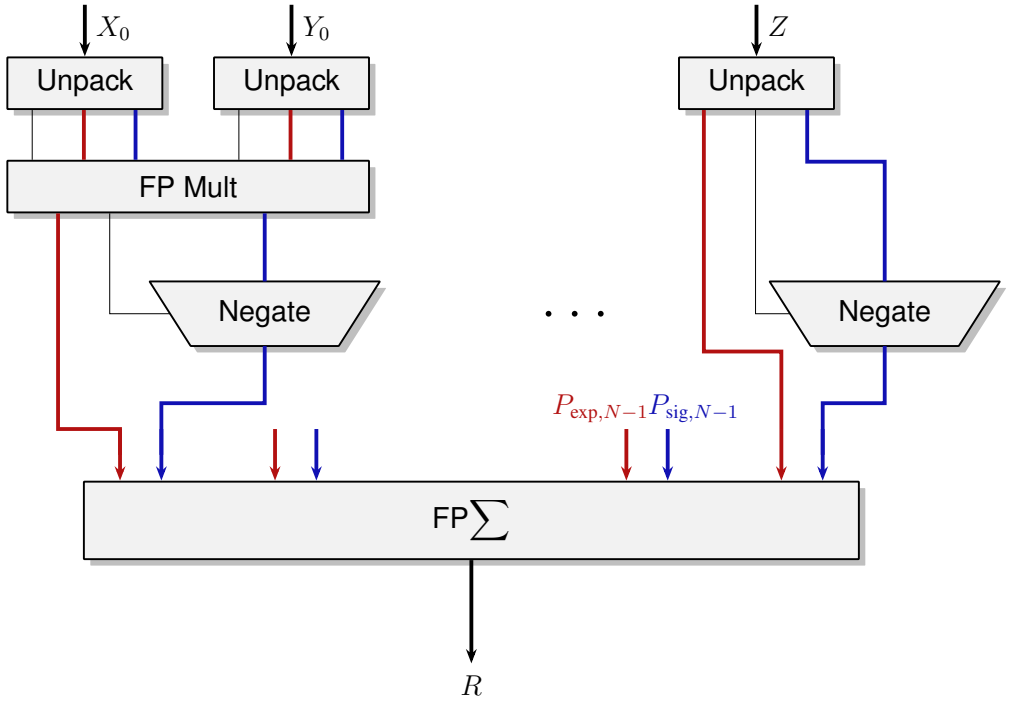


Figure 6.2: High-level architecture of dot products operators.

adder is about the size of a significand, which is much small than the size of the equivalent full-precision fixed-point. The significand with the smallest magnitude is shifted right relative to the largest one, with the bits shifted out ORed into a sticky bit to be able to correctly round the result.

More than Two Terms

When adding more than two terms, for example three terms $R = X + Y + Z$, it is also possible to align all the terms compared to the one with the highest magnitude. However, it is not possible to use sticky bits in order to obtain a correctly rounded result with an adder small than the full precision.

The use of relative alignment and sticky bits may lead to two problems (Fig. 6.4).

The first one, *cancellation* (Fig. 6.4a), is the most devastating to the accuracy of the result. When $Y = -X$, the result $X - X + Z$ is exactly Z. However, if the magnitude of Z is much smaller than the magnitudes of X and Y, then the relative alignment method has completely lost the significand of Z, and returns $X - X + Z = 0$. This extreme case is generally referred to as *total cancellation*. *Partial cancellation* refers to when X and Y have similar most significant bits (with opposite sign), and adding them results in the cancellation of few bits.

The second issue, *multi-sticky* (Fig. 6.4b) causes issues when rounding numbers. When the magnitude of X is much larger than the magnitudes of Y and Z, the alignment will result in both Y and Z compressed in a sticky bit. Having two sticky bits indicating information was lost is not sufficient to round the result, as it is not clear if $Y + Z$ would be negative, positive, or even equal to zero. The sticky bit information is useless when aligning more than two terms, it is not necessary to compute it.

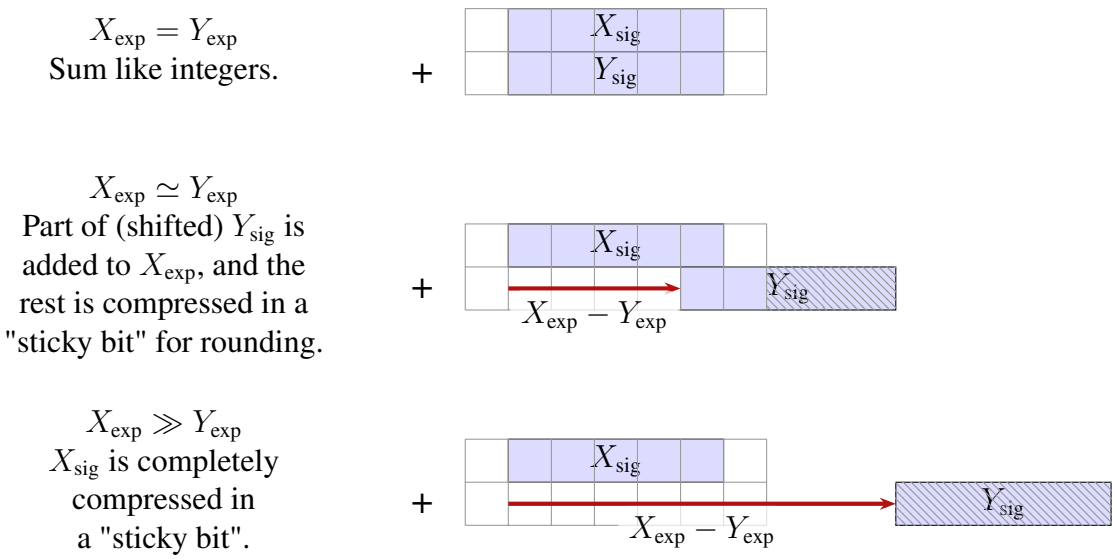
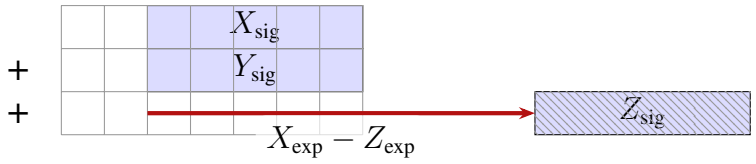
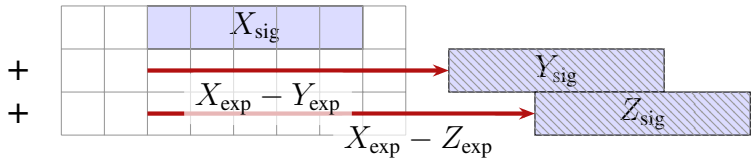


Figure 6.3: Alignment examples when adding two floating-point numbers X and Y, sorted such that $X_{exp} \geq Y_{exp}$.



(a) Cancellation.



(b) Multi-sticky.

Figure 6.4: Problematic cases when using classic floating-point addition to add three terms.

Despite all these problems, this method is sometimes used to add multiple floating-point numbers or products. It does not round the result correctly, as it is sensitive to cancellations and rounding issues.

6.1.2 Dot-Product Implementations

For $z = \circ(x_0 \times y_0 + x_1 \times y_1)$, the two products can simply be computed in parallel and added as a floating-point sum of two numbers [106].

Another operator [76] implements the operation $z = \circ(\circ(x_0 \times y_0 + x_1 \times y_1) + \circ(x_2 \times y_2 + x_3 \times y_3))$ with all intermediate roundings as one instruction. This dot product operator also computes multiple products in parallel.

The Matrix-Multiply-Add (MMA) units of mainstream GPUs [90, 75, 59, 60, 47] use a variant of the floating-point addition architecture: All products are aligned relative to the one with the largest magnitude then added to a floating-point accumulator that can be of an arbitrary size, the popular choice being 24 bits. This method will be referred to as the *truncated floating-point* accumulation.

6.1.3 Existing Kalray Dot Product Architecture

In the FP16 mixed-precision FMA operator of [7], an FP16 product $A \times B$ is first exactly converted to a fixed-point number, which is then added with correct rounding to an FP32 addend Z. The fixed-point format that contains all FP16 values has a Most Significant Bit (MSB) position of 30 and a Least Significant Bit (LSB) position of -48, hence the total size 81 bits. The operator of [7] implements the addition between the FP32 addend and the fixed-point product in a way that ensures a correctly rounded result to FP32, with fewer bits than a Long Accumulator matching the FP32 range (which would be 277 bits, from MSB 127 down to LSB -149).

The FP32 addend is first shifted relative to the FP16 accumulator. The size of the shifted addend is about 130 bits: $81 + 24 + 24$ since the significand can be placed entirely on each side of the FP16 accumulator, separated with a couple guard bits. Fig. 6.5 (not to scale) summarises the main alignment cases in this addition.

- Case 1 in Fig. 6.5 is similar to the way classic floating-point addition is computed. The FP32 addend is shifted relative to the FP16 accumulator and shifted into a sticky bit if needed (Case 1').
- In Case 2 of Fig. 6.5, the FP32 addend has a larger exponent than the MSB of the fixed-point FP16 product. The result of the multiply-add is the FP32 addend, possibly modified only with a rounding contribution from the fixed-point product. The alignment of the FP32 addend is limited to its LSB being one bit larger than the MSB of the FP16 product. As the guard bit stays clear, this allows for a correctly rounded sum.

The two numbers are added, and the FP32 exponent is used as the reference exponent for the result.

The Leading Zero Count (LZC) of the sum (the lower lines of Fig. 6.5) is computed, and then there are three cases to consider to determine the output exponent:

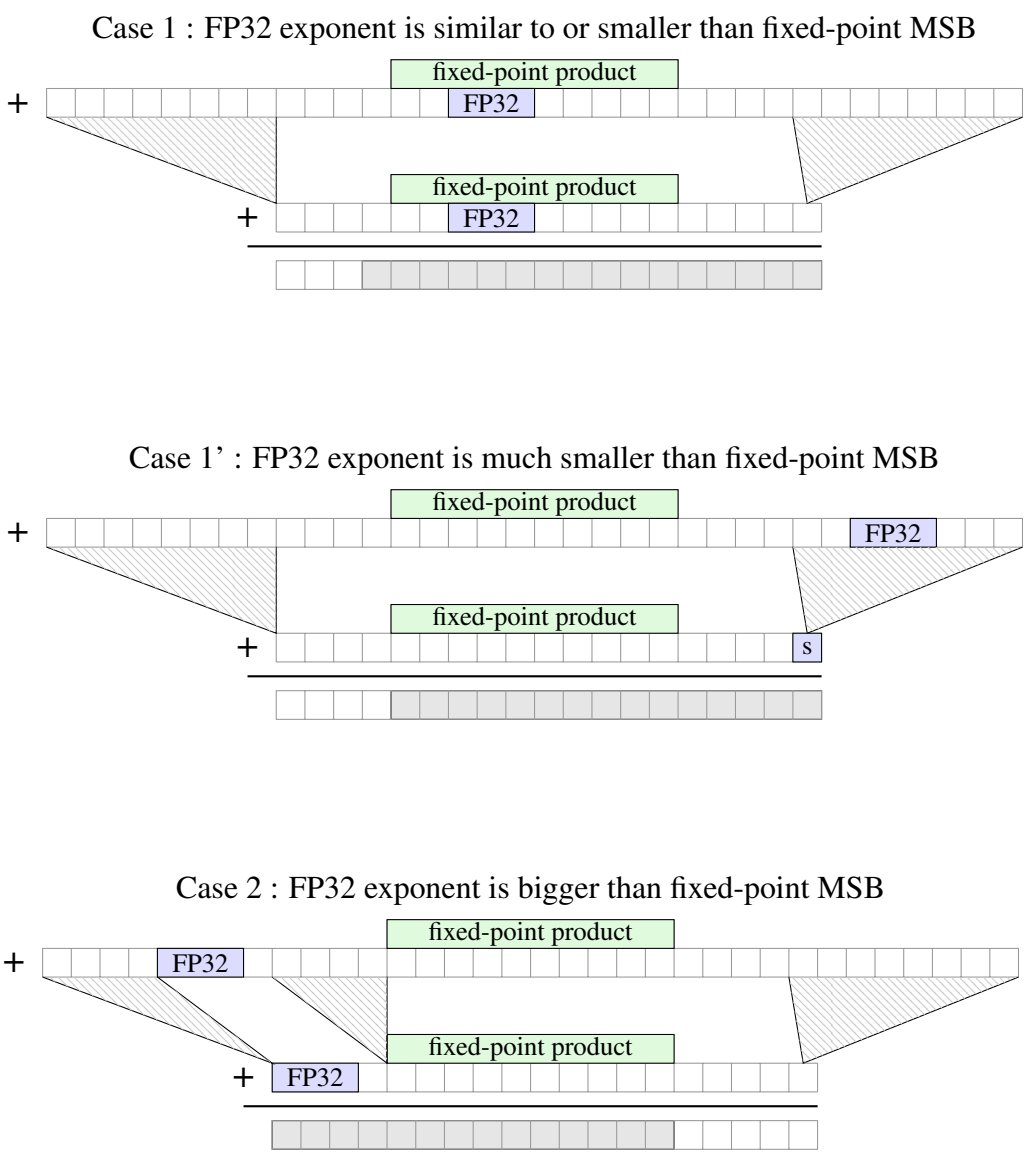


Figure 6.5: Addition of an FP32 number to a fixed-point (30, -48) number [7].

- In case 1, the exponent of the result is $30 + 25 - \text{LZC} + \text{FP32bias}$, where $30 + 25$ corresponds to the (unbiased) weight of the MSB of the sum.

- In case 1', when the FP16 product is 0 (which can be determined from the LZC), the FP32 addend which has been rounded off in the alignment must be restored by wiring it from the input to the output without modification.

- If the exponent of the FP32 addend is larger than $30 + 25 + \text{FP32bias}$, it is used as the result exponent (Case 2).

In the first and last cases, the significand is the normalised and rounded result of the sum.

The minimum exponent of an FP16 product (-48) is within the range of normal FP32 numbers. The only way this operator can produce a subnormal result is if the FP16 accumulator is 0 and the FP32 addend Z is subnormal, so it is wired unchanged to the output (second case). Thus, there is no need for the rounding logic to handle subnormal outputs.

This method can be extended to a $\text{DP}_{\text{FP16}}\text{A}$ operator, as the sum of several fixed-point FP16 products is still a fixed-point number. The fixed-point format must be extended with a few bits to absorb possible overflows. This $\text{DP}_{\text{FP16}}\text{A}$ operator architecture was described in [8].

A similar architecture is described in [88] where the sum with the FP32 is adapted to include a scaling factor to the multiplicands.

6.2 Double-Word Arithmetic in Accelerators

6.2.1 Double-Word Floating-Point Arithmetic

A double-word number [23] X is defined as the unevaluated sum of two floating-point numbers X^h and X^l such that $X = X^h + X^l$ and $X^h = \circ(X)$ using round to nearest. Thus X^l represents the signed rounding error of X: $X^l = X - \circ(X)$.

The exact result of floating-point additions or multiplications can be represented as a double word, which is useful for error-free transforms [96]. Hardware operators that input two floating-point numbers and output the exact sum or product as a double word have been studied in [32].

In a binary floating-point format with p bits for the fraction, the significand has $p + 1$ bits thanks to the implicit bit. Interestingly, a double-word decomposition can always represent at least $2p + 3$ consecutive bits: $p + 1$ bits for X^h, $p + 1$ bits for X^l, and an extra bit encoded in the sign of X^l as follows:

- If X was a midpoint between two exact floating-point values, then only $p + 2$ bits are needed to represent its significand.

- Otherwise, X^l is strictly smaller than the half-ulp [96] of X, therefore there is a gap of at least 1 bit between X^h and X^l.

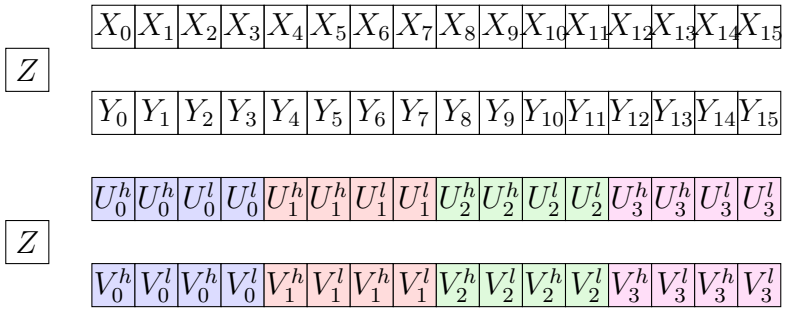


Figure 6.6: FP16 dot product of size 16 $R = Z + \sum X_i \times Y_i$ (top), used as an FP32 dot product of size 4 (bottom).

6.2.2 Double-FP16 for Linear Algebra

This technique [46] can be used for dot products operations (Fig. 6.6).

To accelerate an FP32 matrix multiplication $C = AB$ using an FP16 MMA unit, A and B are decomposed as the unevaluated sum of FP16 matrices $A \simeq A^h + A^l$ and $B \simeq B^h + B^l$. A mixed-precision MMA then computes an approximation of the FP32 matrix C as $C \approx A^h B^h + A^h B^l + A^l B^h + A^l B^l$. The decomposition of an FP32 matrix A into FP16 matrices A^h, A^l is obtained by [90]:

$$\begin{aligned} A^h &= \textbf{toFP16}(A) \quad , \\ A^l &= \textbf{toFP16}(A - \textbf{toFP32}(A^h)) \quad . \end{aligned} \tag{6.1}$$

The **toFP16()** operation converts each FP32 element to FP16. Likewise, **toFP32()** converts each FP16 element to FP32.

This approach has several drawbacks:

1. The dynamic range of the FP16 representation leads to extreme precision losses for large FP32 values [90], since the 8-bit exponent of the FP32 representation is larger than the 5-bit exponent of the FP16 representation. In other words, (6.1) often over/underflows.

2. Even when the FP32 multiplicands fit in the dynamic range of the FP16 format, there is still a loss of precision as the number of bits in two FP16 significands $(2 \times (10 + 1))$ is strictly less than in an FP32 significand $(23 + 1)$. In other words, (6.1) is usually inexact.

3. On GPUs, an additional loss of precision arises from the addition of sub-product matrices $A^h B^h, A^h B^l, A^l B^h, A^l B^l$ in FP32 arithmetic, which in the case of NVIDIA Tensor Cores only supports round to zero [90].

4. The explicit decomposition of the FP32 multiplicand matrices A, B into the FP16 matrices (A^h, A^l, B^h, B^l) increases the complexity of application software.

A solution to issues 1 and 2 is to use triple-BF16 [58], but it makes issues 3 and 4 worse. The motivation of the present work is to address all these drawbacks in hardware.

6.2.3 The TF32 format of NVIDIA Tensor Cores

Tensor Cores in NVIDIA GPGPUs are mixed-precision fused MMA units [90]. They multiply matrices with FP16 elements and add the products to a matrix with FP32 elements. Since the NVIDIA Ampere architecture, they also support other multiplicand formats, including BF16 and TF32 [120], a combination of FP16 and BF16 formats (Fig. 6.7).

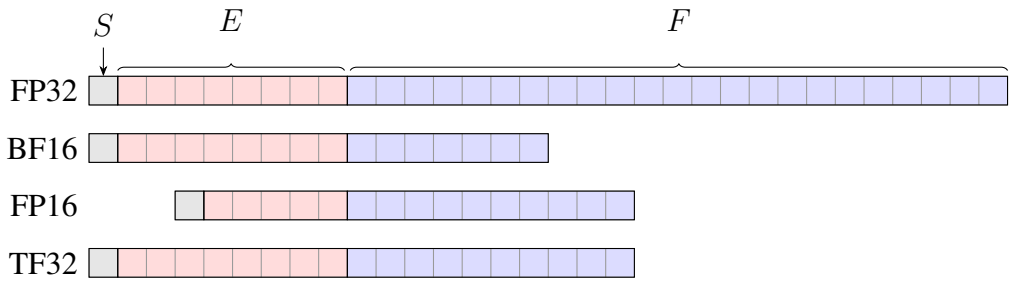


Figure 6.7: Floating-point formats supported by NVIDIA Tensor Cores [120].

Although designed to accelerate deep learning kernels, Tensor Cores are also used in numerical analysis to improve the performance of matrix multiplications in FP32 arithmetic. One approach is to rely on iterative refinement techniques [56], while others adapt multi-word arithmetic techniques. In this setting, the TF32 format is appealing because it has the same exponent size as the FP32 format. This alleviates the dynamic range problem of double-FP16 decompositions of FP32 numbers. However, the limited accuracy of double-FP16 decompositions remains with double-TF32 decompositions.

6.3 Intermediate Floating-Point Format

This chapter presents the architecture of a fused Dot Product Add (DPA) operator that targets both the mixed-precision operations used by machine learning and the FP32 linear operations used in numerical computing. The main enabler is the definition of an intermediate floating-point format called E9S12 (9 exponent bits, 12 significand bits), which is used inside the operators to represent not only FP16 or BF16 multiplicands, but also an exact double-word decomposition of FP32 multiplicands.

On the implementation side, the starting point is a correctly rounded fused DPA operator with FP16 multiplicands and FP32 addend (see Sec. 6.1.3). This operator is extended to support E9S12 multiplicands, as this also enables the use of BF16 and FP32 multiplicands. The FP32 multiplicands are decomposed in hardware as double-E9S12. The resulting operator is still correctly rounded for FP16 multiplicands, and is IEEE compliant when emulating an FP32 Fused Multiply Add (FMA) operator. The proposed operator is not only faster, but also more accurate than software solutions based on decompositions of FP32 multiplicands into pairs of FP16 or BF16 numbers.

Specifically, for an integer N, let $(X_i)_{i\in[0,4N-1]}$, $(Y_i)_{i\in[0,4N-1]}$ be FP16 numbers, Z and R be FP32 numbers. The proposed DPA operator computes the sum of a dot product of X_i, Y_i and the addend Z with a single rounding $\circ$:

$$R = \circ(X_0 \times Y_0 + \ldots + X_{4N-1} \times Y_{4N-1} + Z) \quad .$$

This requires an internal accumulator of 81 bits (see Sec. 6.4.2). If the inputs $(X_i)_{i\in[0,4N-1]}$, $(Y_i)_{i\in[0,4N-1]}$ are BF16 numbers, the same precision is used to compute their dot product. This process and the corresponding truncation is noted ϕ. The DPA operator then computes

$$R = \circ(\phi(X_0 \times Y_0 + \ldots + X_{4N-1} \times Y_{4N-1}) + Z) \quad .$$

Finally, the same DPA operator may compute, for FP32 numbers $(U_i)_{i\in[0,N-1]}$, $(V_i)_{i\in[0,N-1]}$,

$$R = \circ(\phi(U_0 \times V_0 + \ldots U_{N-1} \times V_{N-1}) + Z) \quad .$$

This operator provides the building block of a large pipelined MMA accelerator, the details of which are beyond the scope of this work.

In order to operate on FP16, BF16 and double-word decompositions of FP32 multiplicands, the intermediate format E9S12 is defined to represent all of them and be suitable for implementation of the downstream calculations. This intermediate format is operational, and thus does not need to be frugal in bits, unlike a storage format. In particular, it does not need to be normalised. Moreover, conversion to this format should be easy to implement in hardware.

6.3.1 FP16 and BF16 Multiplicands

The generic *Unpack(w_E, w_F)* subcomponent (see Sec. 5.1.1) unpacks an input X in $X_{\text{sign}}, X_{\text{exp}}, X_{\text{sig}}, X_{\text{flags}}$.

Unpacking an FP16 number requires an Unpack(5, 10), outputting 5 exponent bits and 11 significand bits. Unpacking a BF16 number requires an Unpack(8, 7), outputting 8 exponent bits and 8 significand bits. Therefore, an unpacked format supporting both FP16 and BF16 numbers requires 1 sign bit, 8 exponent bits, 11 significand bits, and 5 bits for X_{flags}.

6.3.2 Decomposing FP32 Multiplicands

Using a hardware-friendly approach, the idea of double-word arithmetic can be adapted to reuse an FP16 / BF16 data path to compute an FP32 dot product.

An FP32 number U is decomposed into an unevaluated sum of U^h and U^l, that is, $U = U^h + U^l$. The key to making this decomposition hardware-friendly is to adapt the intermediate floating-point format to represent U^h and U^l (Fig. 6.8). Since an FP32 number has a significand of 24 bits, it should be split into two significands of 12 bits. There is no need for the rounding trick of Section 6.2.1 that requires $U^h = \circ(U)$. Instead, it is possible to just split the mantissa U_{sig} into its 12 most significant bits that become U^h_{sig}, and its 12 least significant bits which become U^l_{sig}

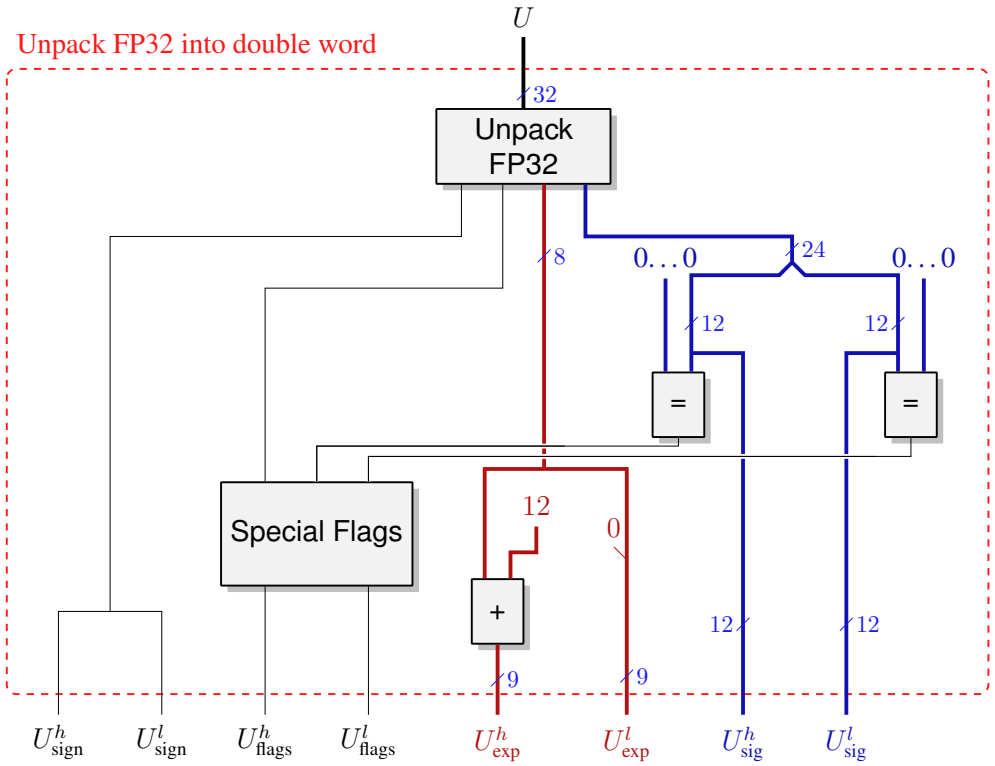


Figure 6.8: Unpacking an FP32 number into two E9S12.

(Fig. 6.8). Since the significands in the intermediate format may be not normalised, it is not necessary either to normalise U^l_{sig} if its leading bit is 0. The signs of U^h and U^l are the same (Fig. 6.8).

The exponent of U^l, U^l_{exp} is the exponent of U minus the constant 12, which is simple to implement. To ensure that 0 encodes the smallest possible exponent, the intermediate format uses an encoding with an exponent bias of $139 = 127 + 12$. Without this bias, negative exponents $U^l_{\text{exp}} = U_{\text{exp}} - 12$ would appear during the decomposition of subnormal FP32 numbers and those with exponent $U_{\text{exp}} < 12$.

Using the bias of 139, the hardware sets $U^l_{\text{exp}} = U_{\text{exp}}$ and $U^h_{\text{exp}} = U_{\text{exp}} + 12$. If the input U is subnormal, then $U^l_{\text{exp}} = 0$ and $U^h_{\text{exp}} = 12$. This bias change is compensated later in the operator when computing the final exponent. However, it requires one additional exponent bit in the intermediate format, hence the 9 in E9S12.

Special care is needed when dealing with this non-normalised, non-standard format. For example, U^l can be zero with a non-zero exponent. In general, this approach is simpler and more energy efficient than implementing (6.1) which enforces $U^h = \circ(U)$.

6.3.3 The Common Intermediate Format

The intermediate floating-point format that captures all these multiplicands is called E9S12 in this work. It is encoded using 27 bits: 1 sign bit, 9 exponent bits, 12 bits for the significand and the 5 flag bits. In other words, the format needed to decompose the FP32 numbers is also large enough for the unpacking of the FP16 and BF16 numbers as well.

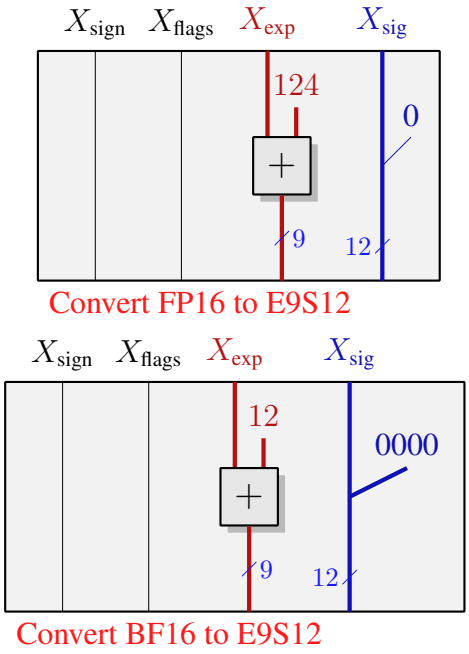


Figure 6.9: Converting an FP16 or a BF16 into E9S12.

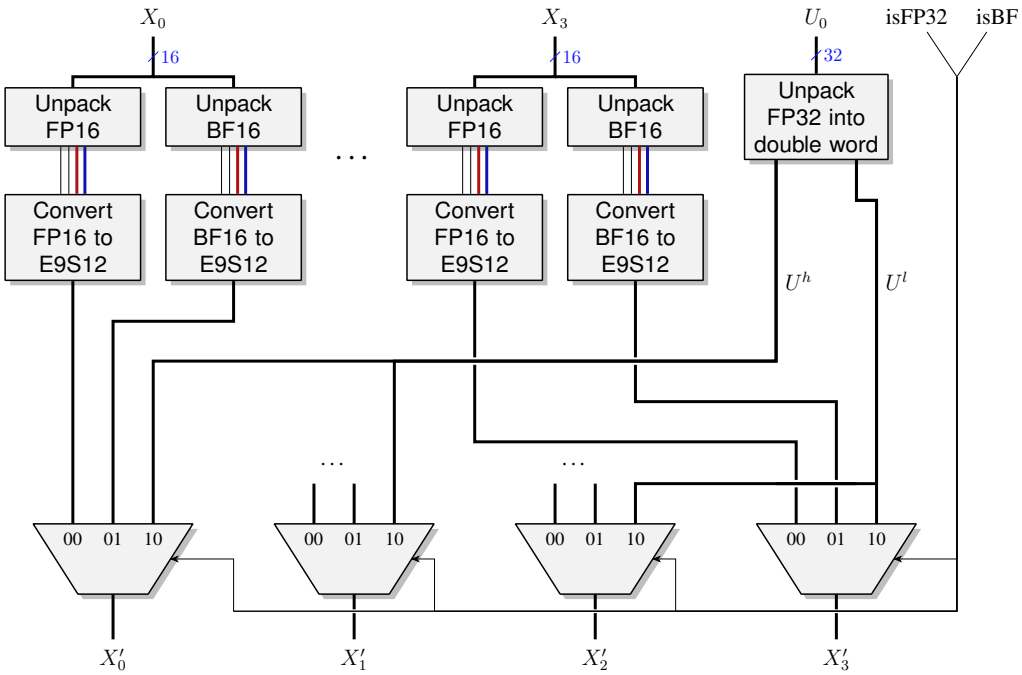


Figure 6.10: Unpacking an FP16, a BF16 or an FP32 number into E9S12.

As shown in Fig. 6.10, the outputs of the BF16 Unpack(8,7) and the FP16 Unpack(5, 10) are converted to the E9S12 format by trivial zero extension of the significand and update of exponent bias.

6.3.4 Intermediate Format Applications

The simplest application of E9S12 is to implement an FP32 Multi-Addition (MA) operator. Each FP32 input U is decomposed into $U^h + U^l$, each in E9S12 format. The MA$2N$ operator for E9S12 can be used as a MAN operator for FP32:

$$\begin{aligned} R &= \circ(U_0^h + U_0^l + \ldots + U_{(N-1)}^h + U_{(N-1)}^l) \\ &= \circ(U_0 + \ldots + U_{N-1}) \quad . \end{aligned}$$

When unpacking the inputs for the Multi-Addition, each FP32 input is decomposed into two E9S12.

However, the main application of E9S12 is to implement a DPA operator with FP16, BF16 or FP32 multiplicands, FP32 addend and FP32 result. In case of FP16 or BF16, each 16-bit multiplicand X_i, Y_i is unpacked into the E9S12 format (Fig. 6.10), and the DP$4N$A operator computes:

$$R = \circ(Z + X'_0 \times Y'_0 + \ldots + X'_{4N-1} \times Y'_{4N-1}) \quad .$$

In case of FP32 multiplicands U (resp. V) is decomposed (Fig. 6.8) into U_h and U_l in E9S12 format such that $U = U_h + U_l$ (resp. $V = V_h + V_l$). The product $U \times V$ is rewritten as:

$$\begin{aligned} U \times V &= (U^h + V^l) \times (U^h + V^l) \\ &= U^h \times V^h + U^h \times V^l + U^l \times V^h + U^l \times V^l \quad . \end{aligned}$$

The DP$4N$A for E9S12 multiplicands can then be used as a DPNA for FP32 multiplicands:

$$\begin{aligned} R &= \circ(Z + \sum_{i=0}^{N-1} (U_i^h \times V_i^h + U_i^h \times V_i^l + U_i^l \times V_i^h + U_i^l \times V_i^l) \\ &= \circ(Z + U_0 \times V_0 + \ldots + U_{N-1} \times V_{N-1}) \quad . \end{aligned}$$

The unpack circuit for this operator is computed in blocks of 4 16-bit inputs. A block is shown in Fig. 6.10 for X. In the corresponding block for Y, the outputs Y'_1 and Y'_2 are switched compared to this figure.

6.4 Dot-Product-Add Operators

The dot product operators in this section follow the high-level architecture of Figure 6.2. This section will detail the subcomponent FPΣ.

6.4.1 Baseline FP16 Dot Product Add Operator

This operator (Fig. 6.11) deals with the multiplicands in a very similar way to the architectures described in Chapter 5. An exact multi-addition of 81-bit numbers allows for correct rounding, reusing the techniques of [8] (see Sec. 6.1.3)

In the subcomponent FPΣ, the products $P_{\text{sig},i}$ are shifted by their exponent $P_{\text{exp},i}$ to obtain their fixed-point value. The sum of these fixed-point products is computed exactly by a fixed-point multi-adder.

The sum also includes the contribution of the FP32 addend Z, which after unpacking has a different format than the products. The relative shift of the fixed-point sums of products and the FP32 addend follows the principles of Section 6.1.3 [7]. Consequently, the conversion from the exact fixed-point sum to FP32 ensures a correctly rounded result.

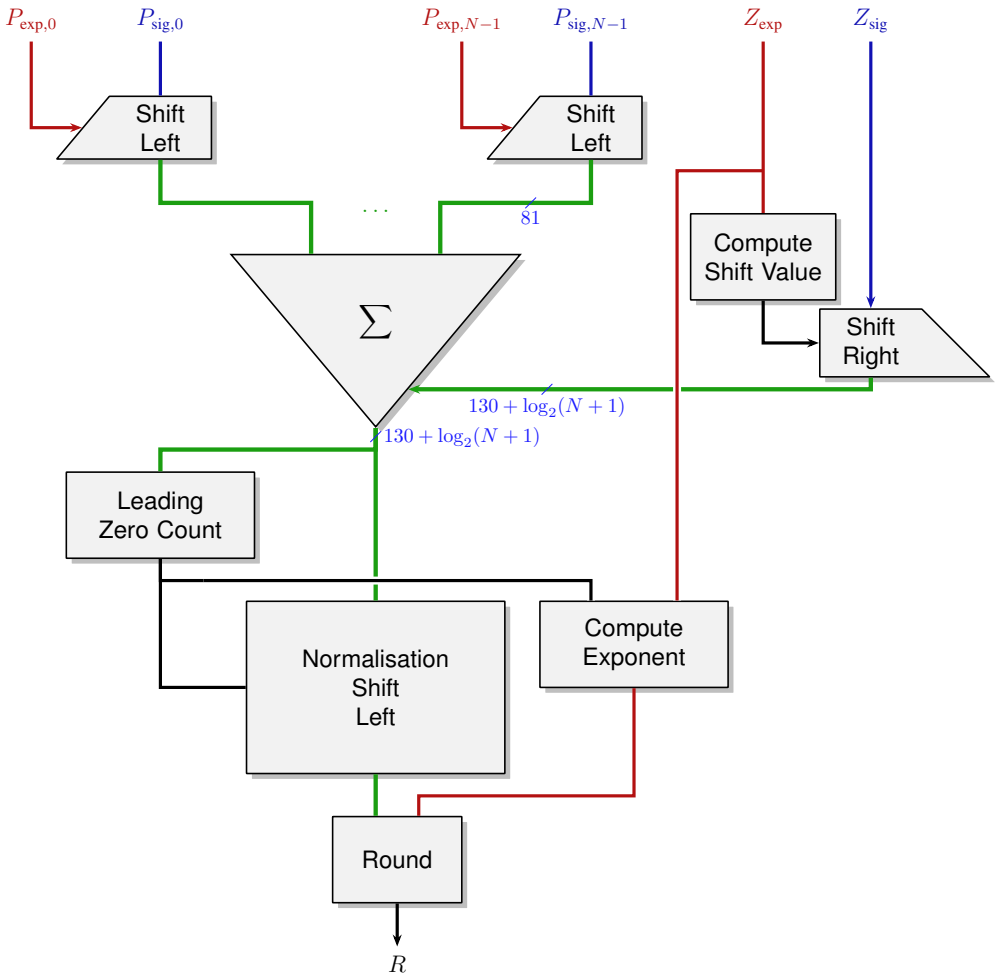


Figure 6.11: Architecture of a correctly-rounded FP16 Dot Product Add operator.

6.4.2 Dot Product Add Operator with E9S12 Multiplicands

The E9S12 format has a notably larger dynamic range than FP16, which would involve shifting and accumulating fixed-point numbers over 540 bits (MSB 254 and

LSB -298). This makes the full-precision fixed-point accumulation impractical, so it is replaced by a truncated floating-point method described in Fig. 6.12.

Architecture overview The products are aligned to the product with the largest exponent E_{max}. The accumulator is reduced to an arbitrary size L, but it is now needed to store its exponent E_{Acc}.

To avoid loss of precision in the case of partial cancellation, the sum size should be at least twice the size of the significand of the result (48 bits for FP32). The sum size was chosen to enable the computation of a correctly rounded FP16 Dot Product Add: $L = 81 + \lceil \log_2(N+1) \rceil$ bits. This has proven to be useful for formats with a small dynamic range [88]. Additionally, this larger accumulator results in additional precision for BF16 and FP32 compared to mainstream GPUs.

The handling of the FP32 addend is similar to a floating-point adder: The sum of products is indeed a floating-point number (possibly non-normalised) of exponent E_{Acc} and of significand size L. However, contrary to a classic FP adder, the proposed architecture always shifts the FP32 significand, since it is much smaller than L. This is actually similar to the addition performed in the baseline operator [7], with the notable difference that the shift amount of the addend Z is not computed from a constant (MSB 30 of the accumulator), but from the exponent E_{Acc}.

Finally, a Leading Zero Count and a Shift are performed to normalise the sum and prepare it for the final rounding.

Overhead over the baseline FP16 DPA Compared to the baseline implementation, the size of the multipliers increases to 12×12 to support E9S12. The accumulation size of $L = 81$ is kept to continue to allow for exact FP16 calculations, which implies that the alignment shifters and adder trees are the same size as in the baseline operator.

E_{max} must be computed with a comparator tree, it can be done in parallel with the multiplication of significands. As the significands are aligned with E_{max}, it is simpler to use a right shifter instead of the previously used left shifter. The significand product $P_{\text{sig},i}$ is shifted by $E_{\text{max}} - P_{\text{exp},i}$.

Subnormal handling As discussed in Section 6.1.3, the baseline FP16 operator does not need subnormal output logic, as the range for the FP16 product is much smaller than the range for the FP32 result.

However, with the E9S12 format, it becomes possible to create a subnormal output from normal inputs. Denormalisation logic is added to the operator, where the final shift computation takes into account the Leading Zero Count and Exponent. The shift performs an incomplete normalisation of the accumulator so that the significand is correctly aligned for the rounding.

Zero detection When splitting an FP32 number into two E9S12 numbers, it is important to independently detect if the two separate parts are zero. Although most flags are duplicated (i.e. $X = +\infty \rightarrow X^h = +\infty \wedge X^l = +\infty$), accurately detecting zeros helps catching bugs in the rest of the operator. The case $X^l = 0, X \neq 0$ can

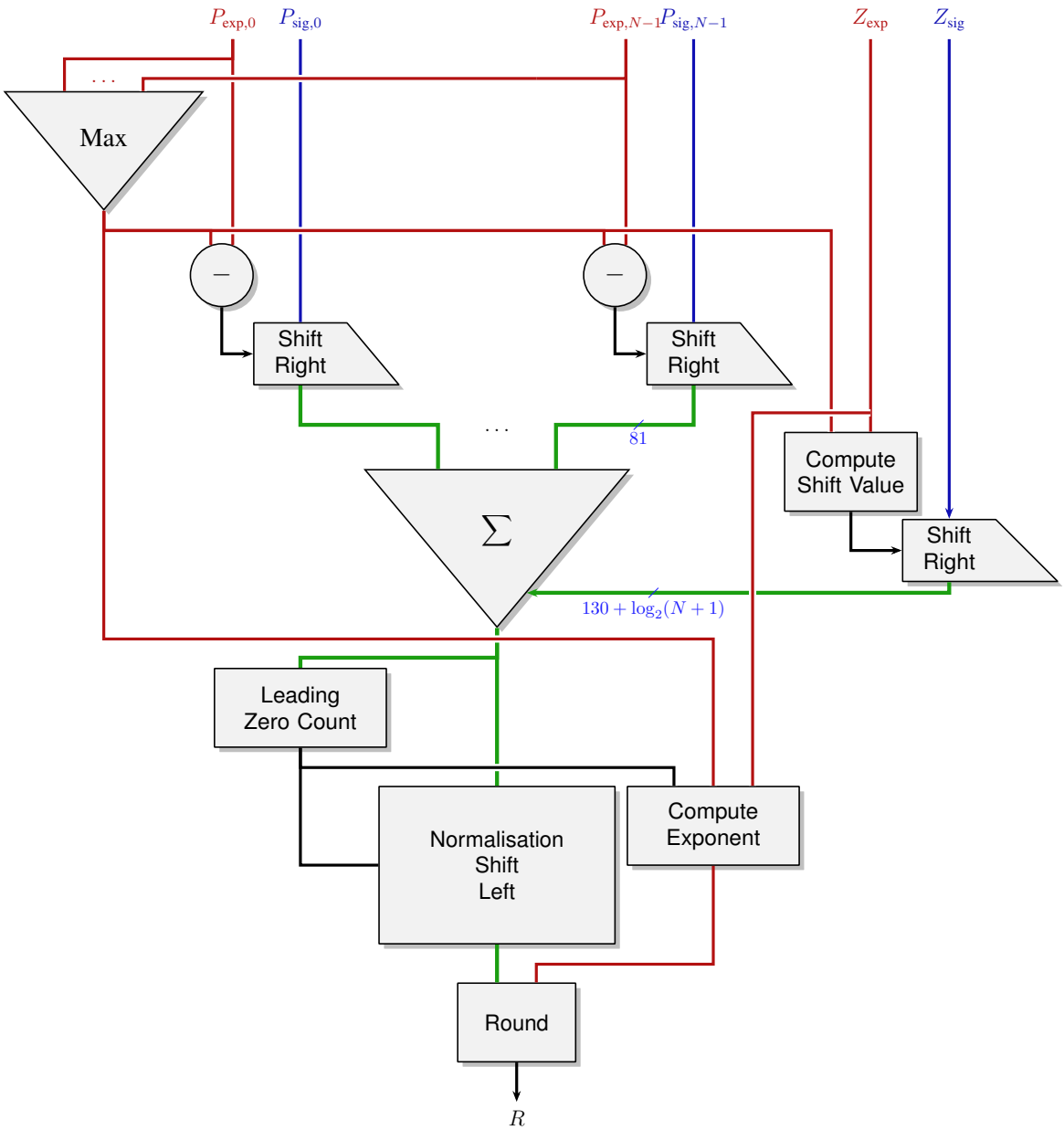


Figure 6.12: Dot product operator for E9S12.

happen when the fraction of X is empty on the lower half. The case $X^h = 0, X \neq 0$ can happen when X is a small subnormal.

6.4.3 Support of the FP32 Fused-Multiply Add

A dot product operator in the E9S12 format

$$R = \circ(\phi(X'_0 \times Y'_0 + \dots X'_{N-1} \times Y'_{N-1}) + Z)$$

as described in the previous section can be used to emulate a correctly rounded FP32 Fused Multiply-Add (FMA) operation.

If all FP32 products except one $U_k \times V_k$ are zero, the result is correctly rounded. Indeed, the product is split into four terms $U_k \times V_k = U_k^h \times V_k^h + U_k^h \times V_k^l + U_k^l \times V_k^h + U_k^l \times V_k^l$ but those terms have a maximum exponent difference of 24. Since the size of each term is also 24, the four products can be exactly represented on 48 bits (the size of the FP32 significand multiplication). Therefore, as soon as $L > 48$, no information is lost in the sum, and the product $U_k \times V_k$ is exact. Implementing the method of [7] then ensures the correct rounding of the addition of Z.

6.4.4 Alternative Architecture with Internal Z

LLMs as well as scientific computing require dot products of much larger sizes than 16. When the DPA operator is used to emulate a larger dot product operator with an arbitrary size D, an alternative architecture, depicted in Fig. 6.13, can be used where the addend is not exposed to the user. It is replaced with an accumulator that can only be reset to 0. This accumulator is here considered as a product; for instance, its exponent can be chosen as the maximum exponent. In this architecture, the final normalisation shift and round components are only used when the D products have been added, to round the result into an FP32 number.

Compared to an FP32 addend, this approach increases accuracy for large dot products: the whole accumulator of $81+\log_2(N)$ bits loops back into the computation instead of the 24 bits of precision of an FP32 addend.

An FP32 number can still be added by first splitting it as two E9S12 and inputting as a product with 1. This is trivial in a single dot product, and takes multiple operations in a matrix multiply add operator to initialise every addend independently. This also does not always provide the correctly rounded sum of an FP32 and an FP16 dot product (consider the case where this FP32 has a very large or very small exponent compared to the accumulator exponent, leading to bits being lost in the shift). For the same reason, this alternative architecture cannot emulate the FP32 FMA.

In short, this variant is a trade-off, as it improves the overall accuracy over many iterations but loses precision over one unique computation. The lack of correct rounding also reduces predictability in distributed systems and software simulation.

This accumulator is in a floating-point format, with an exponent and a fixed-point significand, normalised to avoid precision loss if the accumulator has the largest exponent. This also helps with regularity of the operator's behaviour.

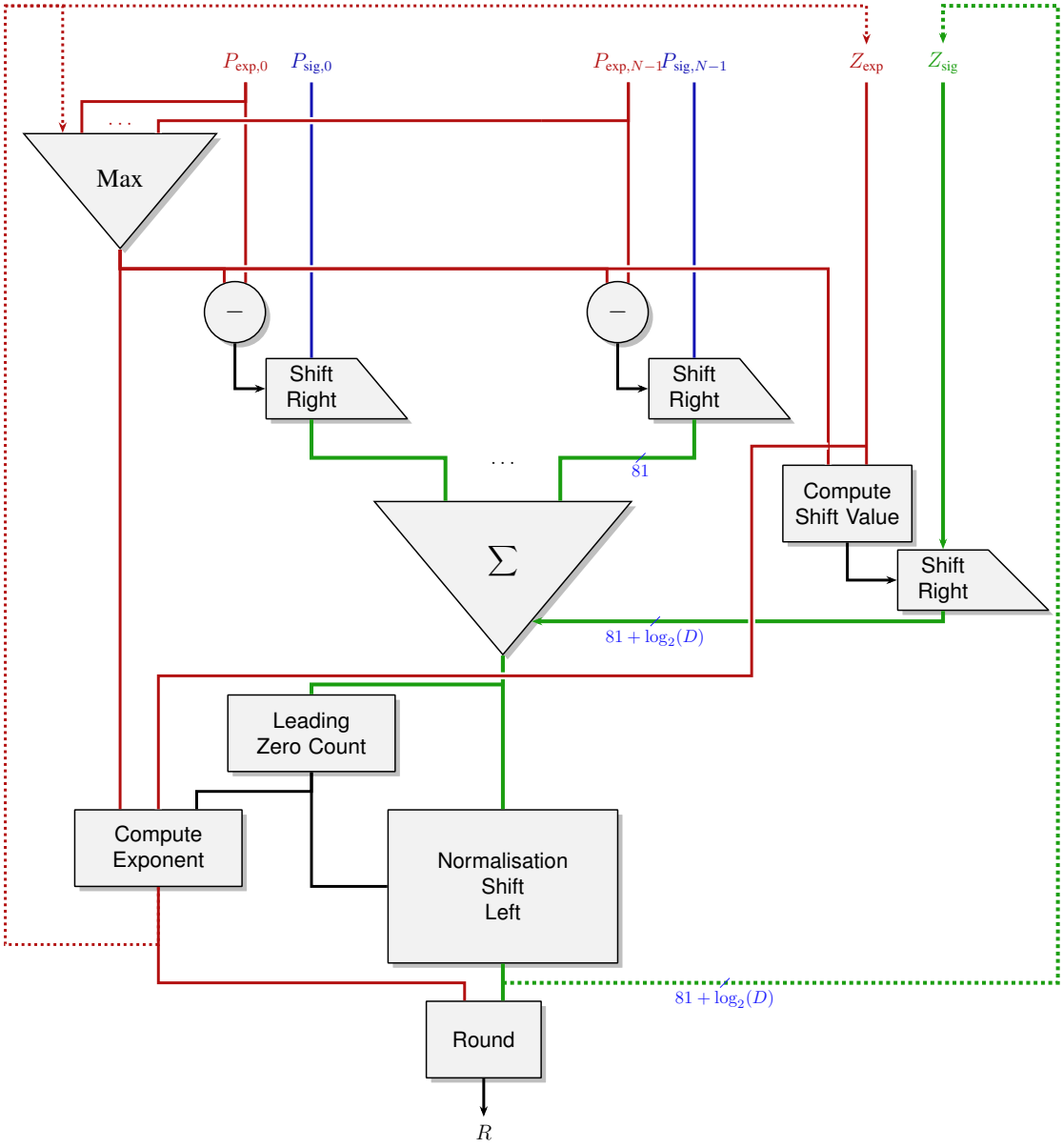


Figure 6.13: Dot product operator for E9S12, with loop on the accumulator.

To avoid unnecessary logic dealing with the sign, the number is normalised in two's complement representation, which can be a problem for negative powers of two. For example, 2 is encoded as 0010_2 in binary on 4 bits, and -2 as 1110_2. In the positive case, the normaliser returns $C = 2, R = 1000_2$, and in the negative case $C = 3, R = 0000_2$. This is actually not an issue when concatenating with the sign bit after normalising. In the positive case, $R \times 2^{-C} = 01000_2 \times 2^{-2} = 2^4 \times 2^{-2} = 2$, and in the negative case, $R \times 2^{-C} = 10000_2 \times 2^{-3} = -2^5 \times 2^{-3} = -2$.

6.5 Experimental Results

6.5.1 Operator Validation

This operator was implemented within the FloPoCo framework [35], which includes MPFR-powered test bench generation. When the operator takes FP16 multiplicands, a correctly rounded result is expected. With BF16 and FP32 multiplicands, the test case is accepted when the result is a faithful rounding of the exact result computed in MPFR, and rejected otherwise. When rejected, the case is checked by hand to verify that it is not a bug but a normal behaviour of the operator.

For every input mode, the FMA is expected to be correctly rounded: if the number of non-zero product is 1 or less, the test bench expects a correctly rounded result.

The FloPoCo framework encourages the definition of standard test cases. Here, these include the tests of negative zeros and subnormals, as well as debug tests for development purposes. It also allows for directed random tests, where the random number generator is biased towards increasing the probability of some rare but important situations. Here, directed random tests enable the verification of the FMA mode, forcing all products but one to be zero. The probability of cancellation cases (where products can have very close exponents and different signs) is also increased. Finally, directed random tests are also used to increase the frequency of subnormal inputs and outputs, as this is often an error-prone part of the operators.

6.5.2 Synthesis Results

This section compares DPA operators of size 16 for FP16 and BF16, and thus of size 4 for FP32 multiplicands. The actual pipelining of the chosen operator will be highly dependent of its integration in the larger context of a MMA unit. For this reason, we prefer to compare combinatorial operators, which are synthesised for one clock cycle at 333MHz to allow for later pipelining in 3 clock cycles at 1GHz. The operators have been synthesised with the Synopsys Design Compiler NXT for the TSMC 4FFC node.

Here, $\mathrm{DP16}_{\text{FP16-BF16}}4_{\text{FP32}}\mathrm{A}$ is the combined FP16, BF16 and FP32 operator based on the E9S12 format. It is compared to an alternative with two separate operators: $\mathrm{DP16}_{\text{FP16-BF16}}\mathrm{A}$, a combined FP16 and BF16 dot product operator dedicated to 16-bit operands, and FDMDA4, a correctly rounded FP32 dot product operator with subnormal support which will be described in Chapter 7. The results of the synthesis are shown in Table 6.1.

Operator	Area	Power	
	(μm^2)	Leakage (nW)	Total (mW)
$DP16_{FP16}A$ [8]	1796	477	1.83
$DP16_{FP16\text{-}BF16}A$	2343	602	2.11
FDMDA4 (Chap. 7)	1865	476	1.87
FDMDA4 (Chap. 7) & $DP16_{FP16\text{-}BF16}A$	4208	1078	2.11
$DP16_{FP16\text{-}BF16}4_{FP32}A$	2504	657	2.62
Alternative $DP16_{FP16\text{-}BF16}4_{FP32}A$	1949	531	2.23

Table 6.1: Synthesis results for the various operators configurations.

The combined operator has a significantly smaller area compared to having two operators $DP16_{FP16\text{-}BF16}A$ and FDMDA4 (-40%), and this is correlated with the leakage power (-39%). However, total power consumption increases (+24%) as the operator is less specialised.

The alternate architecture is cheaper that the other architectures mainly because it does not require the accumulator size ($81 + \log_2(D) = 97$) to be as big as the accumulator size in the other operators ($130 + \log_2(17) = 136$).

6.6 Integration into Kalray products

The operator designed for Kalray is based on the alternate architecture presented in Section 6.4.4 It is pipelined with a latency of 5 cycles (Fig. 6.14).

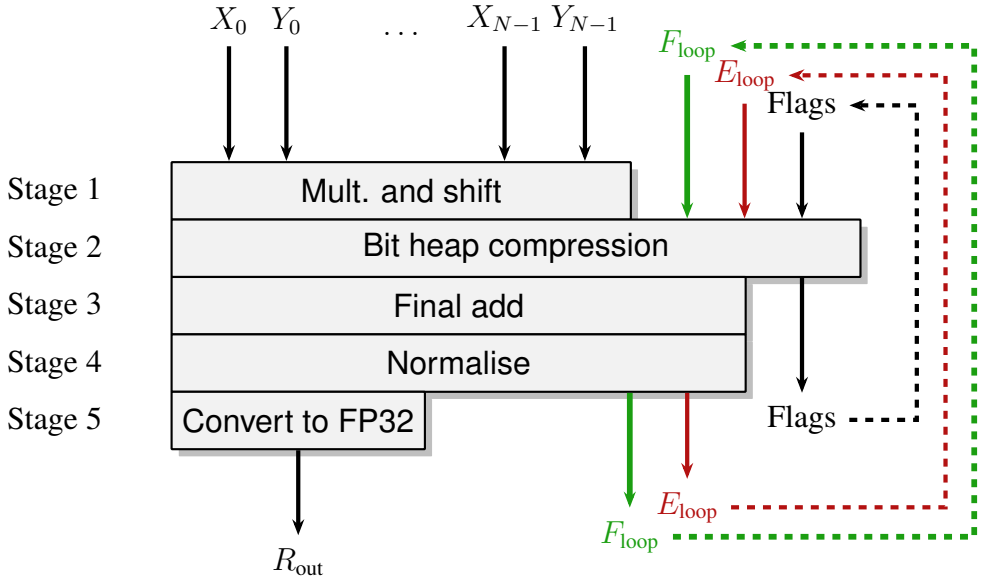


Figure 6.14: Pipeline stages of the dot product operator designed for Kalray.

The registers storing F_{loop}, E_{loop} and Flags for this operator are shared with the ones storing R_{fix} and Flags for the FP8 operator (see Sec. 5.4).

While the latency between the input and the output of the looping accumulator is on 3 cycles, the overall scheme in the MMA context (see Sec. 3.2) reuses the accumulator after 8 cycles.

6.7 Conclusions: The Cost of Accuracy

This work is motivated by the extension of a Matrix Multiply Add (MMA) unit, originally designed for deep learning applications, for use in FP32 numerical applications. The building block for this MMA unit is a Dot Product Add (DPA) operator with FP16, BF16 or FP32 multiplicands, an FP32 addend and an FP32 result.

The proposed DPA operator performs a dot product between vectors of $4N$ 16-bit floating-point elements or N FP32 elements. This operator accepts FP32 multiplicands by decomposing each of them into a pair of numbers represented in a suitably designed internal format, consisting of 12 bits of significand and 9 bits of exponent.

In these architectures, the size of the accumulator greatly correlate with the area of the operator, as well as with the accuracy of the performed computation.

In the previous Chapter (Chap. 5), the size of the accumulator was chosen to accommodate the whole fixed-point representation of the various FP8 formats. In this chapter, the size is such that the result is correctly rounded for FP16 operands.

Adding BF16 support fundamentally changed the structure of the operator from a full-precision fixed-point accumulator scheme, to a truncated floating-point sum. This added logic for the computation of the maximum exponent, as well as a shifter on the input of the accumulator, and the fact that the normaliser is in the loop and not outside.

For the same accuracy and accumulator size, the second architecture is more expensive and has a larger latency that the first one.

Operator	Latency (ns)	Area (μm^2)	Dynamic power (mW)	Leakage power (nW)
Combined FP8 ($N = 32$) full-precision fixed-point	4	1413	4.7	431
FP16 ($N = 16$) full-precision fixed-point	4	1562	5.7	530
FP16-BF16-FP32 ($N = 16$) truncated floating-point	5	2451	8.8	1287

Table 6.2: Synthesis of pipelined operators for 4 nm technology at 1.56 GHz.

When computing with a full-precision fixed-point accumulator, one must take into account extra bits used to absorb the carries during a chain of multiple computations. For the Kalray operator, the value 16 was chosen, to be able to accumulate ≈ 64000 products exactly without worrying about accumulator overflow.

It is not necessary to add as many bits when using a truncated floating-point sum architecture, as the accumulator has an exponent. The size of the accumulator can thus be much smaller.

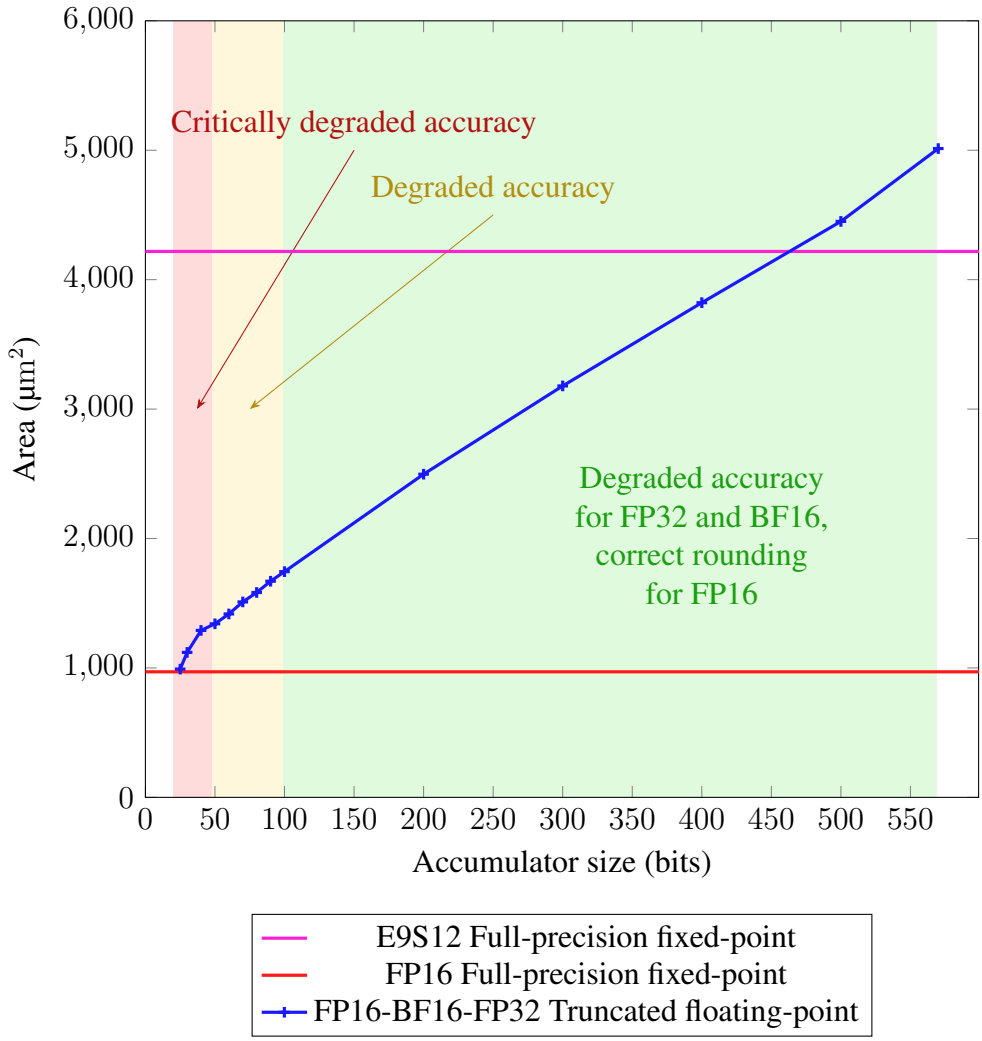


Figure 6.15: Synthesis results of dot product operators depending on the accumulator size, pseudo-pipelined for a latency of 5 ns or 200 MHz on 4 nm technology node.

Figure 6.15 shows the area of the truncated floating-point operators depending on the accumulator size. The operators are *pseudo-pipelined*: the synthesis is set at a latency of $5\,\mathrm{ns}$, which is a frequency of $200\,\mathrm{MHz}$, chosen as all the designs comfortably fit into the latency. Full-precision fixed-point designs are generally faster as they do not require to computed the maximum exponent.

Not including pipelining does not result in an exactly fair comparison, as registers are very expensive, however it can give an idea of the cost when choosing to implement or not a correctly rounded design.

Figure 6.15 also shows the area of a full-precision fixed-point E9S12 operator, which has the same size multipliers than the FP16-BF16-FP32 operator. It also shows the area of a full-precision fixed-point FP16 operator, which is a less fair comparison (as the multipliers are 1 bit smaller) but still interesting to see the cost of the floating-point alignment logic when the accumulator are about the same size.

A popular choice for the accumulator size is 24 as that is the precision of the FP32 output. Synthesis show that this architecture with critically degraded precision is about as expensive as FP16 full-precision fixed-point architecture. This consolidates that it was the right choice to use a fixed-point architecture for FP8, and it would also be the case for an FP16 only operator.

The truncated floating-point method also largely degrades the accuracy, which is not acceptable for small input floating-point formats like FP8 and FP16 where every significant bit counts.

Chapter 7

Correctly Rounded Dot Product on Larger Precision formats

The point made in Chapter 5 still stands: a full-precision fixed-point accumulation in practice consists in mostly adding zeros. The larger the exponent range, the lower the ratio of non-zero bits added over zero bits.

In Chapter 6, the size of the accumulator was reduced by truncating all the bits that were too small compared to the exponent of the largest magnitude. This method does not guarantee correct rounding, in fact it does not even protects against cancellations.

The architecture presented in this chapter starts from the same situation, but instead tries to compress the zeros without sacrificing significant bits.

7.1 Introduction

This work addresses the correctly rounded dot-product-and-add:

$$R = \circ(X_0 \times Y_0 + \ldots + X_{N-1} \times Y_{N-1} + Z)$$

Here $(X_i)_{i\in[0,N-1]}$, $(Y_i)_{i\in[0,N-1]}$, Z and R are floating-point numbers in a representation with a wide dynamic range such as FP64, FP32 and BF16. The multiplicands $X_{i\in\{0;N-1\}}$ and $Y_{i\in\{0;N-1\}}$ have $w_{e,\text{in}}$ exponent bits and $w_{f,\text{in}}$ significand bits. The output R and the addend Z have a possibly wider format with $w_{e,\text{out}}$ and $w_{f,\text{out}}$ bits. The corresponding operator is called FDPNA, possibly suffixed with the format, for example FDPNA$_{\text{FP32}}$ for homogeneous operators and FDPNA$_{\text{BF16}\to\text{FP32}}$ for mixed-precision ones.

7.1.1 Motivations

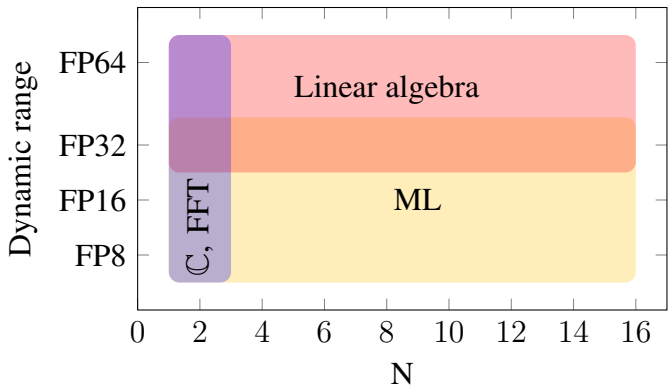


Figure 7.1: Applications of various instances of the FDPNA operators (BF16 has the same dynamic range as FP32).

The main applications of FDPNA operators are accumulations of partial dot products in machine learning and linear algebra applications (Fig. 7.1). In addition, the operator where $N = 2$, also called FDMDA (Fused Dual Multiply Dual Add) or ExSdotp [4] effectively supports complex floating-point arithmetic, as a complex

fused multiply and add $R = XY + Z$ is implemented with the best possible accuracy in only two operations: Noting $j^2 = -1$,

$$R = R_{\text{re}} + jR_{\text{im}} \text{ with } \begin{cases} R_{\text{re}} = \circ(X_{\text{re}}Y_{\text{re}} - X_{\text{im}}Y_{\text{im}} + Z_{\text{re}}) \\ R_{\text{im}} = \circ(X_{\text{re}}Y_{\text{im}} + X_{\text{im}}Y_{\text{re}} + Z_{\text{im}}) \end{cases}.$$

7.1.2 Related Work and Previous Implementations

Various dot-product-and-add implementations are described in Chapter 6.

However, many of those implementations are not correctly rounded, except [116, 8, 4].

The ExSdotp fused dot-product-and-add operator of [4] first sorts the three terms based on their exponents, the exponent of each product being the sums of the multiplicand exponents. The two larger terms are added, and if a full cancellation is detected, the smaller term is restored for the result. This works for the sum of three floating-point numbers. However, with products involving subnormal multiplicands, this approach may result in incorrect results with directed rounding. For example, consider ExSdotp FP16→FP32 with rounding up; $X_0 = Y_0 = X_1 = 1$; Y_1 a small positive FP16 subnormal; Z a FP32 such that $Z = -2X_1Y_1$. Obviously $X_0Y_0 + X_1Y_1 + Z = 1 - X_1Y_1 < 1$, therefore, R should be 1. However, the exponent of X_1Y_1 is larger than that of Z due to Y_1 being subnormal. Therefore the sort, based on the exponents only, will use X_1Y_1 for the addition instead of Z, and the result returned will be $R = 1 + 2^{-23}$. This issue also also affects the inexact flag in round to nearest.

Operators described in Chapter 5 use full-size fixed-point accumulation guarantee the correct rounding of the result. The alternative studied here is to compress identical bits inside the full-size accumulator. It builds on Tao et al. [116], with multiple improvements: adding subnormal support, managing the addend Z, and mixed-precision. Alternative techniques are explored for several sub-problems, including sorting networks and a parallel-prefix computation of the significand shifts.

7.2 Operating Principles

7.2.1 Architecture Overview

The operators in this chapter use the same high-level architecture as the previous chapter (Fig. 7.2).

The component denoted FPΣ aligns the significands and sums the $N+1$ floating-point terms. Following the summation, a Leading Zero Count (LZC) retrieves the exponent of the result. Finally, the sum is normalised or subnormalised, rounded, and output as a floating-point number. Overflow, underflow, NaN and IEEE 754 flags are managed within FPΣ as well.

The FPΣ component can be implemented like described in Chapter 5, using a full-precision fixed-point accumulation. The size of the fixed-point format w_p is summarised in Table 5.1 (p. 80). Extra bits extend the fixed-point format on the left to guard against possible overflows: $w_{\text{full}} = w_p + \lceil \log_2(N+1) \rceil$.

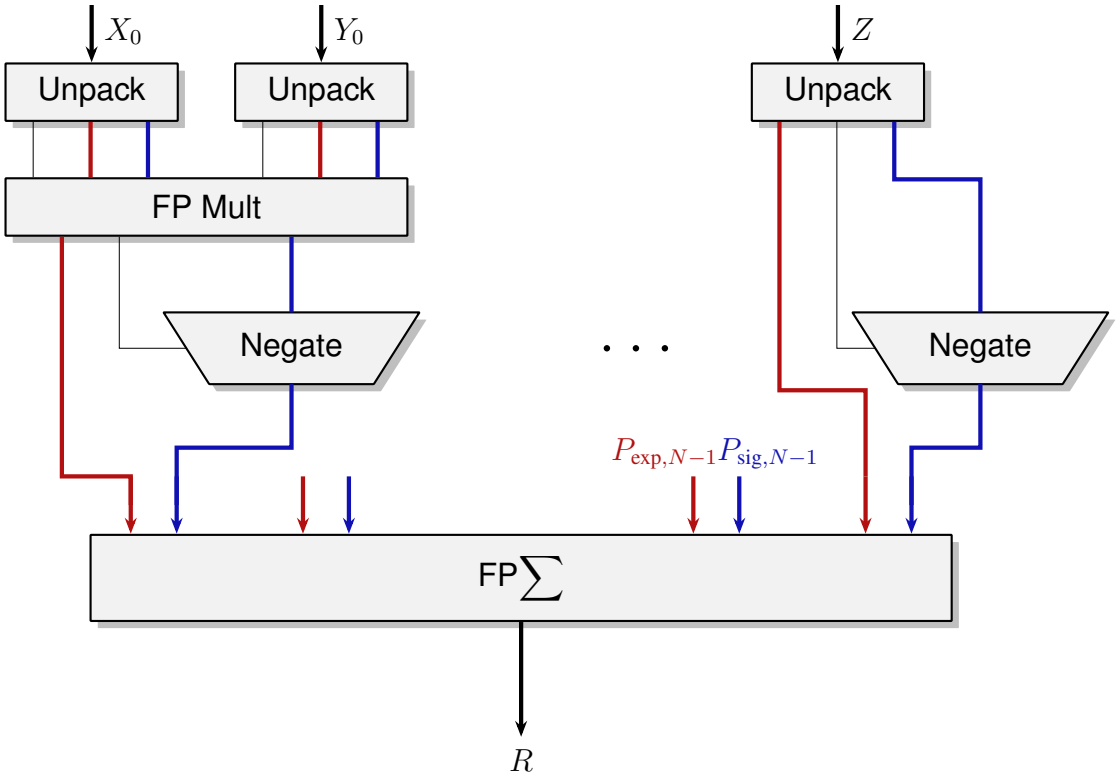


Figure 7.2: High-level architecture of the FDPNA operator.

This approach is quite efficient for formats with small full-size accumulators. It has been used in particular for the exact sequential accumulation of FP16 products [7] and a FDP8A$_{\text{FP16}\to\text{FP32}}$ operator [8]. In the mixed-precision case, it is better to perform the sum of products separately in an accumulator corresponding to the small format, then perform the last addition of Z as in a classical FMA. As the sum of products is exact, it is associative and can be parallelised without any consideration of the exponent values.

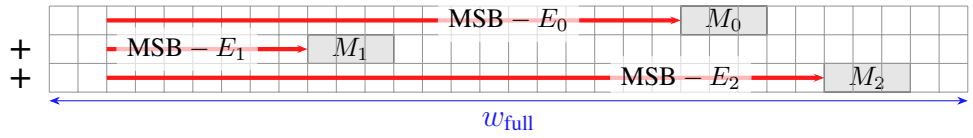


Figure 7.3: Example of alignment for the sum using a full-precision fixed-point accumulation.

7.2.2 Compressed FPΣ Principles

When (as is the case in Fig. 7.3) the accumulator size is much larger than the sum of the widths of the $N+1$ significand terms to add, one observes that most of the summation adds zeros (or sign bits, which are similarly easy to manage). Intuitively, these parts of the summation can be saved, and the worst-case size of actual addition needed should be roughly N times the size of a term significand. In this chapter,

compression refers top the suppression of the columns of predictably identical bits in Fig. 7.3. It is called realignment in [116].

The core idea is to use a single fixed-point multi-operand adder of size $w_{\text{compressed}} < w_{\text{full}}$. The shift values needed to align the $P_{\text{sig},i}$ before their summation are no longer trivially deduced from the $P_{\text{exp},i}$ as in the full-size case: they now need a more complex computation to skip the compressed bits. The main issue is the combinatorial explosion of the alignment situations, because it has to be implemented in hardware.

To address this explosion, the terms a first sorted by their exponent $P_{\text{exp},i}$. Section 7.4.1 discusses our approach for this. Let $(E_0^*, M_0^*), (E_1^*, M_1^*), ...(E_N^*, M_N^*)$ be the renumbered (exponent, significand) pairs such that $E_0^* \geq E_1^* \geq \ldots \geq E_N^*$. The sorting approach requires that the products X_iY_i and the addend Z be represented in the same way. The significand size w that fits all is the maximum width of Z_{sig} and all $P_{\text{sig},i}$:

$$w = \max(2 + m_{\text{out}}, 2(1 + m_{\text{in}}))$$

As illustrated in Fig. 7.4, the significand Z_{sig} of the addend Z is extended with an extra MSB corresponding to the overflow bit of a product. In the homogeneous case, Z_{sig} is extended with extra LSBs. In the mixed-precision case Z_{sig} usually also has more precision to the right than the other $P_{\text{sig},i}$, for example 22 bits for BF16 products versus 24 bits for FP32. In the sequel, all $P_{\text{sig},i}$ are considered to have the same size w. The exponent Z_{exp} of Z must be similarly updated.

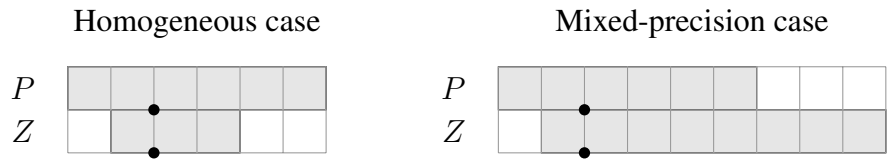


Figure 7.4: Common format for the significands of products and addend.

Accordingly, the architecture proposed is sketched in Fig. 7.5. The (exponent, significand) pairs are first sorted by their exponent, then processed by a component that determines the adjusted shift values S_i. The sorted significands are then shifted before being summed into an adder tree. The sum RM_{full} is converted back to a (sign,magnitude) representation, normalised, then rounded to produce the final significand.

Compared to the full-size FPΣ, this architecture involves one fewer shift since $N - 1$ terms are shifter with respect to the term with the largest magnitude which is not shifted. If $w_{\text{compressed}} < w_{\text{full}}$ the shifts are smaller, as are the adder tree and the LZC. However it also requires a sorting component, more exponent pre-processing, and some exponent post-processing as well. This trade-off is evaluated quantitatively in Section 7.5.

7.2.3 Introductory Considerations

What is the smallest number of bits $w_{\text{compressed}}$ such that all the $N + 1$ terms can be added using integer adders of size at most $w_{\text{compressed}}$ and the exact sum can be recovered for rounding as if it were computed on w_{full} bits? To address this question

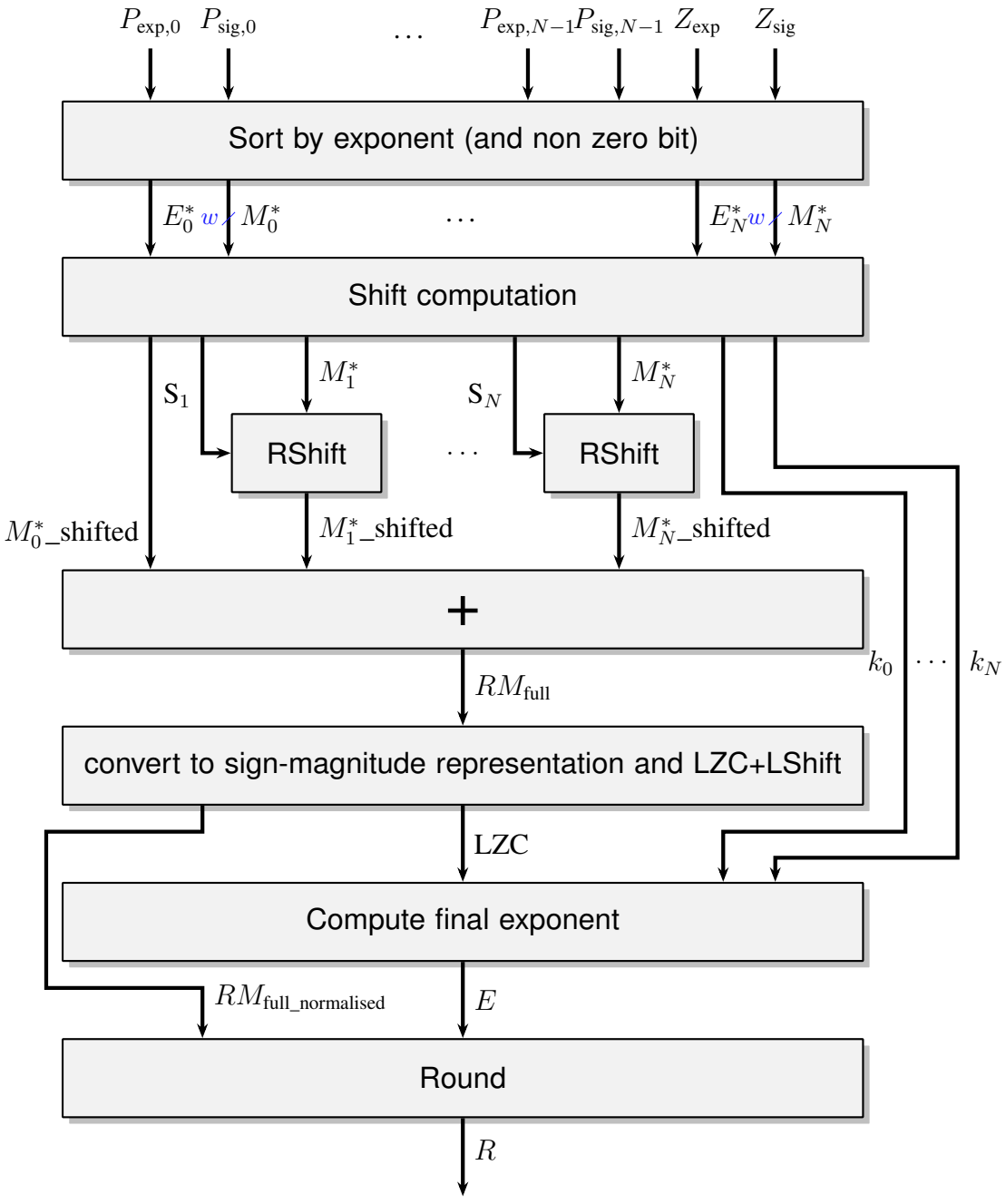


Figure 7.5: Architecture of the compressed FPΣ component.

while introducing notions needed in the sequel, first consider the trivial case $N = 1$ then the simple case $N = 2$.

Two Terms $(N = 1)$

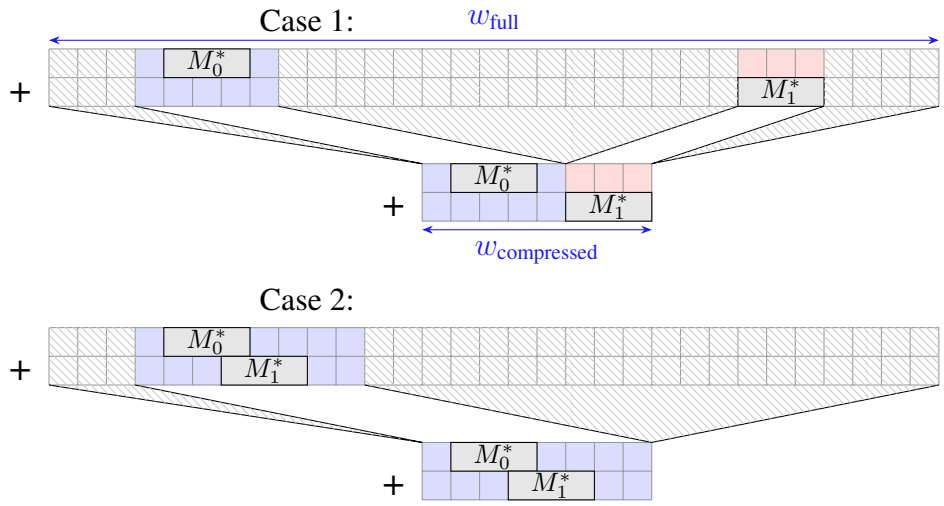


Figure 7.6: Compressing the exact addition of two terms.

Fig. 7.6 shows a fixed-point addition of two terms. There are two coloured zones in this figure (from left to right: blue and red). There are two cases: (case 1) the two significands are far apart, and the bits between them can be omitted, as the result will be M_0^* or one of its immediate FP neighbours (one round bit must be kept to the right of M_0^*); or (case 2) M_1^* overlaps M_0^*, and there will be an addition of size at most $2w$ bits. In both cases the w_{full} bits of the exact sum can be compressed in $2w + 2$ bits. This observation is exploited in classic FP adders (see Sec. 6.1.1), but with an additional trick: the bits in the red zone of Fig. 7.6 can be compressed into a sticky bit and a guard bit [96] in a way that keeps enough information for a correctly rounded addition.

Three Terms $(N = 2)$

There is now the possibility of a complete cancellation of the two leading terms M_0^* and M_1^*. In such a case, the exact result is M_2^*, therefore all the bits must be kept, so it is incorrect to compress them into a sticky bit. This will be the case for all $N > 1$, sticky bits will no longer be mentioned. Fig. 7.7 shows the four alignment cases of fixed-point addition for three terms $(N = 2)$.

In case 1, the three significands are fully separated and all the bits between them can be compressed. There are three coloured zones in this figure (from left to right: blue, red and green), and the architecture will need to remember the exponent of each significand and associate it with the corresponding zone to emulate the full-size accumulator. In all the sequel, zone i is associated with the exponent E_i^* and a priori contains the significand M_i^*.

In case 2, the two lower significands M_1^* and M_2^* overlap, which means that their sum may overflow by one bit to the left of M_1^*. The red zone, in the compressed view,

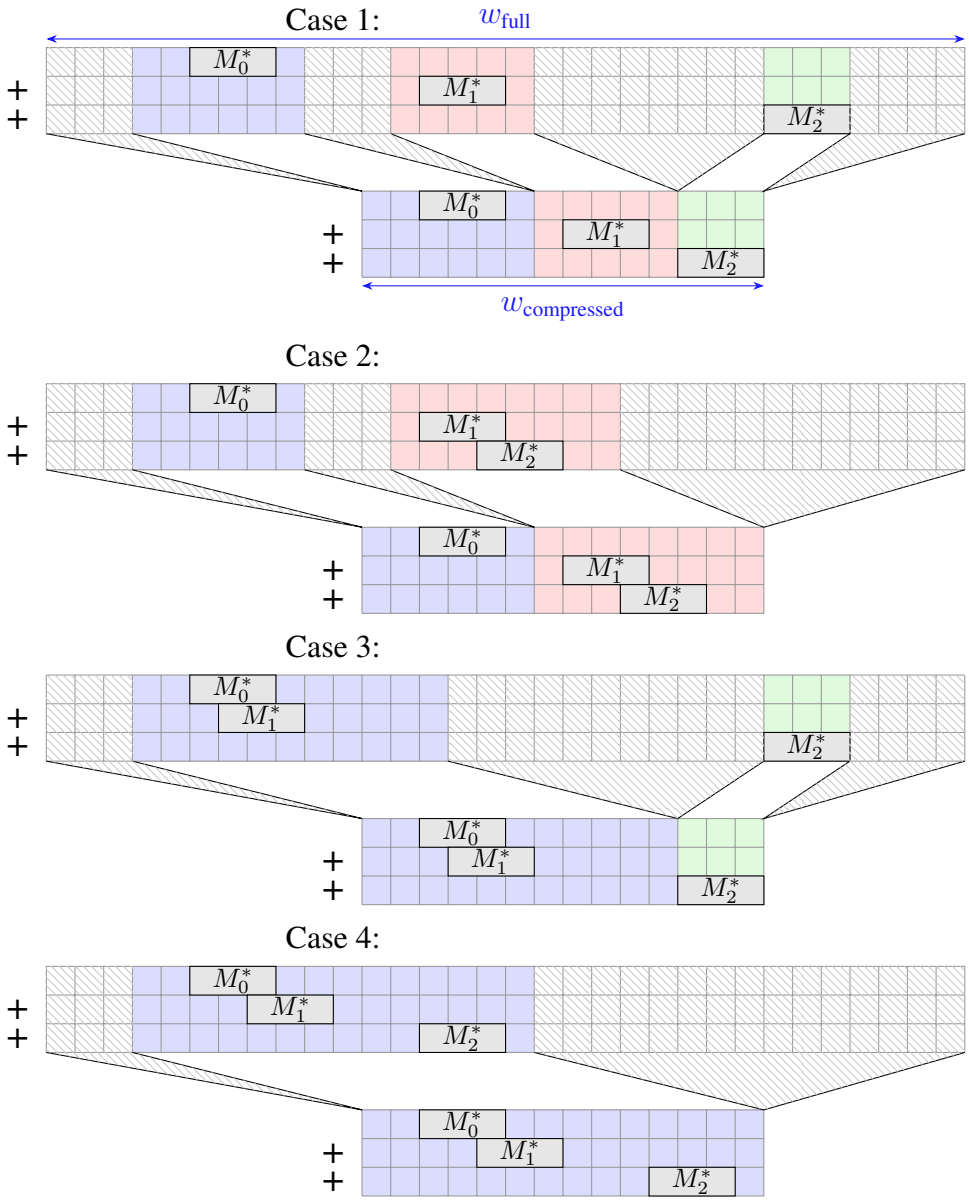


Figure 7.7: Compressing the exact addition of three terms

reserves for this overflow one more bit than in Fig. 7.6 to the left of M_1^*. Besides, there is no longer a red zone as in case 1: significand M_2^* belongs to the red zone: since it must be added to M_1^*, it inherits its exponent E_1^*. The good news is that in the compressed adder tree, the bits necessary to this addition may recycle those of the red zone in case 1.

Case 3 is similar, but with M_1^* now in the blue zone due to M_1^* overlapping M_0^*. Note that the red zone, associated with the exponent E_2, exists in this case. In such a case E_1^* will be used to shift M_1^* to its proper place in the compressed sum, but it is not associated with a zone.

Finally, case 4 shows the situation where only one zone is associated with E_0^*, which happens as soon as the compressed accumulator can hold the exact sum. Here neither E_1^* nor E_2^* are associated with a zone.

7.3 Construction of the Compressed FPΣ

This section first generalises the $N = 1$ and $N = 2$ considerations to formally define the sizes and parameters of the $N + 1$ zones of a FDPNA operator. Then an architecture to shift all the significands to their proper place in the compressed accumulator is defined. Determining those shift values can be implemented as a parallel prefix computation.

7.3.1 Compressed FPΣ Parameters for N Terms

In case of N terms, the key is to determine the zones in the summation where each significand M_i^* should be positioned if it does not overlap with other significands. These zones are specified by their boundaries, denoted d_i. Fig. 7.8 illustrates the d_i values for $w = 3$, in cases $N = 2$ and $N = 4$.

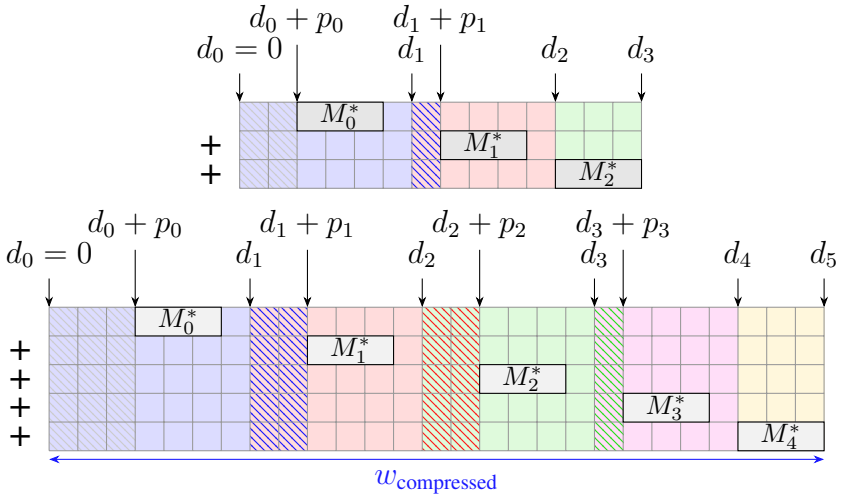


Figure 7.8: Position of the zone delimiters in cases $N = 2$ and $N = 4$.

The number of bits between d_i and d_{i+1} should at least be the width w of the significand. An extra bit is added to the right of the significand as a placeholder

for a round bit. Besides, p_i protection bits are added to the left of the significand (see Fig. 7.8) to absorb the worst-case overflow of the significands of lower or same magnitude:

$$\begin{aligned} p_i &= \lceil \log_2(N-i) \rceil \\ d_0 &= 0 \\ d_i &= d_{i-1} + w + 1 + p_{i-1} \quad . \end{aligned}$$

This recurrence also defines the total size of the compressed accumulator $w_{\text{compressed}} = d_{N+1} - d_0$. The p_i and d_i parameters only depend on the value of N that is set at design time.

7.3.2 Definition of the Shift Values S_i for Three Terms

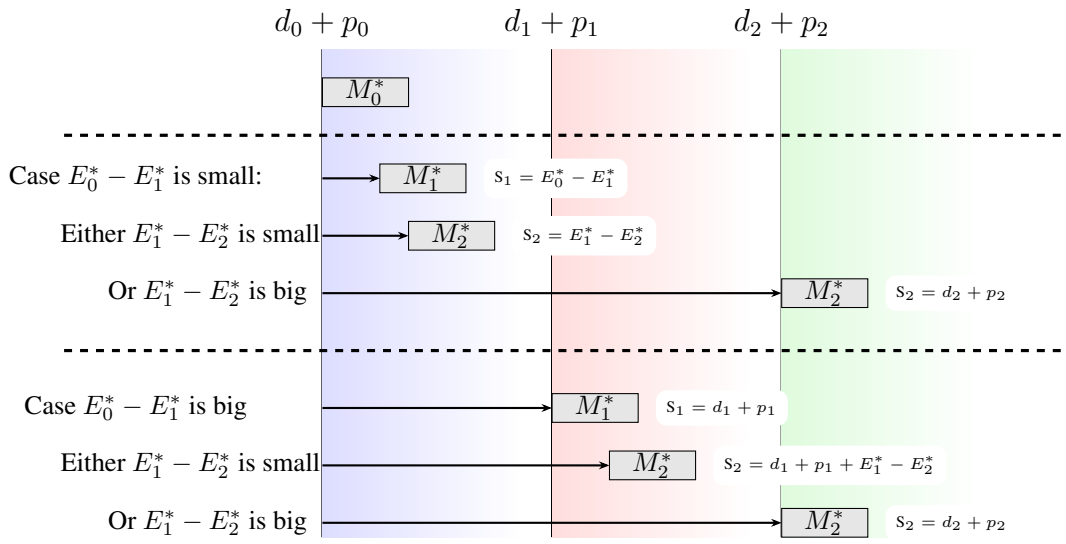


Figure 7.9: Determining the shift values.

M_0^* is placed in a constant position to the left of the addition: $S_0 = d_0 + p_0 = p_0$. Determining the subsequent S_i involves another case analysis, illustrated in Fig. 7.9 for $N = 2$:

- If E_1^* is close to E_0^* then $S_1 = d_0 + p_0 + E_0^* - E_1^*$, and
  - If E_2^* is close to E_0^* then M_2^* is placed in the zone of M_0^* and $S_2 = d_0 + p_0 + E_0^* - E_2^*$
  - Else M_2^* is placed in its own zone: $S_2 = d_2 + p_2$
- Else M_1^* is placed in its own zone: $S_1 = d_1 + p_1$, and
  - If E_2^* is close to E_1^* then $S_2 = d_1 + p_1 + E_1^* - E_2^*$

– Else M_2 is placed in its own zone: $S_2 = d_2 + p_2$.

Specifically, E_i^* *close to* E_j^* means that M_i^* can be placed in the same zone as M_j^*. Then it aligns with E_j^*, and the bits of zone i are recycled to make space for the addition in zone j. This leads to the following definitions for the shifts:

$$\begin{aligned}
S_0 =& d_0 + p_0 \\
S_1 =& \min(d_0 + p_0 + E_0^* - E_1^*,\ d_1 + p_1) \\
&\text{if } S_1 = d_1 + p_1 \\
&\quad \text{then } S_2 = \min(d_1 + p_1 + E_1^* - E_2^*, d_2 + p_2) \\
&\quad \text{else } S_2 = \min(d_0 + p_0 + E_0^* - E_2^*, d_2 + p_2)
\end{aligned}$$

7.3.3 Parallel Prefix Computation of S_i for 3 terms

The recurrence defining S_i may be evaluated faster by reformulating it into a form suitable for parallel prefix computation. First, each level contains dependencies on all the positions of the previous levels, but this is not necessary for a correct result. Indeed, if E_2^* is close to E_1^* and E_1^* is close to E_0^* then:

$$\begin{aligned}
& d_0 + p_0 + E_0^* - E_1^* \leq d_1 + p_1 \\
\Rightarrow\quad & d_0 + p_0 + E_0^* - E_1^* + E_1^* - E_2^* \leq d_1 + p_1 + E_1^* - E_2^* \\
\Rightarrow\quad & d_0 + p_0 + E_0^* - E_2^* \leq d_1 + p_1 + E_1^* - E_2^*
\end{aligned}$$

The formula for S_2 can be rewritten as:
$S_2 = \min(d_0 + p_0 + E_0^* - E_2^*,\ d_1 + p_1 + E_1^* - E_2^*,\ d_2 + p_2)$.

Second, observe that the last argument of the min can be rewritten as $d_2 + p_2 = d_2 + p_2 + E_2^* - E_2^*$. This leads to:
$S_2 = \min(d_0 + p_0 + E_0^*,\ d_1 + p_1 + E_1^*,\ d_2 + p_2 + E_2^*) - E_2^*$. Note that all the $d_i + p_i + E_i^*$ can be computed in parallel as soon as the order of the exponents is known.

7.3.4 Parallel Prefix Computation of S_i for $N + 1$ terms

In the general case of $N + 1$, the first expression of the shift is very long, as each S_i depends on cases of the $i - 1$ previous terms:

$$S_i = \min(\min(d_j + p_j + E_j^* - E_i^*,\ j \in \{0, \ldots, i-1\}),\ d_i + p_i) \quad .$$

The parallel prefix formulas in the general cases are computed in the same way as the case for 3 terms, resulting in the following formulas:

$$\begin{aligned}
S_0 &= d_0 + p_0 \\
S_i &= \min_{j \in \{0,\ldots,i\}} \{d_j + p_j + E_j^*\} - E_i^* \\
k_i &= i \text{ if } (S_i = d_i + p_i) \text{ else } k_{i-1}
\end{aligned}$$

The index k_i is the index of the zone to which M_i^* belongs and is computed along the min.

This is also a convenient form for hardware implementation using a Hillis & Steele parallel prefix tree [61] as shown in Fig. 7.10.

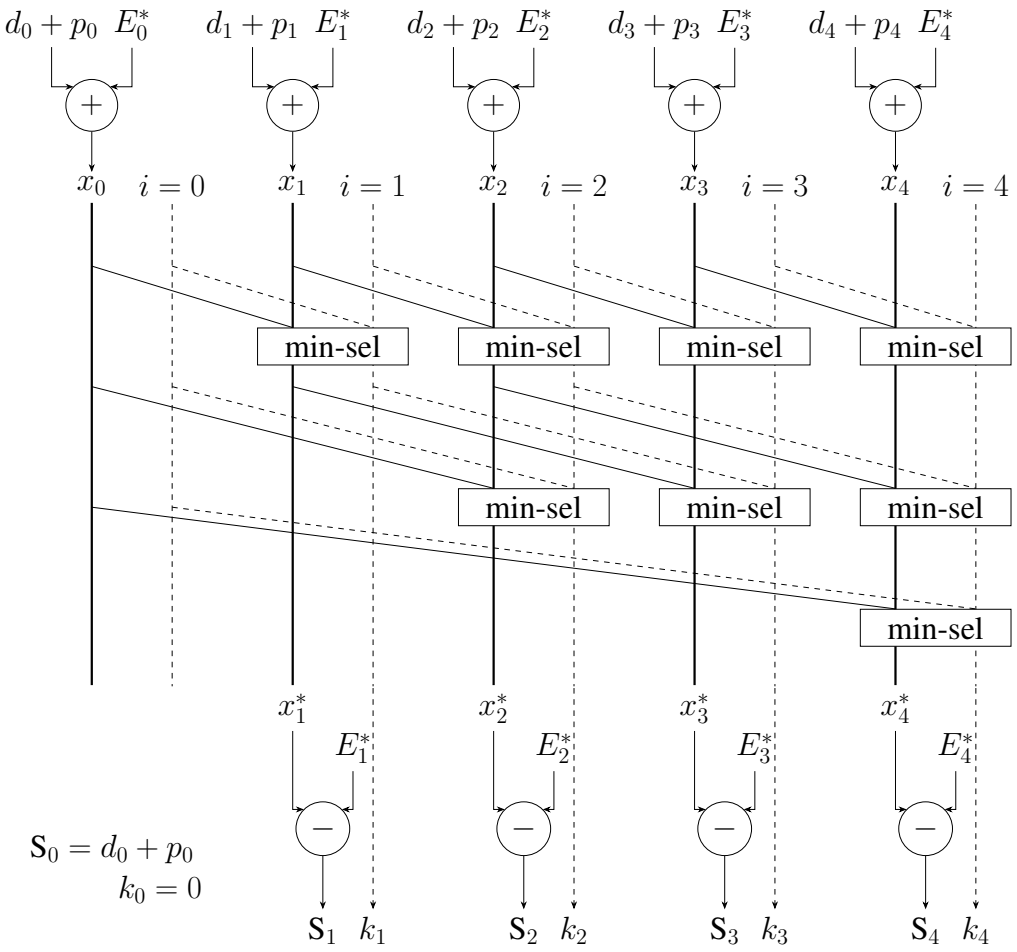


Figure 7.10: Example of parallel prefix computation for $N = 4$.

7.3.5 Computation of Final Exponent E

Once the summation is done, its result is converted back into a sign-magnitude representation and then normalised. Normalisation starts with a leading zero counter (LZC) which determines the number L of zeros before the leading bit in the (compressed) sum. This L is compared to all the d_j to determine the index i of the zone of the result. Then the index k_i is used to retrieve the exponent $E^*_{k_i}$ actually associated with this zone. The result exponent verifies $L - (d_{k_i} + p_{k_i}) = E^*_{k_i} + 1 - E$. Here the +1 captures the fact that our M^*_i have two bits, not one, to the left of the binary point (see Fig. 7.4)

Finally the normalised result exponent E is computed as:

$$\begin{aligned} \text{If } L = d_n &: \text{result is } R = 0 \\ \text{else if } L < d_1 &: E = E^*_0 + 1 - (L - (d_0 + p_0)) \\ \text{else if } L \in [d_i, d_{i+1}[&: E = E^*_{k_i} + 1 - (L - (d_{k_i} + p_{k_i})) \end{aligned}$$

7.3.6 Subnormal Management

Fig. 7.11 illustrates a problematic situation to avoid: due to a cancellation, the normalised result contains bits that belong to two different zones. In the full sum there would be more bits between them, so the resulting significand is incorrect.

This situation cannot happen because of partial cancellations with *normal* inputs, as in this case the zone in which the cancellation occurs has been enlarged by fusing in the bits from zones to the right. See Case 3 of Fig. 7.7: there are more than w extra bits in the blue zone due to fusing in the red zone. A cancellation is either full (then the LZC will skip the blue zone), or it cancels at most $w - 1$ bits and there remains more than a significand worth in the blue zone to the right of the leading one. This is also true if three or more significands share the same zone, as the zone is correspondingly larger.

However, the problematic situation may occur if there is one subnormal input to a product. Managing this properly requires specific considerations on $w_{e,\text{in}}$, $w_{e,\text{out}}$, $w_{f,\text{in}}$, and $w_{f,\text{out}}$.

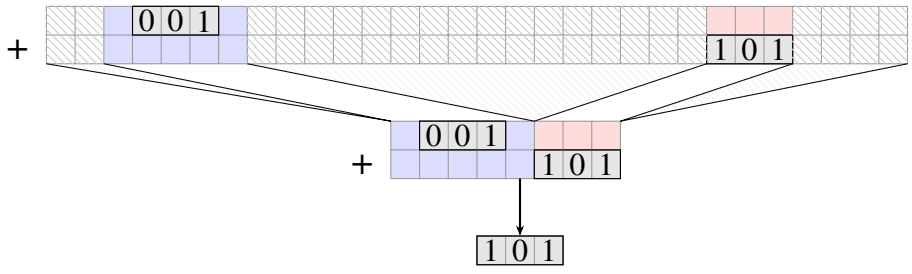


Figure 7.11: Result is using bits from two different zones, giving an incorrect result.

In the following paragraphs, M^*_0 will be used to discuss the problematic situations, since they arise when the leading 1 after sum is positioned near the next zone. In general, those situations may happen in any zone, in case of a complete cancellation in the leading zones.

Homogeneous Case ($w_{f,\text{in}} = w_{f,\text{out}}$ and $w_{e,\text{in}} = w_{e,\text{out}}$)

In this case remember that the width of the M_i^* is $w = 2 + 2w_{f,\text{in}}$. If M_0^*, the largest-magnitude term, is a product between a large normal and a subnormal, it cannot have more than $w_{f,\text{in}}$ leading zeros and the width of the zone is greater than $w = 2 + 2w_{f,\text{in}}$. So the problematic situation in Fig. 7.11 cannot happen.

If M_0^* is a product of two subnormals, then its exponent is the smallest possible, which means that all terms are products of two subnormals, all the following zones will be merged, and there are no problems either.

Mixed-Precision when $w_{e,\text{in}} \leq w_{e,\text{out}}$

In the worst case, the product of the two smallest non-zero subnormals has only one bit left, and nevertheless may be larger than Z. This leads to the problematic situation in Fig. 7.11.

FDP$NA_{\text{BF16}\to\text{FP32}}$ Special Case

The BF16 format does not support subnormals [69]. The operator FDP$NA_{\text{BF16.32}}$ can still support subnormals in its FP32 input Z and output R. A subnormal Z input is not a problem. Even if there is a product of BF16 numbers smaller than Z, the case in Fig. 7.11 cannot happen as the result will also be subnormal. Subnormalisation of the output is managed in all cases by the normalisation and rounding of the compressed exact sum.

Since BF16 and FP32 share the same exponent size, it is possible to support BF16 subnormals by taking $w = 32$ instead of $w = \max(2 + 23, 2(1 + 7)) = 25$. This modification ensures that the significand of the product of a normal by a subnormal cannot extend to the next zone: this product has a maximum of 8 leading zeros, therefore to ensure 24 significant bits in the zone, w must satisfy $w \geq 24 + 8 = 32$.

7.4 Implementation and Validation

7.4.1 Exponent Sorting Network

To ensure that subnormals sort bigger than 0, a bit is first appended to the LSB of each $P_{\text{exp},i}$, equal to 0 iff the significand is 0. These modified exponents are the keys in the (key, payload) pairs processed by the *Sort by exponent* component in Fig. 7.5. Several variants of this component were explored and implemented in FloPoCo [35].

First, there are two options for the payloads: either directly use the significand, or use only their index (which fits on much fewer bits). The second option makes the sorting itself cheaper, but requires an expensive multiplexing step in order to recover the significands from their indices. A detailed evaluation showed that sorting by indices ends up being almost twice as expensive in terms of area. In isolation, it would also be slower due to the extra multiplexing to recover the significands. However, sorting by indices reduces the overall latency since the sorting can be performed while the significand products are being computed. It is thus the option used in the sequel.

For the sorting algorithm itself, two alternatives were considered. The first is to use textbook sorting networks [3]. For $N = 4$ (5 terms to sort) a bitonic sort has a depth of 5 which is optimal [49]. For 9 terms, the network from [125] has a depth of 7 which is proved optimal in [100]. For 17 terms the sort used is from [44], whose depth of 10 is optimal.

The second alternative is from by Tao et al. [116]. To sort $n = N + 1$ terms, it first performs $n(n-1)/2$ parallel key comparisons. The resulting comparison bits or their complement are input to n counters (*population count*) that compute in parallel the rank of each key. Finally a $n \times n$ crossbar completes the sort. This technique obviously has a lower latency than a sorting network. In our experiments, even its area is competitive with sorting networks at least for the values of N considered here, all the more as only the final crossbar moves payloads. This sorting technique is therefore used in the sequel.

The sequential shift computation of [116] is able to recycle the values $E_i^* - E_j^*, \forall i \geq j$ computed in the sort for their sign bit. This saves area in a non-trivial way: it increases the number of multiplexers used at the end of the sort, but reduces the number of adders and saves their latency. The parallel prefix computation of shift values does not allow to use the same recycling trick. However, comparing is much cheaper than computing the difference [34].

7.4.2 Shifter and Bit Heap Sizes

Due to the structure of the summation in the compressed FPΣ component, the *RShift* and + components (Fig. 7.5) can be simplified. As the significand M_i^* can only be positioned in $\{Zone_0, \ldots, Zone_i\}$, the actual size of the shifters is $\text{max\_shift}_i = d_i + p_i$ (Fig. 7.12).

Similarly, this structure reduces the actual number of bits to be added. The summation is performed using a compressor tree that input the bit heap represented in Fig. 7.12. Note that a full-size architecture always requires a rectangular bit array.

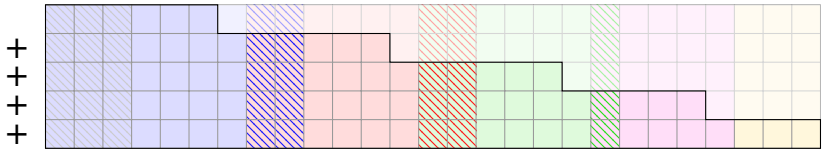


Figure 7.12: Bits that can be omitted from RShift and the summation bit heap.

This configuration enables the adder to start on the LSBs while the MSBs are still being compressed.

7.4.3 Operator Validation

The general FDPNA operators have been validated with the test bench generator of FloPoCo, which compares outputs to the exact results computed using GNU MPFR. The test bench consists of random tests and some directed tests as well as exhaustive corner-case checks. Special cases and flag behaviour are extrapolated from the IEEE standard for fused operations. The directed tests include:

- Testing cancellations by forcing all the inputs to have the same exponent after the product: $E_{X_0} + E_{Y_0} = E_{X_1} + E_{Y_1} = \ldots = E_{X_{N-1}} + E_{Y_{N-1}} = E_Z$, where some products are randomly negative.

- Forcing inputs to have pairs of equal exponents after product: $E_{X_0} + E_{Y_0} = E_{X_1} + E_{Y_1}, \ldots, E_{X_{N-2}} + E_{Y_{N-2}} = E_{X_{N-1}} + E_{Y_{N-1}}$, where one product is negative and another is positive.

- Forcing M_0^* to be a product between normal and subnormal: $E_{X_0} = 0$ and $E_{Y_0} \geq E_{X_1} + E_{Y_1}, \ldots, E_{Y_0} \geq E_{X_{N-1}} + E_{Y_{N-1}}, E_{Y_0} \geq E_Z$.

The FDP2A$_{\text{FP32}}$ and FDP2A$_{\text{FP64}}$ operators have also been validated with a directed framework which tests against a golden model implemented with Sollya [12]. This framework thoroughly explores catastrophic cancellation cases with different significands that multiply to the same number, between Z and a product, different multi-sticky issues, and the use of subnormals. This adds an extra half a million tests.

7.5 Experimental Results

In this section, three variants of the FDPNA operators are compared: the Full-precision fixed-point accumulation from Chapter 5 extended with FP32 Z and R (denoted *Full*), this chapter's approach as per Section 7.3 (denoted *Compressed*) and a reimplementation of Tao et al. [116] adapted to handle subnormals (denoted *Tao*). The main difference between *Tao* and *Compressed* is that the former uses a sequential algorithm to compute the mantissa shift values, whereas *Compressed* uses a parallel prefix computation.

All these FDPNA operators have been synthesised with the Synopsys Design Compiler NXT for the TSMC 16FFC node.

7.5.1 Synthesis Without Pipelining

The first target frequency is set to 1 MHz in order to explore the synthesis trade-off between areas and combinatorial delays. The results appear in Table 7.1a and are plotted in Fig. 7.13, along with the size of the internal additions. There is always a threshold in N above which there is no compression ($w_{\text{compressed}} > w_{\text{full}}$). This threshold is 2 in the homogeneous FP16 case, about 10 in the homogeneous FP32 case, and above 16 for the other formats studied.

In full-size operators, the size of the internal adder does not depend on N. The rest of these operators essentially grows linearly with N. In the compressed operators, the adder sizes increase linearly with N, although extra pre/post-processing may be compensated by the smaller shifters (see Fig. 7.12).

The parallel prefix computation of shift values is not always beneficial. In particular it is not justified for $N = 2$.

Table 7.1: Synthesis results for TSMC 16 nm. A is the area in μm^2, D is timing in ns.

(a) Synthesis results for 1 MHz without pipelining.

		FP16				BF16 → FP32				FP32				FP64			
N		2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
Full	A	1370	**2247**	**3819**	**7209**	5305	7608	11117	19572	7036	10938	17602	**32411**	50724	72350	107090	187916
	D	9.0	**9.1**	**9.12**	**9.35**	32.51	32.99	32.86	**32.83**	35.34	36.02	34.83	**35.44**	209.72	210.44	210.44	210.49
Comp.	A	**1193**	2599	5520	12360	**1298**	**2723**	**6503**	**17972**	**3136**	**6659**	**15128**	39511	**9500**	**19713**	**43078**	**104494**
	D	7.97	12.02	15.28	16.95	10.11	14.26	21.54	33.59	13.57	19.03	29.97	52.05	**23.47**	34.69	56.16	**99.58**
Tao	A	1211	2623	8354	62941	1310	3072	11382	87723	3151	7081	20198	104455	9507	20272	49329	191076
	D	**7.51**	10.54	11.96	14.52	**9.42**	**13.66**	**21.0**	35.39	**13.35**	**18.43**	**29.75**	40.12	24.61	**34.05**	**55.54**	100.05

(b) Synthesis results for 1 GHz with pseudo-pipelining.

	FP16				BF16 → FP32				FP32				FP64			
N	2	4	8	16	2	4	8	16	2	4	8	16	2	4	8	16
n	3	3	3	3	3	4	4	5	3	4	5	6	4	5	5	6
Best	Tao	Full	Full	Full	Tao	Comp.	Comp.	Full	Tao	Comp.	Comp.	Full	Tao	Comp.	Comp.	Comp.
Area	1674	2949	4971	9202	1975	3789	11507	25464	4980	9240	19873	40227	12507	24524	56061	132049

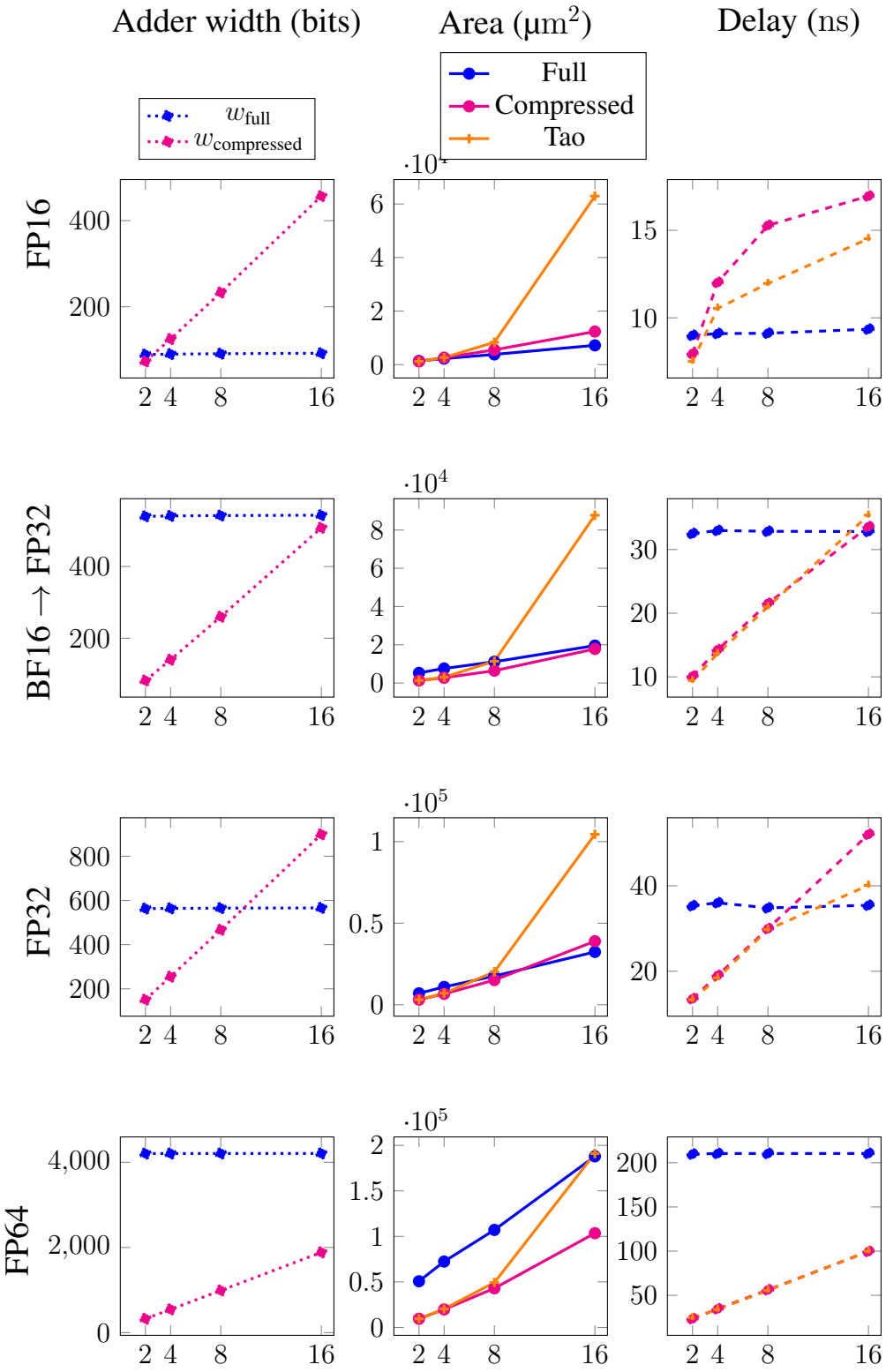


Figure 7.13: Combinatorial synthesis results as a function of N.

7.5.2 Synthesis With Pseudo-Pipelining

In the context of a pipelined floating-point unit (FPU), the operators are under a much stronger timing constraint. To evaluate this situation, the synthesis constraints of n-stage operator pipelining is approximated with *pseudo-pipelining*, that is, single-cycle synthesis at $\frac{1}{n}$ the target FPU frequency. The approximation is that the reported results do not account for the cost of the pipeline registers. Results are reported in Table 7.1b. The number of pseudo-pipelining cycles n in Table 7.1b is selected for each configuration as the area/latency trade-off most relevant for FPU design. Wherever two solutions were close in Table 7.1a, both were re-synthesised for the same n, and the best is reported.

7.6 Correctly Rounded Dot Products for $N = 2$

The case of the FDMDA ($N = 2$) has a particularly interesting trade-off. The complexity of the operator is not egregious to the point it is prohibitive, and it enables to accelerate and increase the accuracy of computation in multiple contexts.

7.6.1 Complex Arithmetic: Accuracy of FFT Twiddle Factor Recurrences

A complex fused multiply and add $R = XY + Z$ is implemented with the best possible accuracy in only two operations: Noting $j^2 = -1$,

$$R = R_{\text{re}} + jR_{\text{im}} \text{ with } \begin{cases} R_{\text{re}} = \circ(X_{\text{re}}Y_{\text{re}} - X_{\text{im}}Y_{\text{im}} + Z_{\text{re}}) \\ R_{\text{im}} = \circ(X_{\text{re}}Y_{\text{im}} + X_{\text{im}}Y_{\text{re}} + Z_{\text{im}}) \end{cases}.$$

An illustration of practical importance of the complex FMA is the on-line computations of the twiddle factors of Fast Fourier Transforms (FFT) [9]. The twiddle factors of a N-point DFT are defined as $W_P^k = e^{\frac{-2j\pi k}{P}}$, with P some power of 2 smaller than N and $k \in [0, P-1]$. They can be computed offline with the results stored in a table, or online using the recurrence $e^{j(k+1)\theta} = e^{jk\theta} \times e^{j\theta}$ with $\theta = -\frac{2\pi}{P}$. Such a sequence of n multiplications by $e^{j\theta}$ may lead to $O(n)$ error [117], so a twiddle factor recurrence should be implemented carefully. Rewriting $e^{j(k+1)\theta} = e^{jk\theta} + e^{jk\theta}(e^{j\theta} - 1)$ with $e^{j\theta} - 1 = -2\sin^2\frac{\theta}{2} + j\sin\theta$ avoids the cancellation in $cos\theta - 1$ [109] since θ is small for large-N FFTs. Computing $e^{j(k+1)\theta}$ then becomes a complex multiply-add that is accurately implemented by the FDMDA operators.

Fig. 7.14 displays the maximum and average errors for the FMA-based and the FDMDA-based twiddle factor recurrences in $\log_{10}$ scale, for N spanning successive powers of two. The reference values are twiddle factors computed using the `libm` standard `cos` and `sin` functions, rounded to FP32 from FP64. The error is defined as the modulus of the difference between a value and the baseline using FP64 arithmetic. Since twiddle factors are roots of unity, these errors are both absolute and relative. The FMA-based recurrence errors grow up to two orders of magnitude larger than those of the FDMDA-based recurrence errors.

This study was conducted in the specific case of the Kalray's MPPA [40].

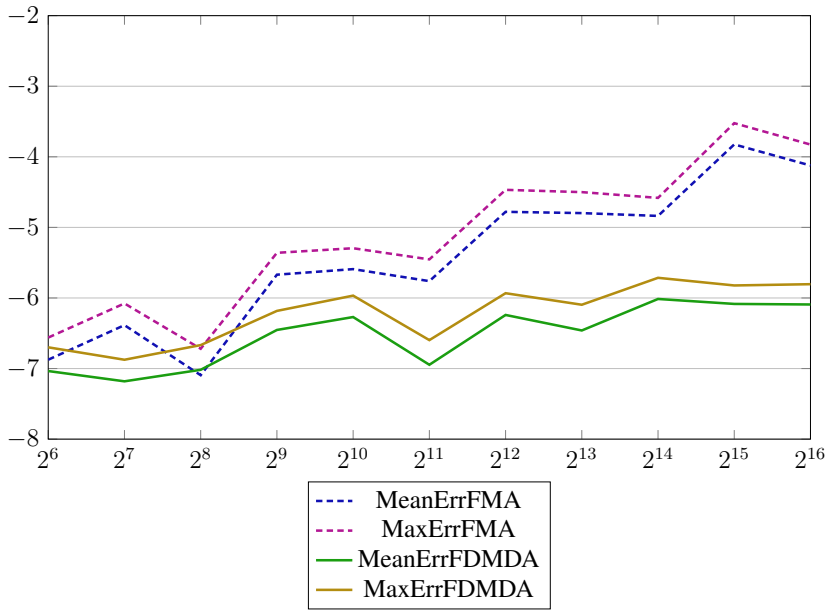


Figure 7.14: Base-10 logarithm of errors for the FMA-based and the FDMDA-based radix-2 twiddle factor FP32 recurrences depending on the FFT size.

7.6.2 Improving Performance and Accuracy for Other Applications

Some of the various uses of this operator have been explored in [65].

Let u be value of the ulp, used to compute the error of a computation.

In the field of complex arithmetic, the FDMDA can perform a correctly rounded complex FMA. It also computes more accurately a complex division, with an error under $3u$ where using an FMA could result in cancellations, and of the norm $\frac{3}{2}u$ instead of $2u$.

For error-free transforms, the FDMDA can compute the error of an FMA in one instruction, and accelerates the TwoSum algorithm from six instructions to two. It improves the speed and accuracy for various double-word operations : adding a double word and a float, adding two double words, multiplying a double word and a float (which is used for the multiplication by a constant), multiplying two double words, dividing two double words and computing the square root of a double word.

The FDMDA can be used to compute the correct rounding of the product of three floating-point numbers, and the product of four floating-point numbers with an error of at most u. This can be used to compute accurately the sign of the discriminant for cubic and quadratic equations.

Using an FDMDA instead of an FMA to compute a dot product of size N approximately halves the error. It also improves the compensated sum algorithm. The FDMDA can be used to accelerate polynomial evaluation.

7.7 Conclusions

Architectures of exact fused dot product add operators with $N + 1$ floating-point operands and a compressed accumulator perform better than those with a Kulisch-like full-size accumulator when the floating-point range is large and N is small.

The FP16 format has a large precision with respect to its range, so the full size approach performs better except for $N = 2$. The FP32 format has comparatively less precision with respect to its range, so the full size approach only becomes relevant for $N > 8$.

The FDMDA operator ($N = 2$) is included in the Kalray core to accelerate computing with complex numbers. This has uses for FFTs but also network applications like 5G acceleration. It is implemented for FP32 numbers in 4 cycles at $1.56\,\mathrm{GHz}$ in a $4\,\mathrm{nm}$ node, and a FP64 operator might also be included (which would take 5 cycles).

This operator is quite slow compared to an FMA and could be accelerated by using a Leading Zero Anticipator (LZA). Such a component can guess the Leading Zero Count in parallel with the final addition, possibly off by one position. This subcomponent is larger in area than an LZC, but would significantly reduce the latency of the operator.

In the evaluation presented in this chapter, since N is potentially large, the terms of the sum are converted into two's complement representation before being added. For $N = 2$, it might be faster and cheaper to use a sign-magnitude representation instead. Due to the shape of the bit heap, it is not clear which way of representing the sign will be cheaper, and would require an evaluation of both options.

Having explored various dot product architectures adapted to various situations, the following chapters will focus on function implementations.

Part III

Function Implementation

Chapter 8

Hardware Implementation of Numerical Functions

This chapter focuses on various methods of hardware implementation of numerical function. It mostly describes state of the art implementation methods, but contains some contributions from before my PhD started and some minor contributions during my PhD.

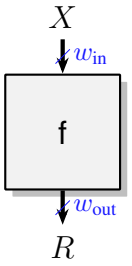


Figure 8.1: Target architecture for hardware function implementation.

The architectures in this chapter (Fig. 8.1) implement the numerical function $f(x)$ on a fixed-point input of width w_{in}, and on a fixed-point output of width w_{out}.

Without loss of generality, the function f can be considered as in from $[0, 1)$ to $[0, 1)$ by shifting the intervals and scaling the fixed-point numbers. This simplifies the approximation and implementation of the functions as $X, R \in \text{uFix}(-1, -w_{\text{in}})$.

Elementary functions, in particular exponential, logarithm, square root and trigonometric functions, are crucial for many scientific computing applications. Software implementations of elementary functions [94] rely on sequential algorithms that cannot always be implemented with a large throughput. They also often use large tables stored in the RAM (for either pre-computed values or polynomial coefficients) that are slow due to memory access. Hardware acceleration can achieve higher throughput for functions used in computationally intensive applications, for example sine and cosine in Fast Fourier Transforms. Hardware acceleration of less used functions can be implemented for *dark silicon* purposes (see Sec. 1.2.6).

In machine learning (Fig. 8.2), functions are used for scaling and activations. Some of the scaling is multiplying by a power of two, which is trivial for floating-point numbers. Sometimes, functions are used, like softmax or square root (see Sec. 3.1.1) when normalising vectors with the euclidean norm.

Popular activation functions (see Sec. 8.3) include scalar functions like ReLU, $\tanh$, ... In some contexts, softmax is considered like an activation function.

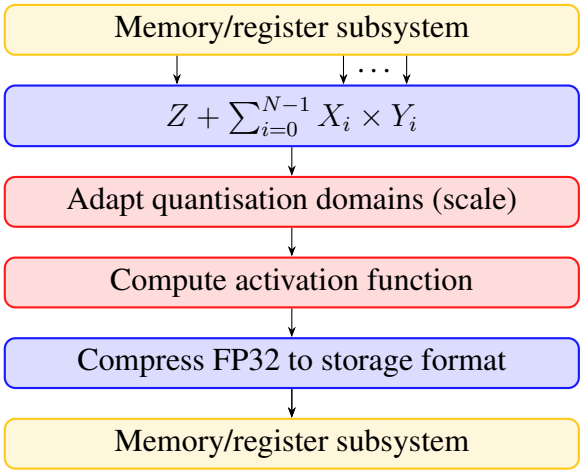


Figure 8.2: Architecture of steps for ML. In red the operators addressed in this part.

8.1 State-of-the-art Implementations

Techniques used to implement numerical functions vary greatly depending on w_{in} and w_{out}.

For small precisions, it is possible to use a direct implementation technique using tables. For larger precisions, implementations based on approximated functions must be used.

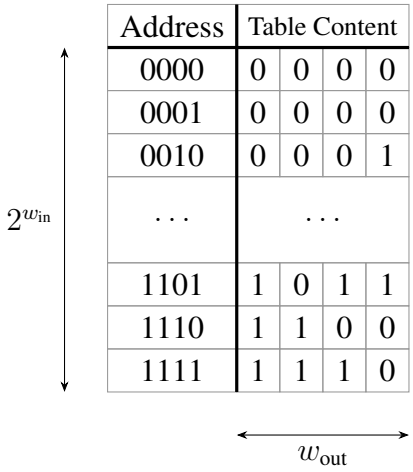


Figure 8.3: Example of Tabulation.

8.1.1 Table Implementations

Plain Tabulation

The simplest method is to store all the values of the function in a table (Fig. 8.3). In this case, the input to the table X is an address, and the data at this address is the value of the function rounded to the output format. The size of this operator in the number of bits stored is exponential in the input size: $w_{\text{out}} \times 2^{w_{\text{in}}}$, making it suitable for small to medium tables.

This technique provides correct rounding and is very fast. It also benefits from extra optimisations from the synthesis tools, which view such tables as a truth table.

This method is particularly suited for FPGAs.

Sometimes trying a more complicated approach results in a more expensive operator. For instance in [118] the reported cost for 5-bit in, 6-bit out sigmoid function is 25 LUT4.

The area unit of FPGAs is the LUTk, for Look Up Table with k inputs with $k \in \{4, 5, 6\}$ for mainstream FPGA families. A tabulated function for $w_{\text{in}} = k$ costs exactly one LUTk per output bit. For $w_{\text{in}} = k + 1$, it costs 2 or 3 LUTk per output bit (depending on the availability of a dedicated multiplexer in the FPGA cell). On modern FPGAs, 8-bit functions can be implemented in less than 4 LUT6 per bit.

The implementation of the 5-bit in, 6-bit out sigmoid function using a plain table would use 3 LUT4 per output bit, or a total of 18 LUT4 (and probably less if the synthesis tool find any optimisation to perform).

Differential Compression

Lossless Differential Table Compression (LDTC) is an exact compression method for tabulated functions which has the potential to reduce by up to 60% the number of bits stored at the cost of one addition [63, 14]. In the example described in paper [14] (Fig. 8.4), a table with $w_{\text{in}} = 8, w_{\text{out}} = 19$ storing 4864 bits was compressed into two smaller tables storing a total of 3808 bits, at a cost of an addition on 7 bits.

This compression separates the values of the large table into T_{ss} the table of sub samples, containing the most significant bits of the result, and T_d the table

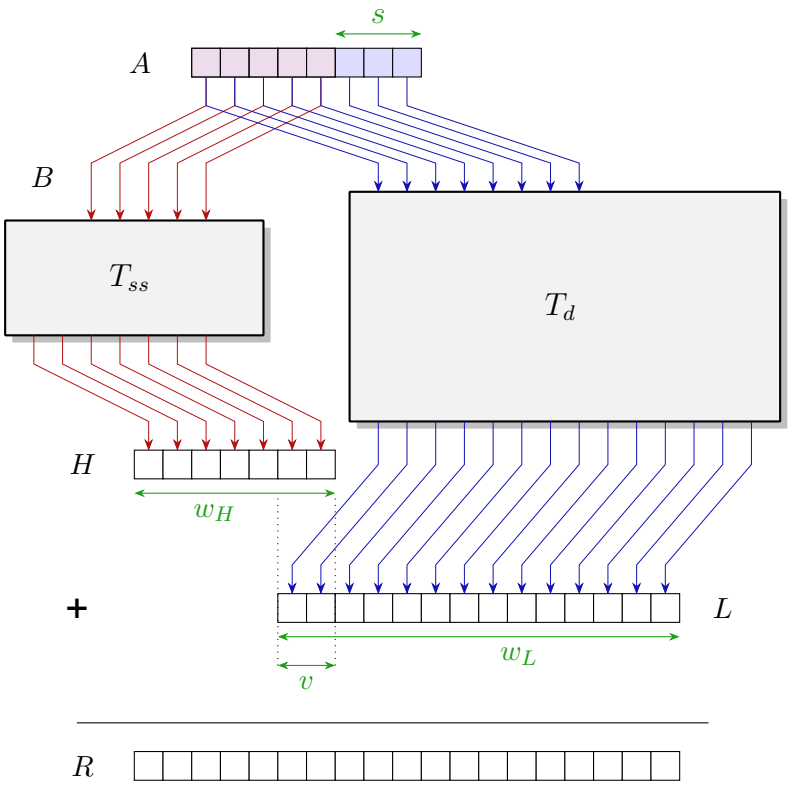


Figure 8.4: Lossless Differential Table Compression. Figure from [14].

of differentials, that adjust the sub sampled value such that the sum results in the target value. This technique works well for table values containing values that are monotone or close to each other.

8.1.2 Analytical Methods

The near entirety of implementations of numerical function use analytical methods to approximate the numerical function. Those methods are based on analytical considerations on the function.

The implementation of numerical functions happens in two steps, with two sources of error. First, the function must be approximated by another function whose implementation relies only on operators that can be easily built in hardware, typically sums and products. Then, this approximation is implemented in hardware.

Polynomial Approximation

Polynomial approximation is among the most reliable techniques for larger target precisions.

The requested function is first approximated with a polynomial function. The Sollya [12] tool assists in this step, providing state of the art approximation techniques. The difference between the real function and its mathematical approximation is called the *approximation error*. Sollya also assists in evaluating this error.

This approximation can then be evaluated in hardware. This implementation performs multiple operations with intermediate rounding, resulting in *implementa-*

tion errors which can be mitigated by adding g guard bits to the LSBs during the computations.

Since extra bits were added, a final rounding must be performed, incurring an extra *rounding error* of $\frac{1}{2}$ulp. For this reason, the analytical methods cannot produce a correctly rounded architecture.

The faithful rounding can be guaranteed by FloPoCo [34]. The number of guard bits and the precision of the coefficients are implementation-specific parameters determined FloPoCo to make sure the sum of the three errors, *approximation, implementation, rounding*, is below 1ulp.

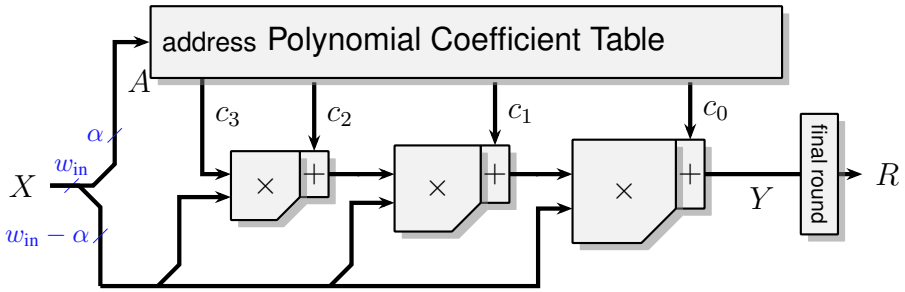


Figure 8.5: Horner Scheme for piecewise polynomial evaluation. Figure from [34].

For most functions, plain polynomial approximation uses a very large degree polynomial. It is often much easier to use break down the interval, and use multiple smaller degree polynomials. Tables are used to store the different coefficients for each table, and the polynomials are evaluated using a Horner Scheme (Fig. 8.5).

While the intervals could be arbitrary in number and size, FloPoCo implements a uniform piecewise approximation, where the number of intervals is a power of two. This enables a cheap indexing of intervals: the most significant bits of the input X make the address A of the interval.

Multipartite Method

The most used polynomial approximation technique is piecewise linear. In the example Figure 8.6, the address of the interval is on $\alpha = 4$ bits, resulting in $2^\alpha = 16$ intervals. For each interval at address A, the position within the interval $x_{-5}x_{-6}$ is multiplied by the first degree coefficient $c_1(A)$ and added to the constant coefficient $c_0(A)$: $R = c_0(A) + c_1(A) \times x_{-5}x_{-6}$.

Since multiplication is so expensive, it is not a big leap to replace the computations $c_1(A) \times x_{-5}x_{-6}$ by a pre-computed table (Fig. 8.7). This technique is called the bipartite method (two tables), invented in the specific case of the sine function [113], and independently re-invented in the specific case of the reciprocal function [19]. The table of the constant coefficients is now called the Table of Initial Values (TIV) and the table containing the pre-computed multiplication of the coefficient of degree one to the interval position is called the Table of Offsets (TO).

A series of improvements generalised this technique to arbitrary functions and to more than two tables [107, 111, 93, 111, 93, 22, 63, 62]. The multipartite method replaces the tabulation of the values of a function by an architecture that sums the output of several tables indexed by various subsets of the input bits (Figure 8.8).

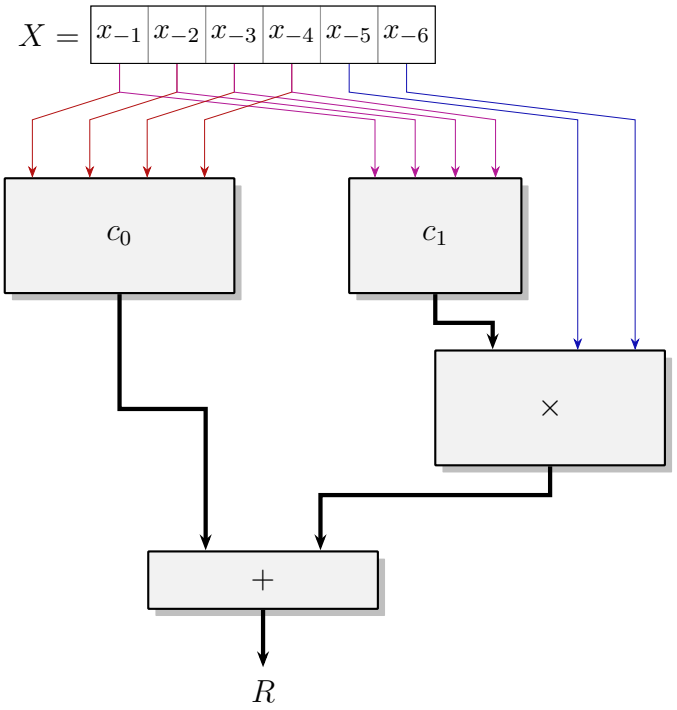


Figure 8.6: Architecture for piecewise linear approximation.

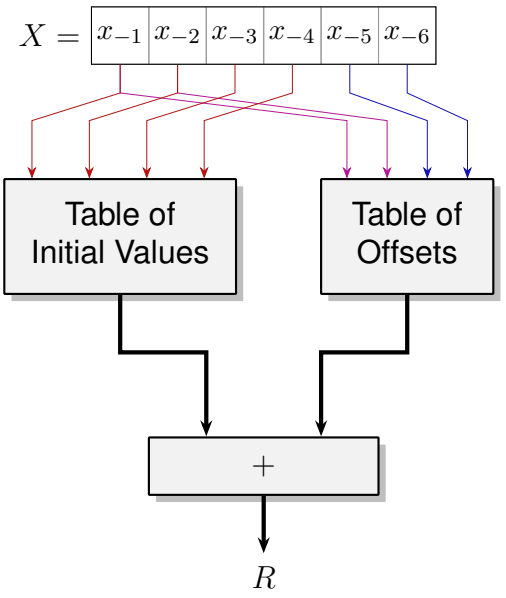


Figure 8.7: Example bipartite architecture.

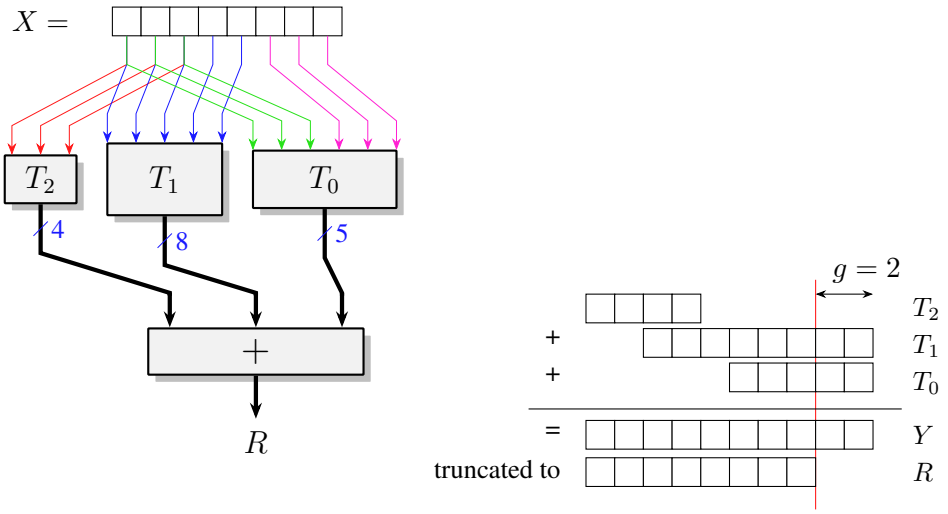


Figure 8.8: Example multipartite architecture.

The construction of the multipartite table minimising table size has been studied in [22, 63]. It is an analytical process that starts with piecewise linear approximation, and can only ensure faithful rounding. The following section presents a different method of construction multipartite tables that can ensure correct rounding.

8.2 Multipartite Construction Using Integer Linear Programming

Instead of using analytical methods, the optimisation of multipartite tables can be formalised using Integer Linear Programming (ILP) so that generic solvers can be used.

8.2.1 Properties of the Multipartite Decomposition

When using solvers, what matters is the resulting architecture, an example of which is shown in Figure 8.8, and the content of the tables, plotted in Figure 8.9. For each point of the horizontal axis of Figure 8.9, the sum of the content of the three tables that are accessed for this point provides a very good approximation to the function.

These figures illustrate various opportunities to reduce the table size that have been exploited in previous works.

First, the value stored in T_2[1] is a multiple of 16/256, which means that its trailing zeroes do not need to be stored. In the general case, the same can hold for other tables, but it is not exploited in this small example.

The multiple tables are not addressed by all the bits of the input. The table T_2 only holds 8 distinct values, addressed by the 3 leading bits of X (Figure 8.8). In Figure 8.9 only one value of T_2 is stored for each horizontal line. The same holds for T_1[2], which only holds 32 values (addressed by the 5 leading bits of X). Table

[1] In previous works, this would be called the sampling table of the compressed TIV.
[2] In previous works, this would be the differential table of the compressed TIV.

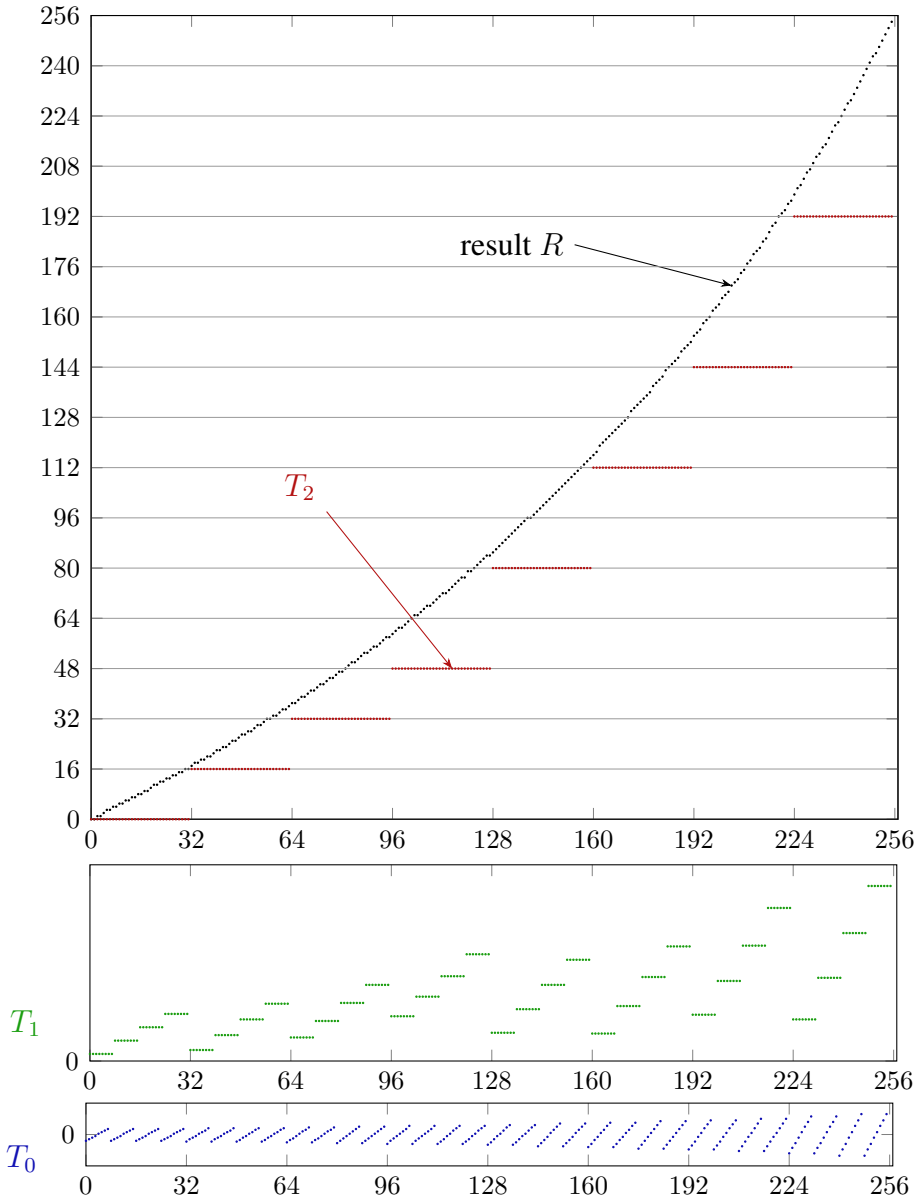


Figure 8.9: Tables for an approximation to $(\frac{2}{2-x} - 1)$.

T_0[3] only holds 8 segments, each of them being repeated 4 times. As each segment contains 8 values, T_0 altogether holds 64 values. This corresponds to inputting to T_0 the 3 leading bits of X (indexing the set of repeated segment) and the 3 LSB bits of X (indexing the value in the segment).

Since each table incurs an additional approximation, a classical technique is to perform to computation in extended precision ($g = 2$ bits on the example of Figures 8.8 and 8.9). The approximation and rounding errors accumulate in these guard bits, which are then simply dropped [22].

In this example, the total number of bits stored in tables is $2^3 \times 4$ (T_2) $+2^5 \times 8$ (T_1) $+2^6 \times 5$ (T_0) $= 608$ bits, much smaller than $2^8 \times 8 = 2048$ bits for a plain tabulation in a 8-bit in, 8-bit out table. This compression ratio improves with the input/output size.

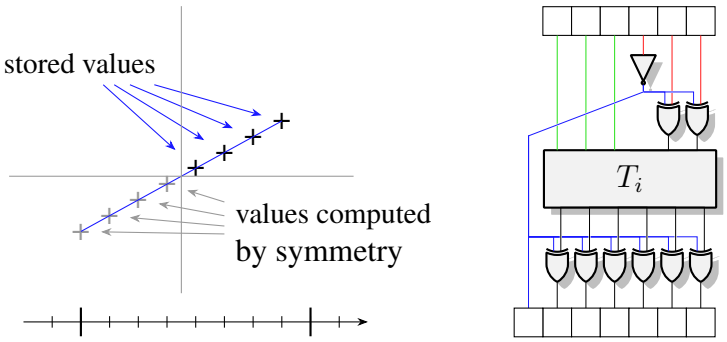


Figure 8.10: Using symmetry to trade one table input bit for two rows of XOR gates.

As most of the tables (on our example T_1 and T_0) contain piecewise linear approximations, another table compression opportunity is to exploit the symmetry of each line segment with respect to its centre [107]. This opens the possibility to replace one table of Figure 8.8 with the slightly more complex architecture shown in Figure 8.10. By removing one input bit to the table, this halves its size, but at the cost of two rows of XORs that implement the negation.

All this defines a fairly large implementation space, but an exhaustive enumeration of this space is tractable [63] for the sizes for which such architectures make sense, that is, for input/output sizes between 8 bits and 24 bits (for larger precision, the tables get really large, and higher-order approximations must be used anyway).

However, this exhaustive exploration of the parameter space is based on a worst-case error analysis that can only guarantee *faithful rounding:* the value returned by the architecture is not always the value nearest to the function value. Another equivalent point of view is that the error $\delta(x) = R - f(x)$ induced by the architecture is bounded by one unit in the last place (ulp) in a faithful architecture. Conversely, the error of a correctly rounded architecture is bounded by half an ulp.

Using an ILP model of multipartite architectures, inspired by comparable work on multiplier-based piecewise approximation [21], replaces analytical considerations on the function, as used in previous works, with a generic global optimisation of the table contents. This results in several minor improvements, in particular smaller values of g can be used. In this model, the lossless table compression of the TIV of

[3] In previous works, this would be a TO.

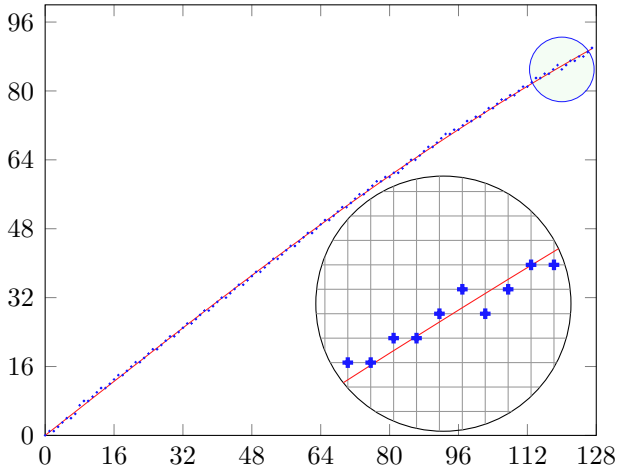


Figure 8.11: Non-monotonicity in a faithful approximation of $\sin(\frac{\pi}{4}x)$.

the most recent works [22, 63] becomes a special case of Tables of Offsets (TO), which simplifies the global optimisation problem.

A qualitative contribution is that this model also enables correctly rounded multipartite architectures. A correctly rounded architecture is much more expensive than a faithful one, and this is expected. However, another point of view is that multipartite architectures become a new approach for the lossless compression of a table of values in hardware. As such, they improve the state of the art in lossless compression [63, 14] by a very large margin. Also, an issue with faithful architectures is that they are not necessary monotonic [71], as illustrated by Figure 8.11. This issue disappears with correctly rounded architectures.

8.2.2 An ILP Model of Multipartite Architectures

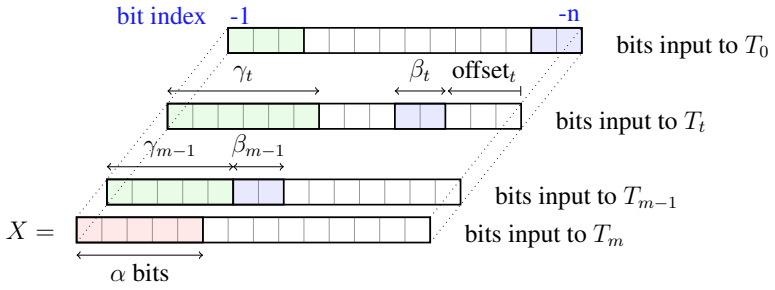


Figure 8.12: Multipartite input word decomposition. Figure from [24].

The input word decomposition used in this section (Fig. 8.12) is the same as the one used in the analytical method.

Ideally we would like to express as an ILP problem the choice of the architecture parameters as well as the choice of values to fill the tables, in such a way that an ILP solver could find the smallest architecture evaluating the function with the required precision. However, an ILP solver can only optimise linear problems. To circumvent this limitation, when a parameter has a non-linear impact on the problem, it will be

enumerated in a loop outside of the ILP, so that its value can be considered a constant inside the ILP.

The core of the method is to have $2^{w_{\text{in}}}$ constraints, one for each possible input value X in the fixed-point interval $[0, 1)$.

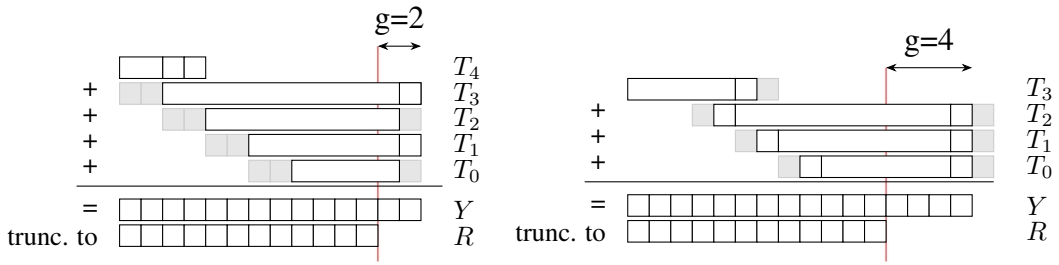
In each of these $2^{w_{\text{in}}}$ constraints, (C1) the input X is constant, thus the non-linear address $A_t(X)$ is also a constant, provided that the values for all the parameters involved (m, α, β_t, γ_t, sym_t) are previously fixed. An external loop is used to enumerate the values of these parameters, leading to multiple calls to the ILP solver.

As fixed-point numbers are implicitly scaled integers, the ILP formulation may use only the integers.

$$\forall X \quad \in \quad \{0, \ldots, 2^{w_{\text{in}}} - 1\} \quad Y_L(X) \quad \leq \quad R(X) \quad \leq \quad Y_H(X) \quad \text{(C1)}$$

The variable $R(X)$ (C1) represents the output of the architecture for the input X. The values $Y_L(X)$ and $Y_H(X)$ depend on the target rounding. In the case of faithful rounding, $Y_L(x)$ and $Y_H(X)$ correspond to the two values the function is allowed to take. In the case of correct rounding, $Y_L(x)$ and $Y_H(X)$ are both equal to the correctly rounded value of $f(X)$: In both cases, for a constant X, the bounds $Y_L(x)$ and $Y_H(X)$ are also constants.

The target function f is completely abstracted from the ILP, where only a target interval of possible values, Y_L and Y_H remain.



(a) Faithful rounding.

(b) Correct rounding.

Figure 8.13: Alignment for $\frac{2}{2-x} - 1$ on 12 bits. The ILP had the possibility to use the greyed bits but chose not to.

Multiple other constraints [24] model the functioning of the architecture, assuring that the truncated sum of all the tables is in the output interval. They also measure the size of those table, and enable to set up a cost function that minimise the number of stored bits, by removing guard bits independently for each table (Fig. 8.13. The grey squares in Fig. 8.13 represent available bits that were removed by the linear program in one specific example.

Solving the ILP Program

An external loop enumerates the non-linear parameter space, and for each parameter vector the solver works on a model that defines the linear parameters and fills the

table. This approach provides the optimal solution with respect to the cost model used, but it is quite slow: firstly, the parameter space is large, secondly the ILP problem is large, too, with more variables than there are bits in the tables. In practice this approach does not scale beyond $w_{\text{in}} = w_{\text{out}} = 12$.

The use of various heuristics like pruning the external loop and replacing part of the bit arrays with integers [24] improves solving time but sacrifices the guarantee of optimality.

Maximum number of guard bits

For faithful rounding, the formula $g = 1 + \lfloor \log_2(m-1) \rfloor$ from the literature [22] provides a safe value of g, in the sense that a solution with this value is guaranteed to exist (and the ILP sometimes finds solutions with fewer guard bits). However, it could happen in principle that an overall better solution exists with a larger g, so this is a heuristic choice.

For correct rounding, an empirical study for $w_{\text{in}} = w_{\text{out}} \in \{8, 10, 12, 14\}$ suggests the formula $g = 1 + (m-1) \times \lceil \log_2(w_{\text{out}}) - 2 \rceil$, so this formula is currently used in the code.

8.2.3 Results

The ILP construction and the outer loop are implemented in the Julia language with JuMP, interfaced to the Gurobi solver. The obtained tables are then imported in FloPoCo to generate the architecture, in particular the compression tree. For the small bit width addressed here, the resulting VHDL could be tested exhaustively for correct or faithful rounding using the (reliable and time-tested) FloPoCo test bench framework.

Those architectures are compared in detail in [24], using synthesis results obtained using Vivado v2022.1 for the Xilinx Kintex7 target.

This method of filling the tables is much slower than the previous multipartite and table compressions methods. It takes a few seconds for small sizes (8 and 10 bits), a few minutes for 12 bits, and a few hours for 14 bits. The computation is slower for faithful rounding compared to correct rounding. This method obviously scales poorly to larger sizes. Moreover architecture that is optimal in table size may no longer be optimal when the operator are synthesised.

With the ILP method, correct rounding is at best twice as expensive as faithful rounding in area, significantly reducing the difference of size between those rounding options.

Since the function is abstracted out of the ILP model, this method could be used for the lossless compression of other sort of tables, similarly to LDTC. This idea will be used in Chapter 9. However, this would require an adjustment of the decomposition loop and the guard bit computation, as heuristics are not likely to work on other types of tables.

8.3 Case Study: Activation Functions for Machine Learning

When building a network in machine learning, the choice of the activation function used will greatly impact the performance and cost of a network [39]. Finding new activation functions, either better or cheaper to implement, is an active field of research.

Many of those are variants based on these 5 common activation functions.

- Hyperbolic tangent: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}}$
- Rectified Linear Unit: $\text{ReLU}(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
- Exponential Linear Unit: $\text{ELU}(x) = \begin{cases} -(e^x - 1) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$
- Gaussian Error Linear Unit: $\text{GELU}(x) = \frac{x}{2}(1 + \text{erf}(\frac{x}{\sqrt{2}}))$
- Sigmoid Linear Unit (or Swish-1): $\text{SiLU}(x) = x\sigma(x)$

$\text{expm}(x) = e^{-x}$, $x \geq 0$ is also often implemented in hardware machine learning accelerators to compute the softmax function.

8.3.1 ALPHA

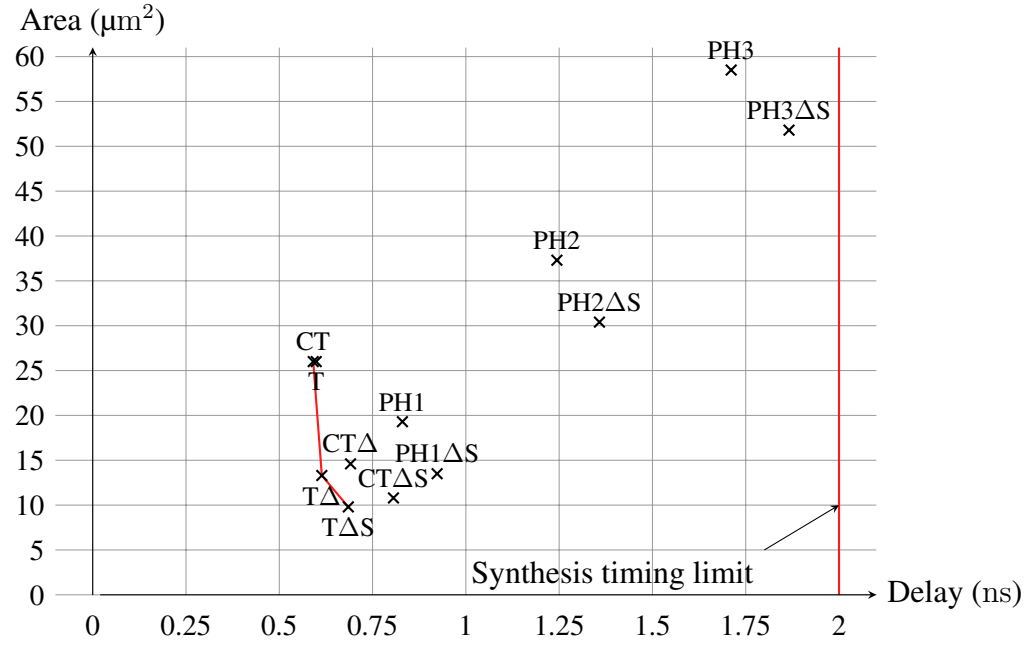
FloPoCo offers a framework called ALPHA [64] (for *Activation in Low Precision with High Accuracy*) for the fixed-point implementation of these common activation functions.

It implements various optimisations on each function like symmetry (noted S in the results) or a reduction from the ReLU function (Δ). The resulting function is then approximated using tables (T), compressed tables using LDTC (CT), multipartite tables (MPT), and a Piecewise Horner scheme of degree 1, 2 and 3 (PH1, PH2, PH3).

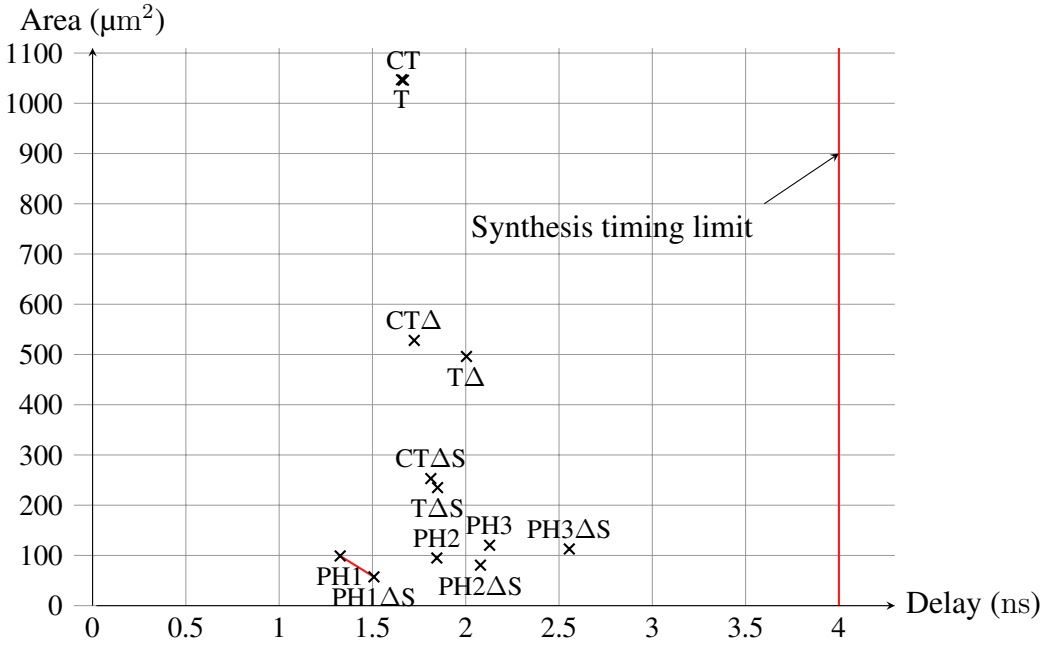
This frameworks allows easy generation of those functions for testing purposes, comparing various methods depending on the target architecture. This comparison work was carried out for VLSI using Synopsys Design Compiler NXT for the TSMC $4\,\text{nm}$ technology node.

As an example on the SiLU function (Fig. 8.14), all the methods are synthesised for a target precision and frequency. Once plotted, the best trade-off for the given synthesis parameters appears.

This best case was determined for each of the six functions offered by ALPHA (Tab. 8.1).



(a) 10 bits synthesised for 500 MHz.



(b) 16 bits synthesised for 250 MHz.

Figure 8.14: VLSI Synthesis for SiLU.

Table 8.1: Comparison of various approximation methods for common activation functions.

w	tanh	σ	GeLU	ELU	SiLU	e^{-x}
8	T 6.6 / 32	T 10.0 / 41	TΔ 3.0 / 42	TΔ 2.3 / 28	TΔS 4.1 / 49	T 9.0 / 35
12	PH1S 35.4 / 100	PH2 69.8 / 100	TΔS 17.8 / 89	PH1Δ 17.6 / 74	PH1ΔS 22.7 / 100	PH2 28.7 / 100
16	PH1S 146.8 / 100	PH2 99.0 / 100	PH1Δ S 55.7 / 100	PH1Δ 73.1 / 100	PH1ΔS 69.3 / 100	PH2 58.8 / 100

Each entry reads: best method, area in μ^2 / % of cycle

8.3.2 Activation Function Implementation in Kalray's Accelerator

Activations in Software

The plain table method is often used in software if no hardware is present to accelerate the implementation. The whole table of results is stored in the cache, and memory accesses are used as function calls. This of course requires the cache to be large enough for all the function values, otherwise this method can be very slow.

For Kalray, this is the implementation that must be beaten to be able to add hardware function implementations in the accelerator. This operator:

- is free in terms of area,
- can implement any functions, changeable at runtime,
- has an input size of up to 16 bits, for any data type (fixed-point, floating-point, or any other exotic format), and an output size of up to 256 bits,
- BUT can compute two values per cycle,
- **BUT** blocks the Load-Store Unit.

Kalray chose not to include any of the implementations of activation functions as optimised in Section 8.3, as they do not beat the table based implementation for two reasons.

The Limited Choice of Functions :

Machine Learning research moves extremely fast. It is hard to predict which activation functions will be used by the time the chip is on the market, in two to three years. One function is cheap to implement, but ALPHA identifies six popular activation functions, making the Special Function Unit (SFU) already much more expensive. If some of these functions become obsolete in the future, their implementation is wasting silicon.

The software implementation does not have any of those limitations, as the table for any function can be loaded at runtime, making it future-proof.

A General Distrust for Fixed-point Arithmetic

While Neural Networks work perfectly with smaller and smaller floating-point formats, fixed-point formats do not seem to work as well to store weight and activations. It is not clear how the activation functions of trained networks are implemented, and how the networks will react if the implementation changes. Using too much precision is a safe way of avoiding this problem. Machine Learning researchers are also attached to representing accurate zeros and near-zero values, keeping them true to floating-point representations.

The software implementation is also a clear winner, as it can take any types of input and output, including floating-point numbers.

Floating-point Functions

The only situations where a software table is not the best solution is for steps that make use of the LSU, or that require a better throughput than two function evaluations per cycle.

The softmax function is computed using the exponential of every coordinate of the vectors requiring $N+1$ evaluation of the exponential function for a vector of size N, making it's hardware acceleration competitive compared to the software table implementation (Chap. 10).

In the case of the reciprocal square root function (Chap. 9), the gain is less straightforward. When using it not normalise a vector of size N, only one inverse square root is used per vector. The hardware acceleration is interesting mainly when the vectors are small, for instance for graphical transformations ($N = 4$). It is also used for the batch normalisation when training convolutional neural networks, for batch sizes N typically between 32 and 128.

For this reason, the reciprocal square root function $\frac{1}{\sqrt{x}}$ is only partially accelerated in hardware, giving more freedom to the user to compute other functions from the same family instead : $\sqrt{x}$ and $\frac{1}{x}$. These more general purpose functions can then be used to help approximate various activation functions, for example $\tanh$.

Chapter 9

Combined Hardware and Software Acceleration for the Reciprocal Square Root Function

The square root function is an elementary function with wide-ranging applications, particularly in the field of vector mathematics.

One of the common variants is the reciprocal square root, used to compute the Euclidean norm of a vector during normalisation. In this process, each vector coefficient is divided by the norm to ensure that the resulting vector has a length of one. This normalisation is essential for various graphical transformations.

It is also used to accelerate training with batch normalisation (see Sec. 3.1.1).

Squaring the reciprocal square root can be used to compute the reciprocal, and multiplying it by the input to compute the square root.

The square root function and its variants are already hardware accelerated in most processors, including Kalray's where one square root function can be computed per cycle in the core. This leads to two issues. First, when intensive computations are carried out within the accelerator, going back and forth with the core for every computation is not ideal. Secondly, the throughput of one function approximation per cycle is too low for an accelerator that manipulates a vector of 8 FP32 numbers.

This leads to the need of a new accelerator-specific way of implementing the family of square root functions, using a mix of hardware and software accelerations.

9.1 Newton-Raphson Method for the Reciprocal and Square Root Functions

Figure 9.1: Babylonian tablet YBC-7289.

While the method's name *Newton-Raphson* suggests it was invented in the 16$^{\text{th}}$ century, traces of this method used in the computation of the square root appear in various older sources. Some Babylonian tablets from around the 16$^{\text{th}}$ century BC (Fig. 9.1) are engraved with a precise approximation of $\sqrt{2}$, and other tablets [51] show this method used to compute square roots. Greek Mathematician Heron of Alexandria describes the same method in his book *Metrica* [1] in the 1$^{\text{rst}}$ Century AD.

Mathematicians Raphson and Newton generalised this method to compute a root a of any function f: the number a such that $f(a) = 0$. It is an iterative method, and formula for each iteration is:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad ,$$

where f' is the derivative of f.

This method is used in modern computers to compute the reciprocal function, and the square root functions [16]. It can also be used for divisions.

This method is proven to converge quadratically, that is the number of "correct" bits doubles with every iteration. Starting with a good approximation of the root x_0 considerably speeds up the method by reducing the number of iterations.

9.1.1 Reciprocal Function

Let $a \in \mathbb{R}^*$. The value $\frac{1}{a}$ can be computed iteratively with Newton-Raphson's method, provided there is a function f such that $f(\frac{1}{a}) = 0$. Many functions satisfy this condition, for example:

$$f(x) = \frac{1}{x} - a \quad , \quad f'(x) = -\frac{1}{x^2} \quad .$$

The recursion to compute $\frac{1}{a}$ is:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{\frac{1}{x_n} - a}{-\frac{1}{x_n^2}} \\ &= x_n + x_n^2 \times (\frac{1}{x_n} - a) \\ &= x_n \times (2 - x_n \times a) \quad . \end{aligned}$$

9.1.2 Square Root and Reciprocal Square Root

Square Root

Let $a \in \mathbb{R}^{+*}$. The value $\sqrt{a}$ is a root of the function f:

$$f(x) = x^2 - a \quad , \quad f'(x) = 2x \quad .$$

The recursion to compute $\sqrt{a}$ is:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{x_n^2 - a}{2x_n} \\ &= \frac{x_n}{2} + \frac{a}{2x_n} \quad . \end{aligned}$$

This formula is not ideal as the iteration requires a division to compute $\frac{a}{x_n}$. Division is an expensive operation that is also computed iteratively, which would greatly increase the time used to compute $\sqrt{a}$.

A way to circumvent this problem is to use the Newton-Raphson method to compute $\frac{1}{\sqrt{a}}$ instead, and then use for $\sqrt{a}$: $a \times \frac{1}{\sqrt{a}} = \sqrt{a}$. It can also be used to obtain $\frac{1}{a}$ as $\frac{1}{a} \times \frac{1}{a}$.

Reciprocal Square Root

When trying to compute $\frac{1}{\sqrt{a}}$, a function that has said root is:

$$f(x) = \frac{1}{x^2} - a \quad , \quad f'(x) = \frac{-2}{x^3} \quad .$$

Then the recursion to compute $\frac{1}{\sqrt{a}}$ is:

$$\begin{aligned} x_{n+1} &= x_n - \frac{f(x_n)}{f'(x_n)} \\ &= x_n - \frac{\frac{1}{x_n^2} - a}{\frac{-2}{x_n^3}} \\ &= x_n + \frac{x_n^3}{2} \times (\frac{1}{x_n^2} - a) \\ &= x_n + \frac{1}{2} \times (x_n - x_n^3 \times a) \\ &= \frac{1}{2} \times x_n \times (\frac{3}{2} - x_n^2 \times \frac{a}{2}) \quad . \end{aligned}$$

9.2 Implementing Correct Rounding in Software

Those formulas work perfectly in the mathematical world, however on computers rounding errors makes the method less perfect.

9.2.1 Software Iterations: Using the Fused Multiply Add

In software, correctly rounded $\frac{1}{a}$ and $\sqrt{a}$ can be obtained if an FMA (Fused Multiply Add) is used [17, 16].

The computations are rearranged to reduce rounding errors. The iterations for the square root algorithm are rearranged into three recursive formulas: h_i is an approximation of $\frac{1}{2\times\sqrt{a}}$, g_i of $\sqrt{a}$ and r_i is the residue of the computation, and is closer and closer to 0.

The seed approximating $\frac{1}{\sqrt{a}}$ is called y_0.

Those values are initialised as:

$$\begin{aligned} y_0 &= \text{seed}(a) = \frac{1}{\sqrt{a}}(1+\epsilon) \\ h_0 &= 0.5 \times y_0 \\ g_0 &= a \times y_0 \\ r_0 &= -g_0 \times h_0 + 0.5 \quad . \end{aligned}$$

The recursions are:

$$\begin{aligned} h_{n+1} &= r_n \times h_n + h_n \\ g_{n+1} &= r_n \times g_n + g_n \\ r_{n+1} &= -g_{n+1} \times h_{n+1} + 0.5 \quad . \end{aligned}$$

Those iterations are still not enough to compute the correct rounding of $\sqrt{a}$. The error on g_m can be computed exactly with an FMA: $e = a - g_m \times g_m$. This enables to correctly round the result: $\circ(\sqrt{a}) = e \times h_m + g_m$.

When computing $\frac{1}{\sqrt{a}}$, the error term is $\frac{1}{a} - \frac{1}{4} \times h_m \times h_m$. Computing this error would require a fused operator including a division, which is not available. However, faithful rounding is achievable.

These algorithms work when none of those computations overflow or underflow the format used. Some strategies used to ensure this are described with the implementation (see Sec. 9.4.1).

Choosing a seed that is sufficiently precise to bootstrap the iterations is crucial to the convergence of the method. The seeds are generally stored in tables, which are either saved in the RAM or in hardware.

An interesting historical example of seed approximation is the Fast Inverse Square Root algorithm for FP32 numbers, used notably in the video game *Quake III Arena* (1999). The seed is computed by taking the FP32 a, and manipulating the encoding of a as an integer (Algo. 4).

Algorithm 4 Seed computation in the Fast Inverse Square Root

```
long i;
float a, seeda;
i = * ( long * ) &a;
i = 0x5f3759df - ( i » 1 );
seeda = * ( float * ) &i;
```

The computed seed is then refined using a Newton-Raphson iteration. The magic constant 0x5f3759df is attributed to Gary Tarolli [110].

9.2.2 Special Case: Significand is all ones

The software algorithm [17] for the reciprocal is not proven to be correct for in the case where the significand is all ones, which corresponds to the biggest number of a given exponent. In this case, the algorithm returns a faithfully rounded result instead of a correctly rounded one. This input is ignored in the construction of our operator, as either the computation does not need the extra precision (if for example in *fast-math* mode), or it must be corrected in software, potentially detected in hardware where a flag is returned.

9.2.3 Hardware Iterations: Existing Kalray Architecture

The square root $\sqrt{a}$ function is implemented in Kalray's core Special Function Unit (SFU) as a table of seed, followed by Newton-Raphson iterations both completely in hardware. This operator is pipelined in 12 cycles, and has a throughput of 1.

It uses standard techniques, starting with a table of seeds approximating the inverse square root $\frac{1}{\sqrt{a}}$ on 9 bits, and two Newton-Raphson iterations plus a correction step that computes the correct rounding of $\sqrt{a}$.

9.3 Seed Tables Redesigned

The proposed architecture removes a Newton-Raphson iteration by redesigning the seed table. Instead of a 9-bit seed, a $\approx$ 15-bit seed is used, and only one iteration of Newton-Raphson and the corrective step is needed to achieve correct rounding for $\sqrt{a}$ in FP32.

9.3.1 Specifications of the Seed Table

The specifications of the seed table are defined by which algorithms will be using it, and the expected result.

Square Root and Reciprocal Square Root for FP32

The algorithms used for FP32 are similar to the ones previously described. The algorithm for the reciprocal square root (Algo. 5) is expected to return a faithfully rounded result.

Algorithm 5 Computation of $\frac{1}{\sqrt{a}}$ in one Newton-Raphson iteration

$y_0 \Leftarrow \text{seed}(a)$
$h_0 \Leftarrow 0.5 \times y_0$
$g_0 \Leftarrow a \times y_0$
$r_0 \Leftarrow -g_0 \times h_0 + 0.5$
$o \Leftarrow r_0 \times y_0 + y_0$

The algorithm for the square root (Algo. 6) is expected to return a correctly rounded result. If the corrective iteration is removed, then it is expected to return a correctly rounded result.

Algorithm 6 Computation of $\sqrt{a}$ in one Newton-Raphson and one corrective iteration

//Newton-Raphson iteration
$y_0 \Leftarrow \text{seed}(a)$
$h_0 \Leftarrow 0.5 \times y_0$
$g_0 \Leftarrow a \times y_0$
$r_0 \Leftarrow -g_0 \times h_0 + 0.5$
$h_1 \Leftarrow r_0 \times h_0 + h_0$
$g_1 \Leftarrow r_0 \times g_0 + g_0$
//$g_1 \approx \sqrt{a}$ in faithful rounding
//Corrective iteration
$e_1 \Leftarrow -g_1 \times g_1 + a$
$o \Leftarrow e_1 \times h_1 + g_1$

The output to the seed operator is packed into an FP32 number so it can easily be manipulated in software.

Reciprocal for FP32

The same $\frac{1}{\sqrt{a}}$ seed could be squared and used as a seed for reciprocal. This requires a way to compute the absolute value of the reciprocal square root as well as conserve the sign for the squaring step: $\text{sign}(a) \times \frac{1}{\sqrt{|a|}}$, which are cheap manipulations of the sign bit.

Algorithm 7 Computation of $\frac{1}{a}$ in one Newton-Raphson and one corrective iteration

```
//Newton-Raphson (Markstein) iteration
y0 ⇐ seed(a)
x0 ⇐ y0 × y0
r0 ⇐ −a × x0 + 1
x1 ⇐ r0 × x0 + x0
//Corrective (Markstein) iteration
r1 ⇐ −a × x1 + 1
o ⇐ r1 × x1 + x1
```

The algorithm for the reciprocal function (Algo. 7) is expected to return a correctly rounded result. If the corrective iteration is removed, then it is expected to return a correctly rounded result.

Correctly rounded FP16 Reciprocal Square Root

A correctly rounded multipartite table implementation of $\frac{1}{\sqrt{a}}$ [24] on 10 bits requires 3 guard bits, resulting in an approximation on 13 bits in total. The 15-bit seed used for the FP32 implementation should be sufficient to compute the correct rounding on 10 bits of $\frac{1}{\sqrt{a}}$.

This constraint is added to the specification. If the input to the seed table is an FP16, then the rounded output seed is expected to be a correctly rounded $\frac{1}{\sqrt{a}}$. The output must packed in an FP16 format.

This specification is only for Round to Nearest ties to Even. To avoid expensive rounding logic, Round to Nearest ties to Away from zero is used in hardware, since ties are not possible [85]. This rounding mode is computed in hardware by adding half an ulp to the seed, and then truncating the significand.

Square root for FP16

Obtaining an FP16 correctly rounded square root for FP16 can be constructed starting with the correctly rounded $\frac{1}{\sqrt{a}}$. It is better for the throughput to start from the correctly rounded FP16 instead of using the FP32 seed, as the seed is packed into an FP32 format.

A first approximation $\sqrt{a} \approx a \times \frac{1}{\sqrt{a}}$ can be used. However, this does not result in a correctly rounded result. For the 2048 FP16 floating-point numbers in $[1, 4)$, for only 58 of them does this computation not result in faithful rounding. Empirical test of the error shows it is always below 1.4 ulp.

When using a refining iteration (Algo. 8), correct rounding is guaranteed for all inputs.

Algorithm 8 Computation of $\sqrt{a}$ in FP16 with one corrective iteration

$y_0 \Leftarrow \text{recsqrt}(a)$
$h_0 \Leftarrow 0.5 \times y_0$
$g_0 \Leftarrow a \times y_0$
//Corrective iteration
$e_0 \Leftarrow -g_0 \times g_0 + a$
$o \Leftarrow e_0 \times h_0 + g_0$

This specification is computed based on correctly rounded FP16 $\frac{1}{\sqrt{a}}$ and does not affect the seed table.

Reciprocal for FP16

Squaring the FP16 output $\frac{1}{\sqrt{a}}$ to obtain the reciprocal does not provide a correctly rounded result. For the 2048 FP16 floating-point numbers in $[1, 4)$, this computation not result in faithful rounding for 266 of them. Empirical test of the error shows it is always below 1.4 ulp.

Algorithm 9 Computation of $\frac{1}{a}$ in FP16 with one corrective iteration

$y_0 \Leftarrow \text{recsqrt}(a)$
$g_0 \Leftarrow y_0 \times y_0$
//Corrective iteration
$e_0 \Leftarrow -g_0 \times a + 1$
$o \Leftarrow e_0 \times g_0 + g_0$

Using a refining iteration (Algo. 9) does not guarantee a correctly rounded result, as two inputs (3.63281 and 3.99805) in $[1, 4)$ are only faithfully rounded. One of those two inputs corresponds to the situation described in section 9.2.2.

This specification is computed based on correctly rounded FP16 $\frac{1}{\sqrt{a}}$ and does not affect the seed table, requiring a software correction when correct rounding is required.

9.3.2 Constructing a Table that Satisfies the Specifications

As a summary, the seed table must satisfy multiple specifications:

- Using Algo. 5 outputs a faithfully rounded $\frac{1}{\sqrt{a}}$ in FP32,
- Using Algo. 6 outputs a correctly rounded $\sqrt{a}$ in FP32, and removing the correction iteration outputs a faithfully rounded result,
- Using Algo. 7 outputs a correctly rounded $\frac{1}{a}$ in FP32, and removing the correction iteration outputs a faithfully rounded result,
- Rounding the seed outputs a correctly rounded $\frac{1}{\sqrt{a}}$ in FP16.

The seed table is constructed such that a seed is deemed valid only if it satisfies those six constraints.

$$\frac{1}{\sqrt{a}} = \frac{1}{\sqrt{2^e \times 1.F}} = \begin{cases} 2^{-k} \times \frac{1}{\sqrt{1.F}} & \text{if } e = 2k, k \in \mathbb{Z} \\ 2^{-k} \times \frac{1}{\sqrt{2 \times 1.F}} & \text{if } e = 2k+1, k \in \mathbb{Z} \end{cases}$$

While the seeds can be computed analytically and proven to satisfy the conditions, it is much faster to exhaustively test the seeds, especially as argument reduction reduces the interval to test to $[1, 2)$. An algorithm using MPFR [50] can test the seeds for all the $8,388,608$ FP32 floating-point numbers in $a \in [1, 2)$ (even exponent), and another $8,388,608$ floating-point numbers $a \in [2, 4)$, which takes 12 seconds in total.

Determining the Size of the Seed Table

A preliminary test can determine the size of the seed table. Thanks to the quadratic convergence of the Newton-Raphson method, a seed of 12 bits should be sufficient to obtain 24 bits of precision (FP32) in one iteration.

The first test uses for an FP32 input a the seed $\frac{1}{\sqrt{a}}$ on w_{out} bits of precision, with $w_{\text{out}} \in \{11, \ldots, 16\}$. This test is equivalent trying a seed table with $w_{\text{in}} = 24$ input bits, and w_{out} output bits, and suggests that $w_{\text{out}} = 14$ is the smallest size where the specifications are satisfied.

The input of the seed table X is computed by truncating the significand of the FP32 input of the operator. When w_{in} is taken into account in the testing algorithm, extra output bits are required to meet the specifications.

The specifications are relaxed to only requiring $\frac{1}{\sqrt{a}}$ and $\frac{1}{a}$.

The smallest seed table size found is $w_{\text{in}} = 14, w_{\text{out}} = 15$, including the implicit bit of the floating-point format. Compared to the ideal quadratic convergence of the Newton-Raphson method, three bits are lost to rounding errors.

Computing an Interval of Suitable Seeds

Not all the inputs X of the table require the final $w_{\text{out}} = 15$ of output precision to find a suitable seed. This is reflected on the number of suitable seeds for each input. To ease the storage and the future approximation of this table, an interval of suitable seeds is computed instead of a list of all the seeds.

Table 9.1 shows the number of table inputs X that have an interval of suitable seeds of size N. The total number of inputs X adds to 2^13 instead of the expected $2^{w_{\text{in}}} = 2^14$ since the implicit bit is always 1. Including the implicit bit in the table size is interesting as the quadratic convergence takes this bit into account.

Table 9.1 shows that an instance of a seed table conforming to the specification is possible, as no inputs X have 0 suitable seeds.

The column $N = 1$ means that there is only one possible seed for 37 different inputs X, which means that for those inputs the seed requirement is very strictly on 15 bits. The columns for $N \geq 3$ show that over 85% of seeds have 3 different options, meaning that most seeds would work with 14 bits instead of 14.

Table 9.1: Number of table inputs X that have N suitable seeds for intervals $[1, 2)$ and $[2, 4)$.

	0	$N = 1$	2	3	4	5	6	7	8	$N \geq 9$
$[1, 2)$	0 0%	4 0.049%	3 0.037%	51 0.62%	183 2.2%	592 7.2%	919 11%	1357 17%	3068 37%	2015 25%
$[2, 4)$	0 0%	0 0%	3 0.037%	17 0.21%	55 0.67%	456 5.6%	3346 41%	4179 51%	131 1.6%	5 0.061%

This is a motivation to look for a seed table that is only precise when it is required, which should reduce the cost of the resulting architecture.

Padding the FP16 Input

When dealing with FP16 inputs, the significand is only 11 bits, but the input of the seed table is $w_{\text{in}} = 14$ bits.

Table 9.2: Number of table inputs X that have a number N of suitable seeds, depending on the number concatenated to the FP16 input

	$N = 0$	1	2	3	4	5	6	7	8	$N \geq 9$
Interval $[1, 2)$										
X (000_2)	0	36	332	1183	2053	3657	913	11	4	3
X $(001_2$	17	56	348	1136	2031	3688	898	11	4	3
X (010_2)	58	73	369	1102	1987	3675	911	11	3	3
X (011_2)	121	87	343	1108	1981	3619	916	10	4	3
X (100_2)	185	88	357	1093	1954	3601	898	10	4	2
X (101_2)	259	92	352	1076	1927	3584	884	12	3	3
X (110_2)	331	105	351	1044	1925	3531	887	12	4	2
X (111_2)	421	99	340	1040	1875	3513	887	11	3	3
Interval $[2, 4)$										
X (000_2)	0	1	34	515	3180	4064	396	2	0	0
X (001_2)	0	6	60	500	3153	4088	383	2	0	0
X (010_2)	9	40	59	502	3129	4059	392	2	0	0
X (011_2)	37	63	70	465	3146	4022	387	2	0	0
X (100_2)	86	71	65	467	3114	4003	384	2	0	0
X (101_2)	137	77	64	472	3076	3969	395	2	0	0
X (110_2)	195	75	65	475	3048	3958	374	2	0	0
X (111_2)	254	72	62	452	3078	3897	375	2	0	0

The FP16 input must be padded with three bits, and those must be a constant value. Experiments described in Table 9.2 tested all the different padding values, showing the best results when the input is padded with zeros.

9.4 Implementation

The implementation for Kalray's accelerator include a hardware acceleration of the seed table, which are then used for software Newton-Raphson iterations using the

vector FMAs operators.

This operator outputs a seed that satisfy the following specifications:

- Using Algo. 5 outputs a faithfully rounded $\frac{1}{\sqrt{a}}$ in FP32,
- Using Algo. 6 outputs a correctly rounded $\sqrt{a}$ in FP32, and removing the correction iteration outputs a faithfully rounded result,
- Rounding the seed outputs a correctly rounded $\frac{1}{\sqrt{a}}$ in FP16.

For both formats, it computes three variants in sign:

- $\frac{1}{\sqrt{a}}$ with a return of NaN if a is negative,
- $\frac{1}{\sqrt{|a|}}$,
- $\text{sign}(a) \times \frac{1}{\sqrt{|a|}}$.

9.4.1 Argument Reduction to a Fixed-Point Function

The floating-point input a of the operator can be reduced into a smaller fixed-point interval by computing on the exponent separately.

$$\frac{1}{\sqrt{a}} = \frac{1}{\sqrt{2^e \times 1.F}} = \begin{cases} 2^{-k} \times \frac{1}{\sqrt{1.F}} & \text{if } e = 2k, k \in \mathbb{Z} \\ 2^{-k} \times \frac{1}{\sqrt{2 \times 1.F}} & \text{if } e = 2k+1, k \in \mathbb{Z} \end{cases}$$

Those two cases correspond to approximating the function on two intervals, $[1, 2)$ and $[2, 4)$ (Fig. 9.2).

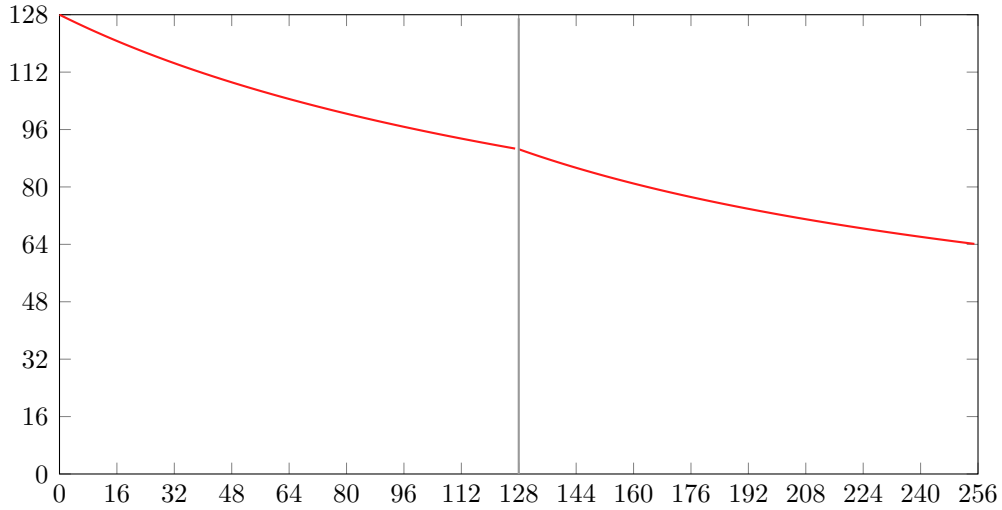


Figure 9.2: Function implemented after argument reduction, normalised.

If the exponent is even, then the function to be approximated is $\frac{1}{\sqrt{1+F}}$, with $F \in \text{uFix}(-1, -w_F)$. If the exponent is odd, the function is $\frac{1}{\sqrt{2+2F}}$, with $F \in \text{uFix}(-1, -w_F)$.

The validation of this operator can be done on all FP32 and FP16 inputs in $[1, 4)$, barring overflow and underflow issues.

9.4.2 Modifying the ILP model of Multipartite Tables

After argument reduction, the function is implemented on two intervals with each a constant exponent, which means that it is now a fixed-point function implementation. The same method as Chapter 8 can be used, modifying the possible output values Y_L, Y_H to fit the interval of possible seeds. Modifications to the computation of the guard bit and the external loop is required to be able to find better solutions.

Unfortunately, I did not have the time to complete this experimentation before submitting the manuscript.

9.4.3 Proposed Architecture

The seeds determined previously used 14 input bits, including the implicit bit. The implicit bit can be abstracted into the function, but the LSB of the exponent must be concatenated to determine on which interval the input is, keeping an input of 14 bits to the multipartite tables.

The architecture is depicted in Figure 9.3. The FP16 and FP32 inputs are first unpacked. Subnormal inputs can be supported by normalising the fractions, which is not represented in the figure.

The fraction F of the FP16 input is concatenated to the LSB e_0 of the unbiased exponent E_{in} and padded with zeros: $e_0 F 000_2$. The fraction $F = f_{-1} \dots f_{-23}$ of the FP32 input is truncated before being concatenated to e_0: $e_0 f_{-1} \dots f_{-13}$. This modified fraction is the input to the seed table previously described.

For FP16, the output of the seed table is rounded to fit the size of the output fraction, before being packed into an FP16. For FP16, the output of the seed table is padded with zeros before being packed.

The tentative exponent must be divided by two, change sign and have the correct: $E_{\text{tmp}} = b + \lfloor -\frac{E_{\text{in}}-b}{2} \rfloor$, where b is the bias, and is an odd number.

$E_{\text{in}} - e_0$ is always even since it is computed by replacing the LSB of E_{in} by 0, and $\frac{E_{\text{in}}-e_0}{2}$ consists of shifting E_{in} by 1 position.

$$
\begin{aligned}
E_{\text{tmp}} &= b + \lfloor -\frac{E_{\text{in}} - b}{2} \rfloor \\
&= b - \lceil \frac{E_{\text{in}} - b}{2} \rceil \\
&= b - \lceil \frac{E_{\text{in}} - e_0}{2} + \frac{e_0 - b}{2} \rceil \\
&= b - (E_{\text{in}} >> 1) + \lceil \frac{e_0 - b}{2} \rceil \\
&= b + \frac{b+1}{2} - e_0 - (E_{\text{in}} >> 1)
\end{aligned}
$$

For FP16, the exponent is computed as $23 - e_0 - (E_{\text{in}} >> 1)$, and for FP32 as $191 - e_0 - (E_{\text{in}} >> 1)$.

The fixed-point number $\frac{1}{\sqrt{1.F}}$ is equal to 1 when $F = 0$ and in $(0.5, 1)$ otherwise. If $F = 0$, then the output of the tables is normalised, and the exponent does not need to be adjusted $E_{\text{out}} = E_{\text{tmp}}$. Otherwise, the fraction must be normalised by one and $E_{\text{out}} = E_{\text{tmp}} - 1$.

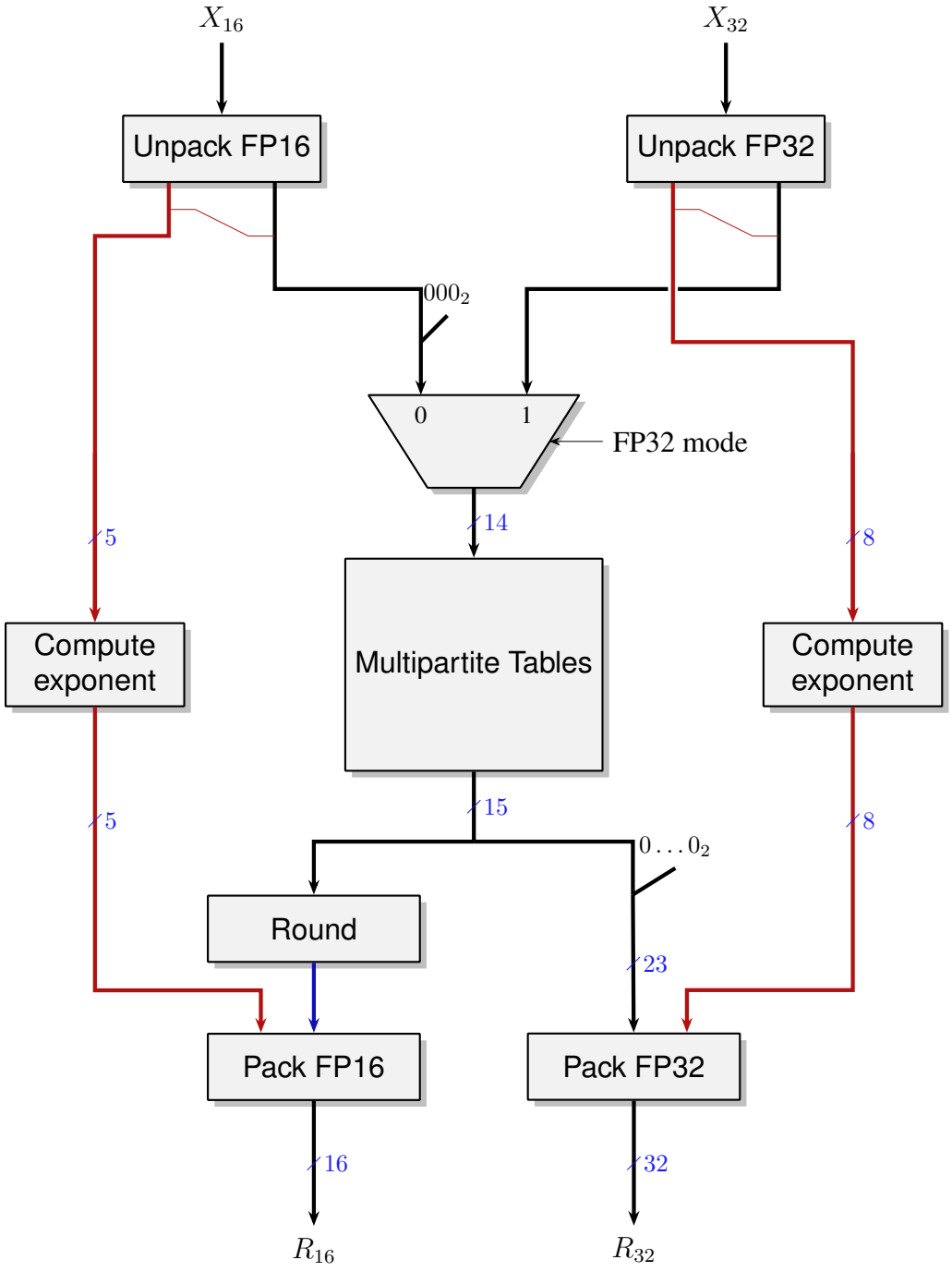


Figure 9.3: Proposed architecture for the seed operator. The red wires indicate the path taken by the exponents.

9.5 Conclusions

This chapter presents an architecture for functions of the square root family based on the Newton-Raphson method. It's main contribution is a seed table that directly provides correct rounding to $\frac{1}{\sqrt{a}}$ in FP16, or a seed to compute $\frac{1}{\sqrt{a}}, \sqrt{a}, \frac{1}{a}$ in one Newton-Raphson iteration in FP32. An extra corrective iteration provides a correctly rounded $\sqrt{a}, \frac{1}{\sqrt{a}}$ in FP32.

The mixing of hardware and software approximation methods can result in extra throughput in an accelerator containing a lot of parallel FMA operators, without incurring a very large hardware cost.

Future works include treating the special case of the last seed table, described in section 9.2.2. This case can easily be detected and corrected in hardware, where the hardcoded value can be returned instead of the seed, raising a flag. In this situation, a conditional move can choose either the result of the software iteration or the hardware operator, depending on the value of the flag.

The implementation of subnormals is relegated to future works.

A comparison of this operator with one using a 9-bit seed and one more Newton-Raphson iteration for each case should be carried out.

The software iterations would require more FMA operations, dissipating more heat and limiting the throughput. A fair comparison is difficult as it must be made on examples reflecting the distribution of FP16 and FP32 usages of the operator.

Chapter 10

Exponential Function: Mixed-Precision and Design Space Exploration

The exponential function $\exp$ is defined as the function that is its own derivative such that $\exp(0) = 1$. It is the only differential function f on $\mathbb{R}$ that verifies $\forall x, y \in \mathbb{R}, f(x+y) = f(x) \times f(y)$ and $f(1) = e$, where e is Euler's number ($e \approx 2.72$). It is noted:

$$\begin{aligned} \exp\colon \mathbb{R} &\to \mathbb{R}^{+*} \\ x &\mapsto e^x \end{aligned} \quad .$$

As described in earlier chapters, the exponential function is crucial to the functioning of AI models and transformers. Its main use is in the softmax σ function:

$$\sigma(\vec{x})_i = \frac{e^{x_{\max} - x_i}}{\sum_{j=0}^{n-1} e^{x_{\max} - x_j}}$$

It is also a useful elementary function for scientific computing, enables to compute the power function [102, 42, 38], or carry out Monte-Carlo simulations [43, 73].

10.1 Implementing the exponential

10.1.1 Software implementations

When the first IEEE 754 standard [66] was published in 1985, a guideline of instructions on how to implement the exponential function in hardware and with IEEE 754 compliant arithmetic was published by Tang [115]. The paper presents both an algorithm and the error analysis, and considers the various contributions to the error: reduction, approximation and rounding.

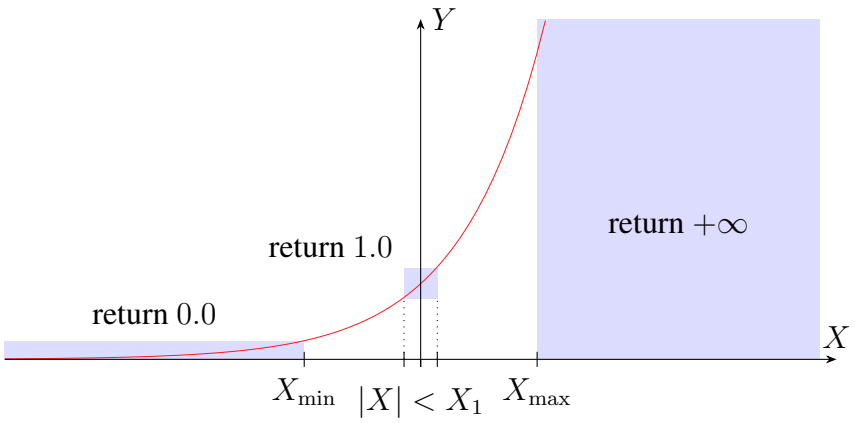


Figure 10.1: Special cases in the exponential function (Figure from [34]).

Special cases:

The first step in evaluating the exponential is to deal with special cases. As seen in Fig. 10.1, the exponential returns a special value for most of the range of the input. If the input $X > X_{\text{max}}$, then $e^X = +\inf$, if $X < X_{\text{min}}$, $e^X = +0.0$, and if $|X| < X_1$ then $e^X = 1.0$. For an IEEE 754 [68] floating-point number of format $\mathbb{F}(w_E, w_F)$ (see section 2.4), the thresholds are computed with the following formulas:

$$X_{\text{min}} = -\lfloor \log(2^{-2^{w_E-1}+2-w_F}) \rfloor$$
$$X_{\text{max}} = \lceil \log((2 - 2^{-w_F}) \times 2^{2^{w_E-1}-1}) \rceil$$
$$X_1 = 2^{-w_F-2}$$

Reduce the input:

The reduction step follow the idea that X can be reduced to E, integer and $Y \in \left[-\frac{\log(2)}{2}, \frac{\log(2)}{2}\right]$, such that:

$$X = E \times \log(2) + Y \Rightarrow e^X = 2^E \times e^Y$$

This formulation is such that E can be the exponent of the result, and e^Y the significand, albeit with some corrections. This is due to the fact that $e^Y \in [0.7, 1.42]$,

and might require to be shifted by one, causing the exponent E to be off by 1. In Tang's paper, E is called j.

Y is separated into $Y = A + Z$ such that $Z < 2^{-k}$, and A is a multiple of 2^{-k}. This operation is cheap, and can be done with shifts and masks. A formula based on a first order Taylor approximation is used: $e^Y \approx e^A + e^A \times (e^Z - 1)$. In this case, e^A is tabulated, and $e^Z - 1$ can be computed with a polynomial approximation. In Tang's paper, A is called j and $k = 5$ is used.

For more precise formats, a second order Taylor reduction can be used: $e^Y \approx e^A + e^A \times (Z + (e^Z - Z - 1))$.

Approximate the rest of the exponential:

The formula $e^Z - 1$ is approximated using a polynomial function. In Tang's paper, Z is represented with a double-word representation: $Z = R_1 + R_2$, where R_1, R_2 are floating-point numbers matching the target precision. A polynomial function of degree 6 is used for double precision, but this can change depending on the value of k. If a second order reduction were used, a polynomial function with a smaller degree can be enough.

Reconstruct the exponential:

The exponential is reconstructed according to the previous formulas: $e^X = 2^E \times (e^A + e^A \times (e^Z - 1))$ or $e^X = 2^E \times (e^A + e^A \times (Z + (e^Z - Z - 1)))$.

Progress made since

The article performs an error analysis proving that the algorithms computes a faithfully rounded exponential. The correctness was also proven formally in HOL [57], and later in Coq using the same method [20].

Advances in polynomial approximation [12] and evaluation [85, 16] have enabled better exponential implementations.

Some correctly rounded versions have been designed since, based on detecting cases where the correct rounding had issues (near a midpoint), and recomputing the value with more precision [127]. The algorithm was much slower when recomputing, but this did not hurt too much the average execution time as those cases were uncommon.

By studying those complicated cases [84, 33], it was determined that the most significant bits needed to compute the correct rounding in double precision was 115 bits of precision on the polynomial approximation.

Similar implementations are used in the CORE-MATH [108] state of the art open-source library for correctly rounded elementary functions.

10.1.2 Hardware exponential

The algorithm described in the previous section is the basis for the implementation of the floating point exponential in hardware. The main difference is how step 3 is implemented, as a polynomial approximation is not the only option for the evaluation

of the reduced exponential function. This will induce variation in the reduction and reconstruction steps, but the idea of the algorithm is very similar.

In hardware, it is necessary to always compute the different branches of an *if* statement. This makes the correct rounding of the function always costly, instead of only being slow for some inputs. While it could be desirable to implement correct rounding, the target accuracy for this hardware exponential is faithful rounding.

Another specificity of the hardware implementation of the exponential function is that it is possible to reduce the floating-point input to a fixed-point number. This would not be practical in software as the fixed-point format needed is not of standard format.

A different option [31] for the polynomial approximation step is to further reduce the input in order to need to approximate a smaller interval of the exponential function. This can be done recursively until the interval is small enough to tabulate. This method requires the use of many rectangular multipliers, which are cheap for an FPGA that does not have DSPs.

An iterative algorithm can also be used for an online high-radix implementation [45, 101, 102, 122]. Muller's book [94] presents those algorithms and more, like some CORDIC and iterative fixed-point algorithms on a small domain for the computation of the exponential.

Exponential in FloPoCo:

The implementation in FloPoCo is described in detail in [36, 34], and illustrated by Fig. 10.2.

The input is first converted into a fixed-point number. It's MSB and LSB are computed depending on the format of the output, using the values of $X_{\text{min}}, X_{\text{max}}, X_1$ (see Fig. 10.1). The implementation uses custom FloPoCo floating-point numbers NFloat, that do not have subnormals and do not encode infinities or NaN using the largest exponent binade. This induces a difference in the formula for the computation of the thresholds.

$$X_{\text{min}} = -\lfloor \log(2^{-2^{w_E-1}+1}) \rfloor$$

$$X_{\text{max}} = \lceil \log((2 - 2^{-w_F}) \times 2^{2^{w_E-1}}) \rceil$$

$$X_1 = 2^{-w_F-2}$$

Multiple methods can be used to approximate e^Y, depending on the precision needed.

If the target precision is small enough such that e^Y can fit into the block RAM of the target FPGA, then it is tabulated. Otherwise, a first or second order reduction can be used, similar to the software implementation. FloPoCo uses a first order of reduction for half precision, and a second order for float and double. The approximation of the function $e^Z - 1$ or $e^Z - Z - 1$ can either be tabulated if it is small enough, or a polynomial evaluation with a Horner scheme is used.

In accordance to FloPoCo's motto, "Computing Just Right", every bus and truncated multiplier are the exact right size to not avoid superfluous computations

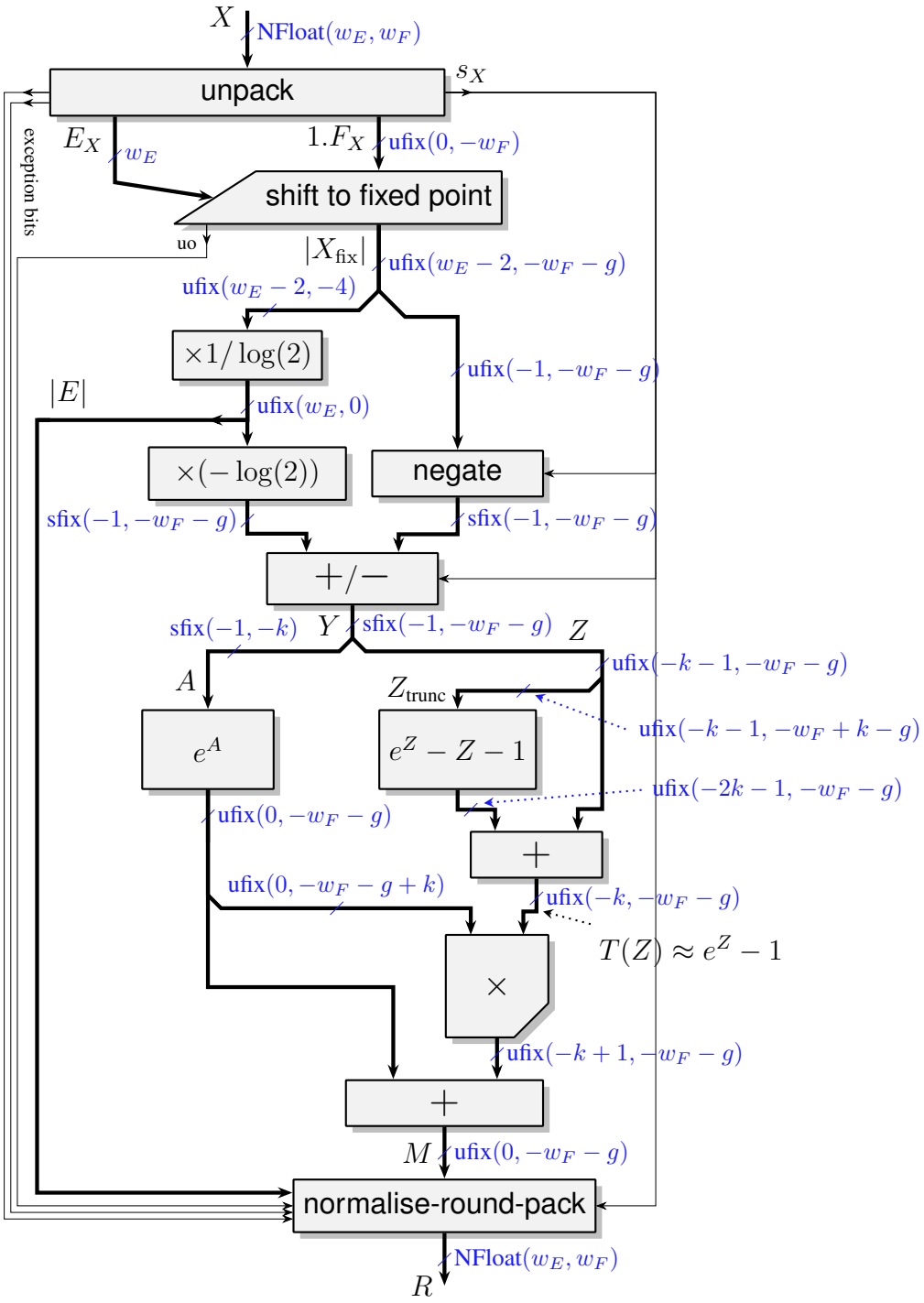


Figure 10.2: Architecture of FPExp with a second order reduction (Figure from [34]).

and still meet the accuracy target of faithful rounding. The computations for this are detailed in [34].

The framework also provides a test bench generator to check the operator, based on the Multiple Precision Floating-Point Reliable Library (MPFR) [50] golden model.

10.2 Implementation in FloPoCo of the exponential in other number formats

10.2.1 Fixed-point in, floating-point out

The exponential in FloPoCo would only support the NFloat format, and was contained all as a single operator in one file FPExp. The goal was to be able to compute the exponential on the IEEE floating-point format, without duplicating code.

The first step was to separate the NFloat exponential into 3 parts (Fig. 10.3):

- All the automatic determination of architecture parameters are moved into a HighLevelArithmetic file ExpArchitecture. This includes the MSB and LSB of the fixed-point, k the size of A, d the degree of the approximation and g the number of guard bits needed on the fixed-point to assure enough precision is used.

- The core of the exponential Exp is isolated: fixed-point in and pseudo floating-point out. All the internal and interface parameters are already computed by ExpArchitecture. Exp takes as an input an unsigned fixed-point number, computes its exponential as described earlier, and returns an exponent and a fixed-point number, without the corrections necessary for it to be a proper NFloat.

- A wrapper FPExp is created. It converts the floating-point input into the required fixed-point for Exp, and corrects the output so that it is a normalised NFloat.

10.2.2 Support of IEEE floating-point numbers

Some modifications were required on ExpArchitecture to support the fact that subnormal output is possible with IEEE floating-point numbers. This changes the computation of X_{min}, which in turn modifies the MSB of the fixed-point format.

The wrapper IEEEFPExp (Fig. 10.4 for the IEEE floating-point numbers is similar to FPExp. The subnormal inputs do not cause any complications as they are smaller than the X_1 threshold.

However, modifications are needed to support the subnormal outputs. A shifter is used to denormalise the significand of the result when the exponent is smaller than the minimum exponent of the format. This shifter also performs the correction shift mentioned for NFloats.

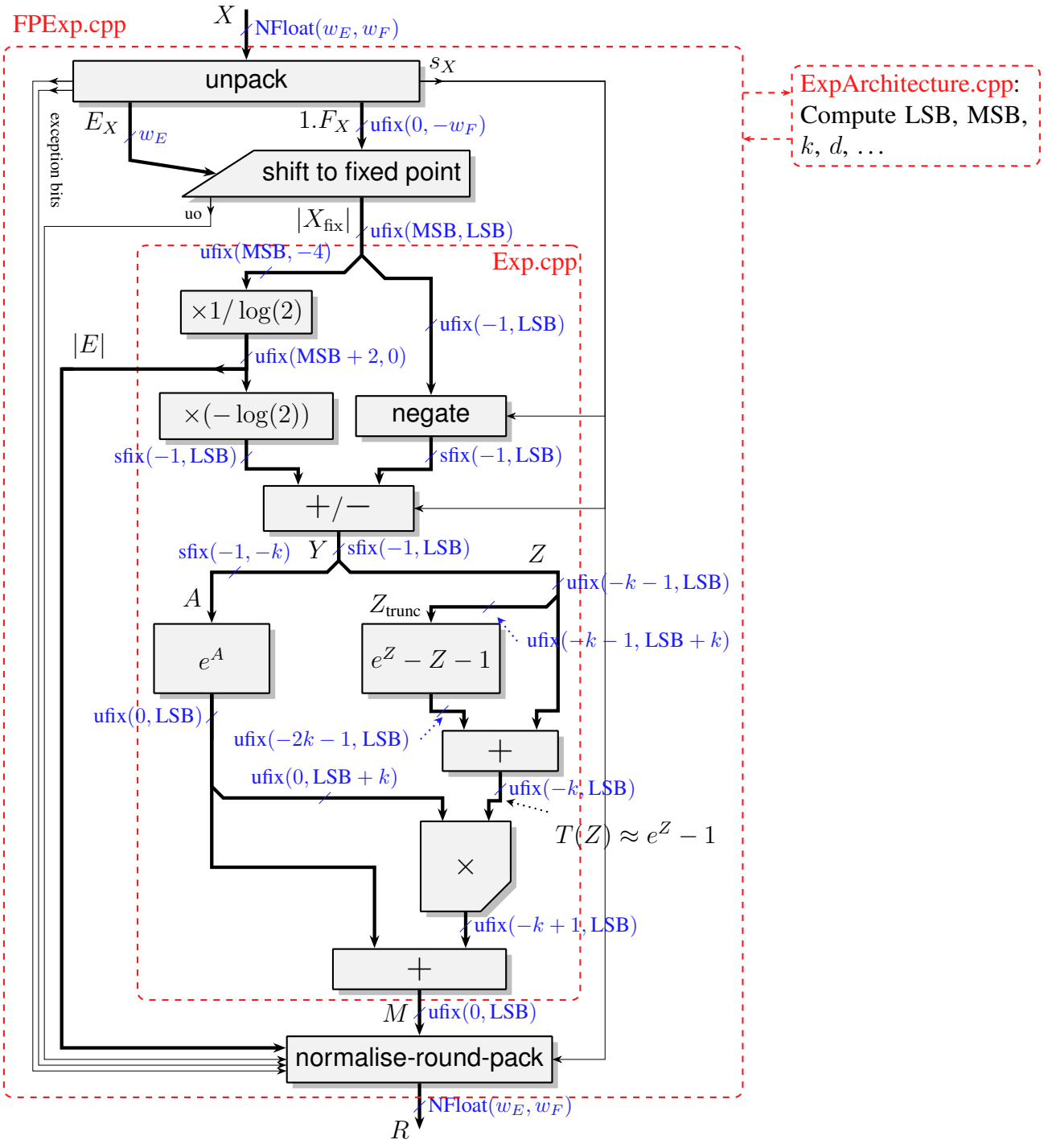


Figure 10.3: Separation of the exponential into 3 parts. Figure adapted from [34].

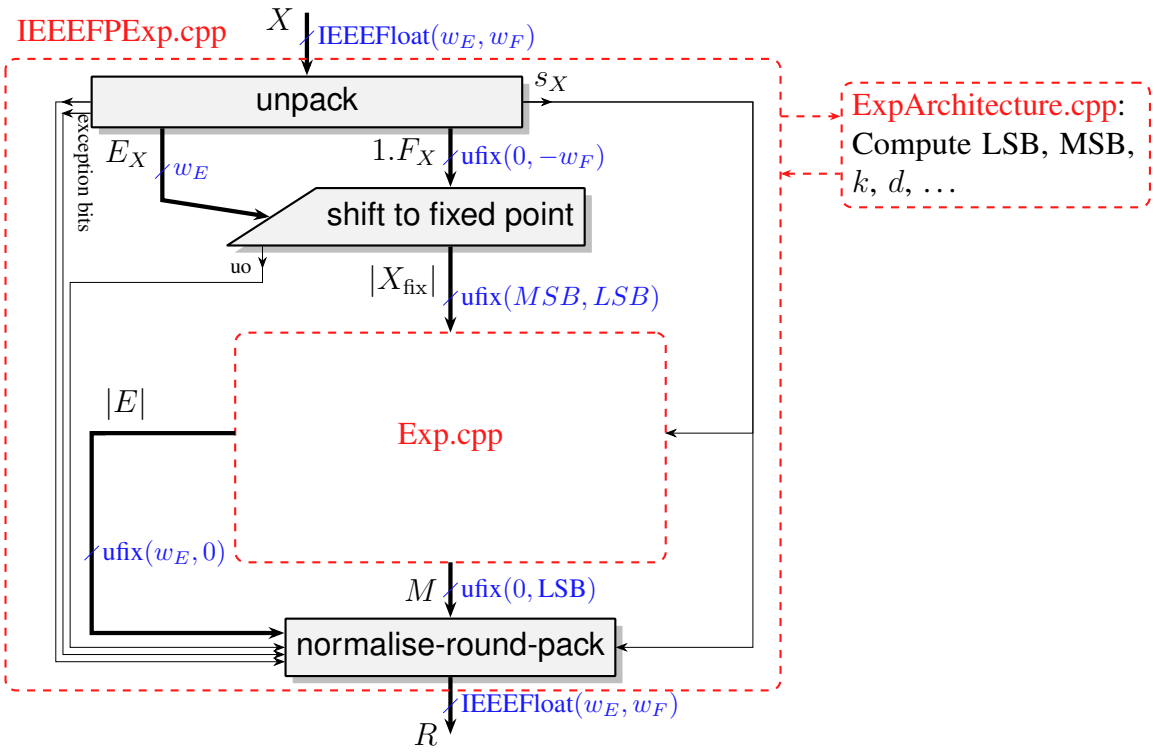


Figure 10.4: Add FloPoCo support of IEEE floating-point format. Figure adapted from [34].

Once the significand is shifted to the correct placement, it is rounded, and the exponent is updated. It is then packed into the IEEE floating-point format.

It is easier to compute correctly the IEEE flags than the correct rounding for the exponential. If it were necessary to generate them, this would be the implementation for the flag computation:

- Inexact: always raised, except when the input is ± 0, NaN or $-\infty$, since floating-point numbers are rational, and the exponential of a non-zero rational is always irrational [94].
- Invalid operation: raised if the input is a signalling NaN
- Division by zero: never raised
- Overflow: raised when the result is $+\infty$
- Underflow: raised when the result is subnormal or zero, except when the input was $-\infty$

10.2.3 Mixed-Precision input

The Kalray accelerator required the use of a FP32 exponential. For machine learning applications, the main use is softmax (see chap. 3). There, the input format FP16 could also be useful, but it was not crucial for the output format to also be FP16. Since the exponential is defined by its output format, it is possible to add additional input formats as long as they can be converted to the fixed-point input to `Exp`.

A production operator was specified as needing to accept the following input formats: FP32, BF16, FP16, FP8-E5M2 and FP8-E4M3. This was implemented as another wrapper for `Exp`, heavily inspired by `IEEEFPExp` as it would use the same IEEE floating-point output logic. The various input formats were managed ahead of the first shifter. This was implemented as a few muxes to extract the correct significand, and constant adders to compute the correct shift value. The impact on the design area was minimal.

10.3 Exploration of parameters for VLSI

The exponential in FloPoCo is optimised to be the best for FPGA. However, VLSI offer different trade-offs, and especially is less efficient for the implementation of tables.

This operator was synthesised on the TSMC 4nm node with Synopsys Design Compiler.

The first test (Fig. 10.5) was to vary the parameter k that decides of the size of A, the input to the table storing e^A. For single precision, FloPoCo uses by default a reduction of the second order, in which the function $f(Z) = e^Z - Z - 1$ is evaluated with a table ($d = 0$).

The size of the table containing e^A grows exponentially with k. If k is too big, then the table containing e^A is large. There is still a trade-off, as when k is too small, then the table containing $e^Z - Z - 1$ starts to become large.

The default value used in FloPoCo is $k = 9$ for single precision, as it was the best value when exploring the design space for FPGAs (see [36]). This also seems to be the best value when synthesising for VLSI. This figure also shows that the majority of the area of the operator is occupied by tables.

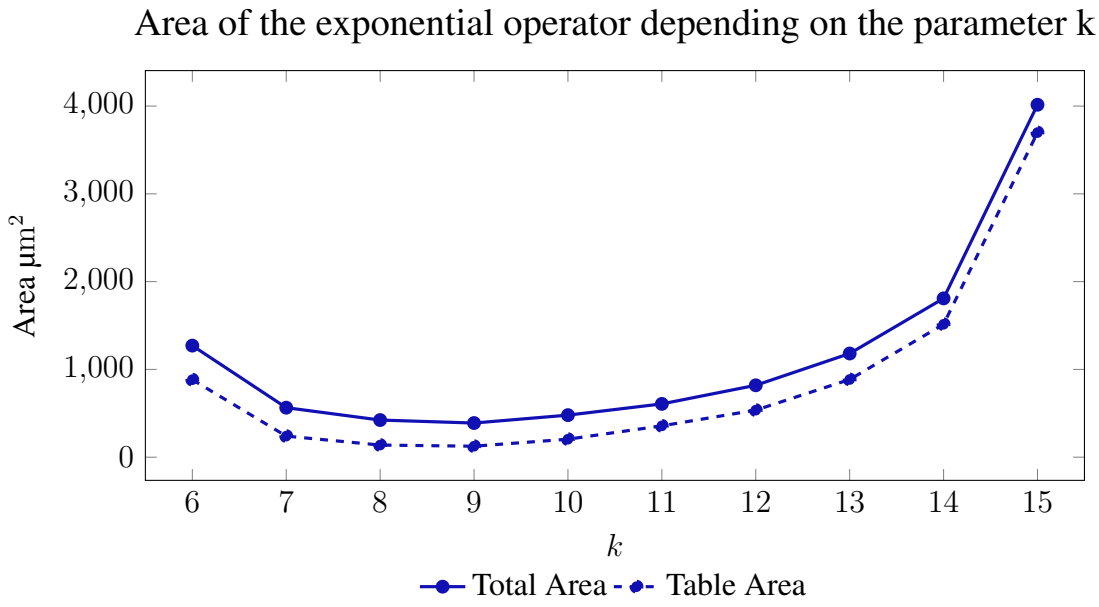


Figure 10.5: Parameter exploration of the exponential architecture, with a reduction of the second order, for varying k and $f(Z) = e^Z - Z - 1$ approximated by a table ($d = 0$).

Since it is likely the operator can be reduced in size by reducing the number of tables, it is natural to also explore higher order of approximation for $f(Z) = e^Z - Z - 1$ (Fig. 10.6). When k is large, the input to the function evaluator is small, and the polynomial approximation does not need too high degrees to work. This is why there is no result for $d = 2$ when $k > 7$, and for $d = 1$ when $k > 11$.

It is interesting to see that for $k \geq 9$, the size of the table evaluating e^A is so large that it overtakes the efficiency of the approximation of Z. In that case, the methods for $d = 0$ and $d = 1$ give very similar results.

This exploration shows that the best architecture of a single precision exponential for VLSI is for $k = 8, d = 1$, while for the FPGA it was $k = 9, d = 0$ as explored in [36]. This is not surprising as VLSI has more available optimisations for multipliers, and is worse at implementing tables than FPGA.

10.4 Conclusions

The IEEE-754 standard [68] does not require that the exponential function be present for a CPU to be IEEE compliant, but encourages its implementation to be correctly rounded. The proposed exponential is not IEEE-754 compliant, but merely accepts IEEE floating-point numbers as inputs and outputs. It does not generate IEEE flags,

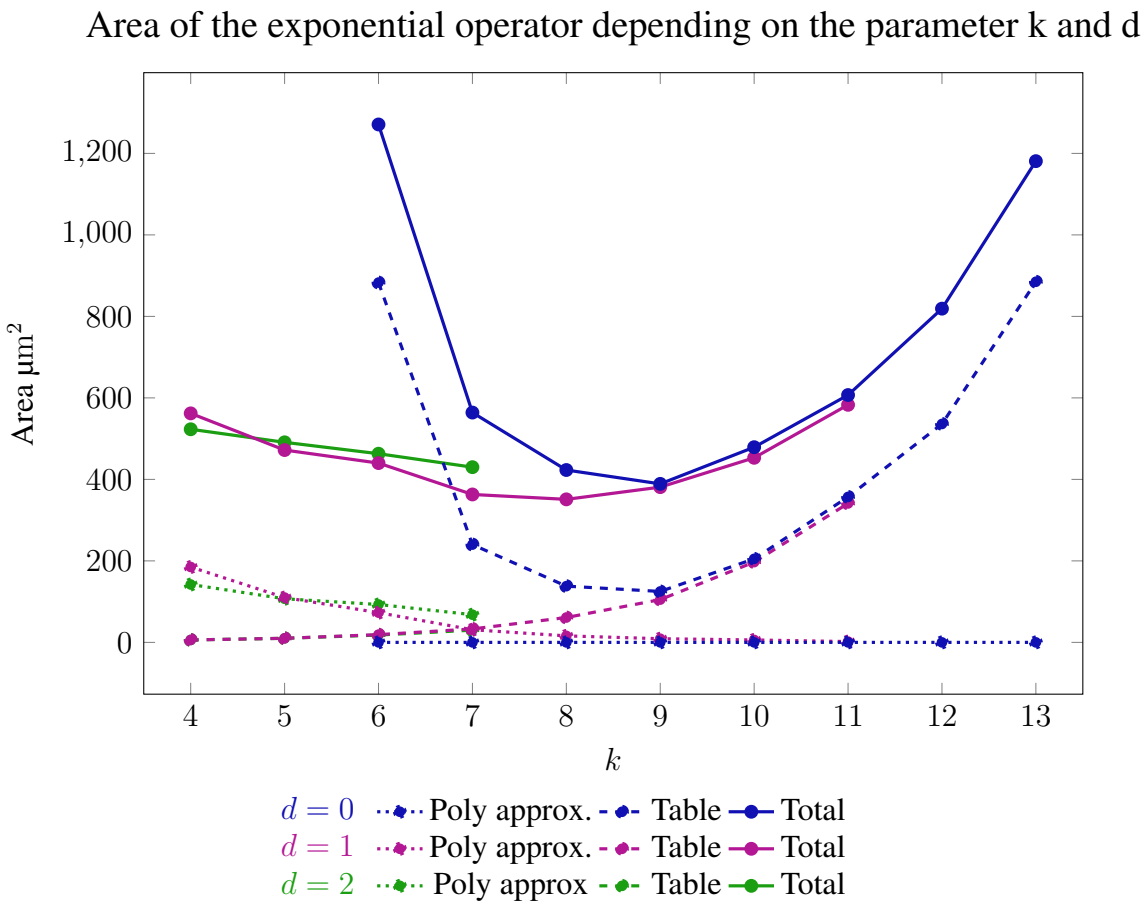


Figure 10.6: Parameter exploration of the exponential architecture, with a reduction of the second order, for varying k and $f(Z) = e^Z - Z - 1$ approximated with an approximation of varying degree d.

and is faithfully rounded. This exponential function is implemented in the vector acceleration unit of Kalray's MPPA, which does not enable many IEEE flags since it is difficult to be sure which element of the vector raised the flag. Moreover, 8 or 16 instances of the exponential will be included in each Processing Element, that is either 640 or 1280 instances in a whole MPPA. A lighter implementation is preferred, and faithful rounding is still an acceptable precision for most applications.

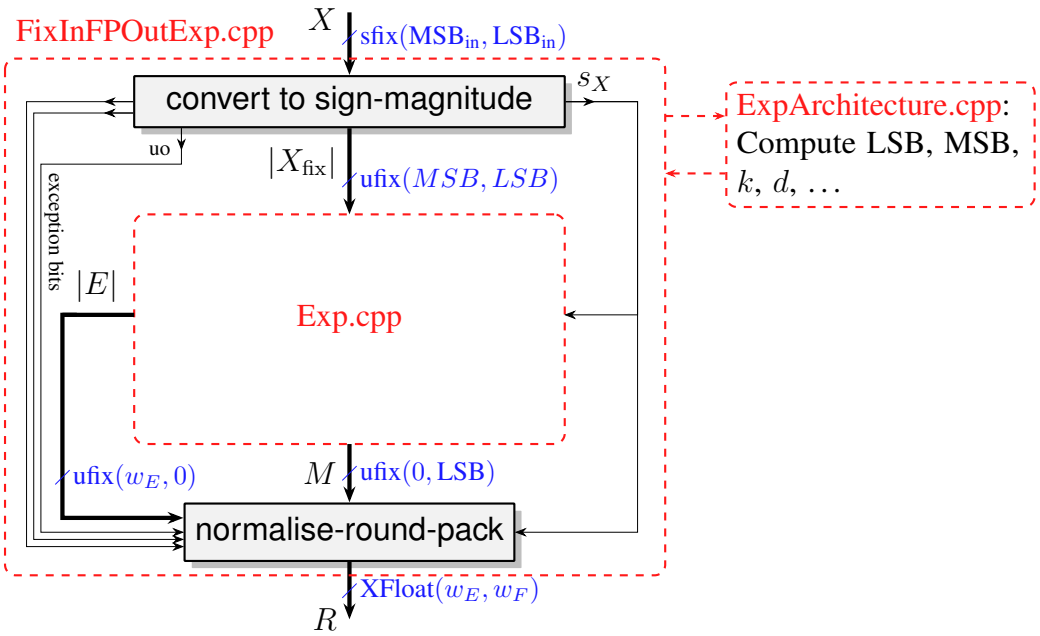


Figure 10.7: What FloPoCo support of fixed-point in, floating-point out could look like. The floating-point format in output could either be NFloat or IEEEFloat, it would just change the computation of MSB. Figure adapted from [34].

In future works, it would be interesting to implement a fourth wrapper for the exponential to compute a generic fixed-point in, floating-point out (Fig. 10.7). This would require careful handling of the sign of the fixed-point number.

In machine learning, the output of a matrix multiplication can be a large fixed-point accumulator, and a softmax computation is performed on the outputs. It could be useful to compute the exponential of the accumulator directly, without needing to convert it in floating-point and then fixed-point again. This would be a very interesting combination, however it is impractical in the Kalray system as the fixed-point accumulator of the matrix-multiplication exists as a internal format in the matrix unit, and the exponential in the vector unit.

Conclusion and Future Works

In the era where the impending limits of *Moore's Law* meet the AI boom, the importance of efficient hardware design is higher than ever. Pressures to design cheaper operators that compute more FLOPs have often resulted in designs sacrificing accuracy, sometimes even without significant area improvement. This thesis specifically focused on designing operators for matrix multiplication, broken down into dot products, and the implementation of functions.

This thesis proposes three different architectures for dot product operators, comparing them depending on various parameters: precision of the formats used, size of the dot product, and what accuracy the applications required.

For machine learning, specifically deep neural networks, the dot product operate on very small number formats and can be optimised as such, often using the Full-precision Fixed point method of adding multiple floating point. This is a variant of the classic Kulisch accumulator [78]. Synthesis allows comparison of the implementation cost of various small floating-point formats used in machine learning: two FP8 variants, INT8, four Posit8 variants and FP16, showing promising hardware cost for FP8.

A Full-precision architecture for FP16 can be modified to provide moderately accurate dot-product computations for intermediate number formats. Instead of a Full-precision fixed-point accumulation method, a Truncated floating-point accumulation is used to support formats with larger ranges without increasing the accumulation size. This enables BF16 support without increasing the cost of the multiplier and FP32 support can be added using double-word techniques common in software.

This thesis also presents a correctly rounded dot product architecture able to operate on the larger number formats used for scientific computing. It compares with two state-of-the-art method, the Full-precision Fixed point method, and Tao's method [116], expanding Tao's architectures with subnormal support and optimising subcomponents like the alignment computation. For a dot product and add of size 16 on FP64 numbers, comparison shows an improvement of 45% compared to both state-of-the-art methods.

In the area of function approximation, this thesis proposes iterative square root algorithms with partial hardware acceleration, allowing versatility in the operators ($\frac{1}{\sqrt{x}}$, $\sqrt{x}$, $\frac{1}{x}$), the supported formats and the accuracy. The software refinement steps enable to evaluate trade-offs when coding the application, allowing to optimise for various needs.

Exploration of the parameter space for state-of-the-art exponential architectures shows that the table-to-multiplier trade-off for VLSI is a bit different than for FPGAs,

but still closer than one might think, given that the basic gate for FPGAs is the Look Up Table.

Future Works

Studying the Impact of Truncated Floating-Point Accumulators on Computations

Most GPUs implement their dot products using a truncated floating-point accumulator described in Chapter 6. This architecture computes an inaccurate result, which works fine for machine learning applications but is problematic to use for scientific computing applications. It it is hard to predict, limit and control the errors made by the architecture [47, 46]. Studies have for instance revealed monotonicity issues in the operator [92]. Efficient usage of those operators often stems from a in-depth knowledge of their architecture.

Small Precision Dot Products with Scaling

In machine learning models, working with quantised 8-bit floating-point numbers requires scaling steps. Some formats specify [104] a version of the FP8 formats that also include a shared 8-bit external exponent. This exponent E_{ext} is encoded exactly like FP32 exponents are. It is shared for a vector of FP8 numbers $\vec{X} = (X_i)_{0 \leq i < N}$, where X_i are FP8 numbers used to represent the value $x_i \times 2^{E_{\text{ext}}}$ that have a much larger range. LLMs perform best when the vector is small, that is not too many numbers share the same exponents, with the usual size N of the vector being 4 or 8.

Since an operator typically inputs one shared exponent $E_{\text{X,ext}}$ for all the inputs (X_i), and another $E_{\text{Y, ext}}$ for all the inputs (Y_i) have, the size of the input vectors is limited to 4 or 8.

Some recent architectures [88] include this scaling in the computation of the dot product, and achieve a correctly rounded result. The scaling is applied once, when adding to the FP32 addend Z, because the scaling can be factored out of the sum of products:

$$R = Z + \sum_{i=0}^{n-1} (x_i \times 2^{E_{\text{X, ext}}}) \times (y_i \times 2^{E_{\text{Y, ext}}}) = Z + 2^{E_{\text{X,ext}} + E_{\text{Y,ext}}} \times \sum_{i=0}^{n-1} x_i \times y_i \quad .$$

This architecture is an intermediate step in terms of precision, it cannot accumulate exactly as many products as the full precision fixed-point architecture from Chapter 5, but does not require as much expensive alignment logic as the truncated floating-point architecture from Chapter 6 since the FP8 sum of product is computed with a full precision fixed-point accumulator, and is only later seen as a floating-point number. The output format, a FP32 floating-point number, is also much easier to manipulate than the larger internal formats implemented in those chapters, although it sacrifices precision.

This architecture deserves to be added to the cost comparison carried out in section 6.7. The architecture from [88] was compared in that paper to the one

presented in [29] (which Chapter 5 expands upon), and future works could also compare it to the truncated floating-point method commonly used in GPUs.

Adding scaling to Kalray's accelerator is a challenge for many reasons. The FP8 dot product operators in Kalray's accelerator have large input vectors ($N = 32$). A question is: is 32 still fine-grained enough for LLMs ? The micro-scaling format is also very specific to vector operations. Another challenge is to define to define scaling formats for matrix operations. Addressing these problems is necessary to run inference for models that were trained on other platforms.

Proving the Correctly Rounded Dot Products on Large Precision

The large precision dot product operators presented in Chapter 7 has been extensively tested, but they would benefit from being formally proven to round correctly.

Those operators would be used [65] to perform Error Free Transforms and improve the error bounds of some operations in double word. Using an unproven operator in a proven algorithm is not very satisfying.

Correctly Rounded Multipartite Tables without ILPs

Chapter 8 explains the use of Integer Linear Programming (ILP) to obtain correct rounding in multipartite architectures. The Lossless Differential Table Compression (LDTC) method can also be used to obtain correct rounding without needing to use optimisation models. Some background work during this thesis was dedicated to trying to extend the LDTC method of filling tables to replace the ILP for correctly rounded multipartite tables, without much success due to the little amount of time I was able to dedicate to it. At first sight, the multipartite problem is not NP-Complete and using ILP methods to solve may not be necessary.

The modifications carried out until now managed to remove the optimisation of the size of the table (MSB and g for each table) outside of the ILP model, and into the enumeration of the architecture. This reduces the ILP model to just a satisfiability problem without making the overall method take more time, nor less.

Being able to fill the multipartite tables without an ILP model should speed up the method and hopefully scale to larger precisions.

Publications and patents

This thesis lead to several publications and patents:

- Chapter 5: This work was presented as a conference paper at the Digital Systems Design (DSD) conference in 2023 [29].
- Chapter 6: This work was presented as a conference paper at the IEEE International Symposium on Computer Arithmetic (ARITH) in 2025 [27]. It also resulted in a patent [26].
- Chapter 7: This work was presented as a conference paper at the IEEE International Symposium on Computer Arithmetic (ARITH) in 2023 [28]. It also resulted in a patent [25]. Use of this work as well as some minor extra

contributions was made in a conference paper presented at High Performance Extreme Computing (HPEC) conference in 2023 [40].

- Chapter 8: This chapter describes contributions from before my thesis, presented in the International Conference on Field-Programmable Technology (ICFPT) in 2022 [24], as well as some minor contributions to the paper [64] that is not yet published.
- Chapter 9 is yet to be submitted to a conference.

Appendix A

Technical Contributions to the FloPoCo framework

Si vis pacem, pars à vélo.

– F.

This thesis has lead to multiple contributions to the FloPoCo framework. While most of the operators coded were not open-sourced by Kalray, some subcomponents were published in the FloPoCo repository. In particular, two hardware sorts evaluated in the context of the operator from Chapter 7 were added, as well as an exponential that supports IEEE numbers (in particular subnormals) from Chapter 10.

Other open-source contributions lie in modifications to the FloPoCo framework generating the operator, enabling the use of FloPoCo to generate VHDL for VLSI instead of only FPGA. This includes the creation of a new FloPoCo target: `ManualPipeline`.

A.1 Pipelining in FloPoCo

A.1.1 Automatic Pipelining

Operators in FloPoCo are written such that each operation is paired to the delay it takes to carry it out once all the inputs are ready. Since the actual delay depends on the FPGA target, generic helper functions are used to abstract the individual values:

- `logicDelay(i)` : The delay of a simple logic computation with i bits of input (using Look Up Tables LUTs).
- `adderDelay(i)` : The delay of adding two integers that are of width i, using LUTs.
- `eqComparisonDelay(i)` : The delay of testing if two vectors of width i are equal.
- `ltComparisonDelay(i)` : The delay of testing which of two integers of width i is the biggest.

Different FPGA targets are described in FloPoCo, supplying the information for different parameters, like the number of inputs to each LUT, the delay of a LUT gate, the size of the BRAMs,...

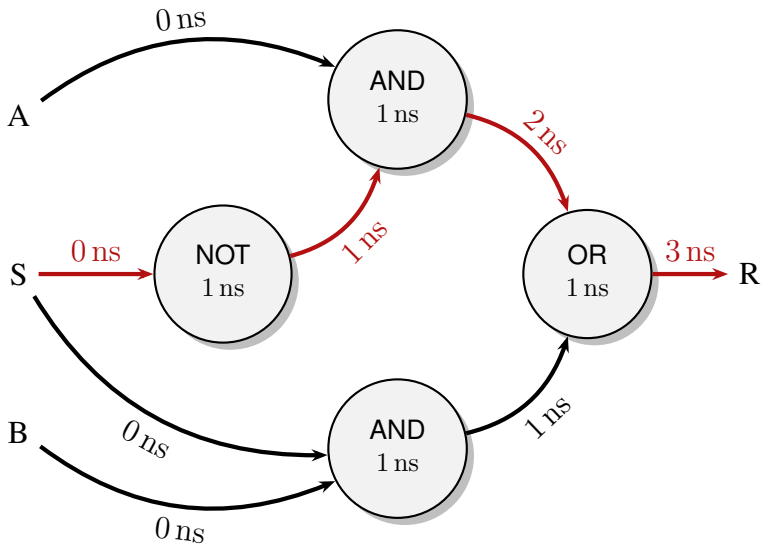


Figure A.1: Example of a operator graph for a MUX, with dummy operation times.

The FloPoCo scheduler [72] labels dependencies between signals with the time needed execute each computation. Each computation can be seen as a node in a directed acyclic graph, where the operands are the input edges, and the result the output edge of each node. One can compute at which time an output signal is ready by adding the delay of the computation creating it to the input which is ready the latest (Fig. A.1). An input signal is considered ready at time 0, and the value of the output signal enables to compute the total latency of the operator. The longest path, in red in the figure, is called the critical path. This graph can be generated by FloPoCo using the `dependencyGraph` command line option.

When a path becomes too long to fit in the timing budget set by the target frequency, a pipeline stage is added so that the signal is computed at the next cycle. The scheduling method used is As Soon As Possible, this means that as soon as the path is too long, a pipeline register is added.

When a signal `sig` is pipelined, different copies of this signal are created for their use in subsequent cycles : `sig_d1` (signal delayed by 1 cycle), `sig_d2`,

`sig_d3`, etc... The scheduler keeps track of the copies of the signals, to make sure that computations use synchronised data. Pipelining in FloPoCo is correct by construction: if a circuit works unpipelined, it works pipelined.

A.1.2 Manual Pipelining for VLSI

In FPGAs, not many different basic bricks are available, making automatic pipelining a great option. However, doing the same for VLSI would be time consuming, as all the different standard cells can be used with different options depending on the drive and how fast they should be.

In addition, while pipelining is free for FPGAs as there is a register after each LUT, this is not the case for VLSI. It might even be desirable to use expensive fast gates to cram a large number of operations in a pipeline stage just to reduce the number of registers used. This is for example the case in the matrix multiplication operators, where pipelining between the large shift and the addition tree would be very expensive. Reducing the number of registers reduces the power consumption of the final operator.

All those possibilities make it very hard to automatically pipeline for VLSI the same way as for FPGAs. It is more realistic to pipeline manually and let the synthesis figure out if the pipeline holds. However, it is still interesting to use the power of the FloPoCo scheduler, as the pipeline is correct by construction.

A new target is created to this effect, called `ManualPipeline`. It sets to 0 all the previously described delays, and introduces a new function: `cycleDelay(i)` that describes a signal taking i cycles to compute. This forces the scheduler to pipeline just after a `cycleDelay` was used, and still profits from the correct by construction pipelining of FloPoCo.

The issue with this solution arises when subcomponents need to be pipelined. This was mostly the case for Shifters, as multiple types are needed at different stages of a computation. The solution used was to avoid pipelining shifters, however that could become necessary to meet latency demands. Possible future works would be adding a new shifter type, or a hidden parameter to the existing shifter, that enables to manually pipeline at given shift stages.

A.2 Various Modifications to Code Generation

A.2.1 Signal Renaming

The default naming of signals is to write of how many cycles this signal is delayed: if signal `sig` was defined in the second cycle of an operator, then `sig_d2` is a copy of that signal at the fourth cycle.

The new command line option `NameSignalByCycle` changes this to rename all the signals by their cycle instead of a delay. The previous example `sig` would be renamed during scheduling as `sig_c1` for its first occurrence when defined in the second cycle (as cycle numbers start at 0), and `sig_c3` for its copy at the fourth cycle.

This was necessary to better interface with synthesis scripts and naming conventions at Kalray.

It later turned out to be also very useful in the optimisation of the pipeline of legacy operators targeting FPGAs. For instance, one cycle was saved in the floating-point adder. This absolute cycle information is now also used by default in the standard output of FloPoCo:

```
./flopoco FPAdd we=8 wf=23 frequency=400
*** Final report ***
Output file:  flopoco.vhdl
Pipeline constructed using approximate timings for target
DummyFPGA @ 400 MHz
|--Entity RightShifterSticky24_by_max_26_Freq400_uid4
| R: (c1, 1.630000ns) Sticky:  (c2, 1.620000ns)
|--Entity IntAdder_27_Freq400_uid6
| R: (c3, 1.140000ns)
|--Entity Normalizer_Z_28_28_28_Freq400_uid8
| Count:  (c5, 1.580000ns) R: (c5, 2.130000ns)
|--Entity IntAdder_34_Freq400_uid11
| R: (c6, 1.710000ns)
Entity FPAdd_8_23_Freq400_uid2
R: (c7, 0.510000ns)
```

A.2.2 Write Enable

Another contribution was a rehaul of the `WriteEnable` option. Previously, this was added to pause the operator by blocking all the pipeline stages at once.

The new write enable adds one signal per pipeline stage, enabling to pause all the stages independently. The previous functionality stil exists as a toggleable option.

When an operator finishes a last computation, this technique helps save dynamic power keeping the transistors in the same state until they are used again, effectively flushing the pipeline with minimal power consumption.

A.2.3 Staggered Inputs and Outputs, and their Test Bench

In computation units, it is often desirable to have some late inputs (in terms of cycles), as well as have some early outputs. FloPoCo could already create early outputs, but this would break the test bench operator, that assumed all the outputs came out at the same cycle.

An optional delay was added to the input declaration to be able to make them arrive a set amount of time late: for example `cycleDelay(2)` to arrive on cycle 2, but any of the other delay functions can be used.

The test bench operator was then modified to accommodate this feature. If an input is late, then the test bench pipelines it before feeding it in the operator. If an output is early, it is pipelined before it is compared to the expected result. While FloPoCo's pipeline is correct by construction, the framework also urges us to check everything ourselves, and the test bench operator is central in this check.

Many thanks to Florent de Dinechin, Martin Kumm, and Pierre Cochard who were the main FloPoCo maintainers during my thesis, as well as all past, present and future contributors. FloPoCo greatly simplified VHDL code generation for all the operators designed during my thesis. The flexibility in the pipeline generation has saved me a lot of time when going back and forth with the Synthesis, trying to make my operators fit in the target number of cycles.

Bibliography

[1] Heron of Alexandria. *Metrica*. 1rst Century AD (cit. on p. 156).

[2] *Arm®A64 Instruction Set Architecture Armv9, for Armv9-A architecture profile*. Nov. 2021 (cit. on p. 45).

[3] Sherenaz W. Al-Haj Baddar and Kenneth E. Batcher. *Designing sorting networks: A new paradigm*. 2012 (cit. on p. 129).

[4] Luca Bertaccini et al. "MiniFloat-NN and ExSdotp: An ISA Extension and a Modular Open Hardware Unit for Low-Precision Training on RISC-V cores". In: *29th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2022, pp. 1–8 (cit. on pp. 116, 117).

[5] Sylvie Boldo and Guillaume Melquiond. "When double rounding is odd". In: *17th IMACS World Congress*. 2005 (cit. on p. 38).

[6] Andrew D Booth. "A signed binary multiplication technique". In: *The Quarterly Journal of Mechanics and Applied Mathematics* 4.2 (1951), pp. 236–240 (cit. on p. 65).

[7] Nicolas Brunie. "Modified Fused Multiply and Add for Exact Low Precision Product Accumulation". In: *24th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2017 (cit. on pp. 81, 95, 96, 104, 105, 107, 118).

[8] Nicolas Brunie. "Towards the basic linear algebra unit: replicating multi-dimensional FPUs to accelerate linear algebra applications". In: *54th Asilomar Conference on Signals, Systems, and Computers (ASILOMAR)*. IEEE. 2020, pp. 1283–1290 (cit. on pp. 81, 97, 104, 110, 117, 118).

[9] C. Sidney Burrus et al. *Fast Fourier Transforms*. 2008 (cit. on p. 133).

[10] Zachariah Carmichael et al. "Deep Positron: A Deep Neural Network Using the Posit Number System". In: *CoRR* (2018) (cit. on pp. 52, 54).

[11] João P L de Carvalho, José E Moreira, and José Nelson Amaral. "Compiling for the IBM Matrix Engine for Enterprise Workloads". In: *IEEE Micro* 42.05 (Sept. 2022), pp. 34–40 (cit. on p. 45).

[12] Sylvain Chevillard, Mioara Joldeş, and Christoph Lauter. "Sollya: An Environment for the Development of Numerical Codes". In: *Mathematical Software – ICMS*. 2010, pp. 28–31 (cit. on pp. 130, 142, 171).

[13] Maxime Christ, Florent de Dinechin, and Frédéric Petrot. "Low-precision logarithmic arithmetic for neural network accelerators". In: *33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2022 (cit. on p. 52).

[14] Maxime Christ, Luc Forget, and Florent de Dinechin. “Lossless Differential Table Compression for Hardware Function Evaluation”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.3 (Mar. 2022), pp. 1642–1646 (cit. on pp. 141, 142, 148).

[15] Marco Cococcioni et al. “Small Reals Representations for Deep Learning at the Edge: A Comparison”. In: *Conference for Next Generation Arithmetic (CoNGA)*. 2022 (cit. on p. 55).

[16] Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific computing on Itanium-based systems*. 2002 (cit. on pp. 157, 158, 171).

[17] Marius A Cornea-Hasegan, Roger A Golliver, and Peter Markstein. “Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms”. In: *14th Symposium on Computer Arithmetic (ARITH)*. IEEE. 1999, pp. 96–105 (cit. on pp. 158, 159).

[18] Luigi Dadda. “Some Schemes For Parallel Multipliers”. In: *Alta Frequenza* 45.5 (1965), pp. 349–356 (cit. on pp. 60, 65).

[19] D. Das Sarma and D.W. Matula. “Faithful Bipartite ROM Reciprocal Tables”. In: *12th Symposium on Computer Arithmetic (ARITH)*. IEEE, 1995, pp. 17–28 (cit. on p. 143).

[20] Marc Daumas and Guillaume Melquiond. “Certification of bounds on expressions involving rounded operators”. In: *ACM Transactions on Mathematical Software (TOMS)* 37.1 (2010), pp. 1–20 (cit. on p. 171).

[21] Davide De Caro et al. “Minimizing coefficients wordlength for piecewise-polynomial hardware function evaluation with exact or faithful rounding”. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 64.5 (2017), pp. 1187–1200 (cit. on p. 147).

[22] Florent de Dinechin and Arnaud Tisserand. “Multipartite table methods”. In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330 (cit. on pp. 143, 145, 147, 148, 150).

[23] Theodorus Jozef Dekker. “A floating-point technique for extending the available precision”. In: *Numerische Mathematik* 18.3 (1971), pp. 224–242 (cit. on p. 97).

[24] Orégane Desrentes and Florent de Dinechin. “Using integer linear programming for correctly rounded multipartite architectures”. In: *2022 International Conference on Field-Programmable Technology (ICFPT)*. IEEE. 2022, pp. 1–8 (cit. on pp. 148–150, 161, 184).

[25] Orégane Desrentes and Benoît Dupont de Dinechin. “Multiple operand floating point adder with correct rounding”. US20250130768A1. 2023 (cit. on p. 183).

[26] Orégane Desrentes and Benoît Dupont de Dinechin. “Multiplieur d’opérandes à virgule flottante utilisant une décomposition des opérandes en nombres à virgule flottante de plus basse précision”. 2025 (cit. on p. 183).

[27] Orégane Desrentes, Benoît Dupont de Dinechin, and Florent de Dinechin. "Double-Word Decomposition in a Combined FP16, BF16 and FP32 Dot Product Add Operator". In: *32th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2025, pp. 1–8 (cit. on p. 183).

[28] Orégane Desrentes, Benoît Dupont de Dinechin, and Florent de Dinechin. "Exact Fused Dot Product Add Operators". In: *30th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2023, pp. 151–158 (cit. on p. 183).

[29] Orégane Desrentes, Benoît Dupont de Dinechin, and Julien Le Maire. "Exact Dot Product Accumulate Operators for 8-bit Floating-Point Deep Learning". In: *26th EUROMICRO Conference on Digital System Design (DSD)*. IEEE. 2023, pp. 642–649 (cit. on p. 183).

[30] Orégane Desrentes, Diana Resmerita, and Benoît Dupont de Dinechin. "A Posit8 decompression operator for deep neural network inference". In: *Conference on Next Generation Arithmetic (CoNGA)*. Springer. 2022, pp. 14–30 (cit. on pp. 55, 77).

[31] Jérémie Detrey and Florent de Dinechin. "A Parameterized Floating-Point Exponential Function for FPGAs". In: *Field-Programmable Technology*. Singapore, 2005 (cit. on p. 172).

[32] William R Dieter, Akil Kaveti, and Henry G Dietz. "Low-cost microarchitectural support for improved floating-point accuracy". In: *IEEE Computer Architecture Letters* 6.1 (2007), pp. 13–16 (cit. on p. 97).

[33] Florent de Dinechin, Alexey Ershov, and Nicolas Gast. "Towards the post-ultimate libm". In: *IEEE Symposium on Computer Arithmetic (ARITH)*. 2005, pp. 288–295 (cit. on p. 171).

[34] Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic*. 2024 (cit. on pp. 23, 61, 65–68, 92, 129, 143, 170, 172–176, 180).

[35] Florent de Dinechin and Bogdan Pasca. "Designing Custom Arithmetic Data Paths with FloPoCo". In: *IEEE Design & Test of Computers* (2011) (cit. on pp. 109, 128).

[36] Florent de Dinechin and Bogdan Pasca. "Floating-point exponential functions for DSP-enabled FPGAs". In: *Field Programmable Technologies*. Dec. 2010, pp. 110–117 (cit. on pp. 172, 178).

[37] Florent de Dinechin et al. "An FPGA-specific Approach to Floating-Point Accumulation and Sum-of-Products". In: *Field-Programmable Technologies*. 2008, pp. 33–40 (cit. on p. 79).

[38] Florent de Dinechin et al. "Floating-Point Exponentiation Units for Reconfigurable Computing". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 6.1 (2013) (cit. on p. 169).

[39] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. "Activation functions in deep learning: A comprehensive survey and benchmark". In: *Neurocomputing* 503 (2022), pp. 92–108 (cit. on p. 151).

[40] Benoît Dupont de Dinechin, Julien Hascoët, and Orégane Desrentes. "In-Place Multi-Core SIMD FFTs". In: *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2023, pp. 1–6 (cit. on pp. 133, 184).

[41] Benoît Dupont de Dinechin, Julien Le Maire, and Nicolas Brunie. "System for processing matrices using multiple processors simultaneously". 06. US Patent App. 17/566,562. 2022 (cit. on p. 47).

[42] Pedro Echeverría and Marisa López-Vallejo. "An FPGA Implementation of the Powering Function with Single Precision Floating-Point Arithmetic". In: *Real Numbers and Computers*. 2008 (cit. on p. 169).

[43] Pedro Echeverría et al. "An FPGA Run-Time Parameterisable Log-Normal Random Number Generator". In: *Reconfigurable Computing: Architectures, Tools and Applications*. Vol. 4943. 2008, pp. 221–232 (cit. on p. 169).

[44] Thorsten Ehlers and Mike Müller. "New bounds on optimal sorting networks". In: *Evolving Computability: 11th Conf. on Computability in Europe, (CiE)*. 2015, pp. 167–176 (cit. on p. 129).

[45] Milos D Ercegovac. "Radix-16 evaluation of certain elementary functions". In: *IEEE Transactions on Computers* 100.6 (1973), pp. 561–566 (cit. on p. 172).

[46] Massimiliano Fasi et al. "Matrix multiplication in multiword arithmetic: Error analysis and application to GPU tensor cores". In: *SIAM Journal on Scientific Computing* (2023) (cit. on pp. 92, 98, 182).

[47] Massimiliano Fasi et al. "Numerical behavior of NVIDIA tensor cores". In: *PeerJ Computer Science* (2021) (cit. on pp. 95, 182).

[48] Maxim Fishman et al. "Scaling FP8 training to trillion-token LLMs". In: *arXiv preprint arXiv:2409.12517* (2024) (cit. on p. 52).

[49] Robert W Floyd and Donald E Knuth. "The bose-nelson sorting problem". In: *A survey of combinatorial theory*. 1973, pp. 163–172 (cit. on p. 129).

[50] Laurent Fousse et al. "MPFR: A multiple-precision binary floating-point library with correct rounding". In: *ACM Transactions on Mathematical Software (TOMS)* 33.2 (2007), 13–es (cit. on pp. 163, 174).

[51] David Fowler and Eleanor Robson. "Square root approximations in Old Babylonian mathematics: YBC 7289 in context". In: *Historia mathematica* 25.4 (1998), pp. 366–378 (cit. on p. 156).

[52] David Goldberg. "What every computer scientist should know about floating-point arithmetic". In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48 (cit. on p. 35).

[53] IEEE SA P3109 Working Group. *Interim Report on Binary Floating-point Formats for Machine Learning*. https://github.com/P3109/Public/blob/main/IEEE%20WG%20P3109%20Interim%20Report.pdf. Sept. 2023 (cit. on p. 53).

[54] Posit Working Group. *Standard for Posit Arithmetic (2022) Release 5.0*. 2022 (cit. on pp. 52, 54, 81, 83, 85).

[55] John Gustafson and Isaac Yonemoto. “Beating Floating Point at Its Own Game: Posit Arithmetic”. In: *Supercomputing Frontiers and Innovations* (2017) (cit. on p. 54).

[56] Azzam Haidar et al. “Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers”. In: *International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE, 2019 (cit. on p. 99).

[57] John Harrison. “Floating point verification in HOL light: the exponential function”. In: *International Conference on Algebraic Methodology and Software Technology*. 1997, pp. 246–260 (cit. on p. 171).

[58] Greg Henry, Ping Tak Peter Tang, and Alexander Heinecke. “Leveraging the bfloat16 Artificial Intelligence Datatype For Higher-Precision Computations”. In: *26th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2019 (cit. on p. 98).

[59] Brian Hickmann and Dennis Bradford. “Experimental Analysis of Matrix Multiplication Functional Units”. In: *IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019 (cit. on p. 95).

[60] Brian Hickmann et al. “Intel Nervana Neural Network Processor-T (NNP-T) Fused Floating Point Many-Term Dot Product”. In: *27th Symposium on Computer Arithmetic (ARITH)*. 2020 (cit. on p. 95).

[61] W Daniel Hillis and Guy L Steele Jr. “Data parallel algorithms”. In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183 (cit. on p. 125).

[62] Shen-Fu Hsiao et al. “Hierarchical Multipartite Function Evaluation”. In: *Transactions on Computers* 66.1 (2017), pp. 89–99 (cit. on p. 143).

[63] Shen-Fu Hsiao et al. “Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation”. In: *Transactions on Circuits and Systems II* 62.5 (2015), pp. 466–470 (cit. on pp. 141, 143, 145, 147, 148).

[64] Tom Hubrecht, Orégane Desrentes, and Florent de Dinechin. “Activations in Low Precision with High Accuracy”. In: HAL. 2023, pp. 1–8 (cit. on pp. 151, 184).

[65] Tom Hubrecht, Claude-Pierre Jeannerod, and Jean-Michel Muller. “Useful applications of correctly-rounded operators of the form ab + cd + e”. In: *31st Symposium on Computer Arithmetic (ARITH)*. Vol. 31st Symposium on Computer Arithmetic (ARITH). Málaga, Spain, June 2024 (cit. on pp. 134, 183).

[66] IEEE. *754-1985 - IEEE Standard for Binary Floating-Point Arithmetic*. Oct. 1985 (cit. on pp. 33, 36, 52, 170).

[67] IEEE. *754-2008 - IEEE Standard for Binary Floating-Point Arithmetic*. June 2008 (cit. on p. 36).

[68] IEEE. *754-2019 - IEEE Standard for Binary Floating-Point Arithmetic*. June 2019 (cit. on pp. 170, 178).

[69] Intel. *BFLOAT16 – Hardware Numerics Definition Revision 1.0*. Nov. 2018 (cit. on pp. 52, 128).

[70] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International Conference on Machine Learning*. 2015, pp. 448–456 (cit. on p. 44).

[71] C. Iordache and D. W. Matula. "Analysis of Reciprocal and Square Root Reciprocal Instructions in the AMD K6-2 Implementation of 3DNow!" In: *Electronic Notes in Theoretical Computer Science* 24 (1999) (cit. on p. 148).

[72] Matei Istoan and Florent de Dinechin. "Automating the pipeline of arithmetic datapaths". In: *DATE*. Mar. 2017 (cit. on p. 186).

[73] Qiwei Jin et al. "Multi-Level Customisation Framework for Curve Based Monte Carlo Financial Simulations". In: *International Symposium on Applied Reconfigurable Computing*. 2012, pp. 187–201 (cit. on p. 169).

[74] Jeff Johnson. "Rethinking floating point for deep learning". In: *CoRR* (2018) (cit. on p. 52).

[75] Himanshu Kaul et al. "Optimized Fused Floating-Point Many-Term Dot-Product Hardware for Machine Learning Accelerators". In: *IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019 (cit. on p. 95).

[76] Donghyun Kim and Lee-Sup Kim. "A floating-point unit for 4D vector inner product with reduced latency". In: *IEEE Transactions on computers* 58.7 (2008), pp. 890–901 (cit. on p. 95).

[77] Nick G Kingsbury and Peter JW Rayner. "Digital filtering using logarithmic arithmetic". In: *Electronics letters* 7.2 (1971), pp. 56–58 (cit. on p. 52).

[78] Reinhard Kirchner and Ulrich Kulisch. "Accurate arithmetic for vector processors". In: *Journal of parallel and distributed computing* (1988) (cit. on pp. 79, 181).

[79] Andreas Knofel. "Fast hardware units for the computation of accurate dot products". In: *10th IEEE Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society. 1991, pp. 70–71 (cit. on p. 79).

[80] Ulrich Kulisch and Van Snyder. "The Exact Dot Product as Basic Tool for Long Interval Arithmetic". In: *Computing* 91.3 (Mar. 2011), pp. 307–313 (cit. on p. 79).

[81] Martin Kumm and Johannes Kappauf. "Advanced Compressor Tree Synthesis for FPGAs". In: *IEEE Transactions on Computers* 67.8 (2018), pp. 1078–1091 (cit. on pp. 61, 65, 85).

[82] Hsiang Tsung Kung and Charles E Leiserson. "Systolic arrays (for VLSI)". In: *Sparse Matrix Proceedings*. Vol. 1. Society for industrial and applied mathematics. 1979, pp. 256–282 (cit. on p. 45).

[83] Andrey Kuzmin et al. "FP8 Quantization: The Power of the Exponent". In: 2022 (cit. on pp. 52, 53).

[84] Vincent Lefèvre and Jean-Michel Muller. “Worst cases for correct rounding of the elementary functions in double precision”. In: *IEEE Symposium on Computer Arithmetic (ARITH)*. IEEE. 2001, pp. 111–118 (cit. on p. 171).

[85] Ren-Cang Li et al. *The Libm library and floating-point arithmetic for HP-UX on Itanium*. Tech. rep. Technical report, Hewlett-Packard company, 2001 (cit. on pp. 161, 171).

[86] Aixin Liu et al. “Deepseek-v3 technical report”. In: *arXiv preprint arXiv:2412.19437* (2024) (cit. on p. 52).

[87] Jinming Lu et al. “Evaluations on deep neural networks training using posit number system”. In: *IEEE Transactions on Computers* (2020) (cit. on pp. 52, 54, 55, 77).

[88] David R. Lutz et al. “Fused FP8 4-Way Dot Product With Scaling and FP32 Accumulation”. In: *31st Symposium on Computer Arithmetic (ARITH)*. IEEE, June 2024 (cit. on pp. 97, 105, 182).

[89] Albert Paul Malvino and David J Bates. *Principes d’électronique-7e éd.: Cours et exercices corrigés*. 2008 (cit. on p. 8).

[90] Stefano Markidis et al. “NVIDIA Tensor Core Programmability, Performance & Precision ”. In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2018 (cit. on pp. 63, 95, 98, 99).

[91] Paulius Micikevicius et al. “FP8 Formats for Deep Learning”. In: *arXiv preprint arXiv:2209.05433*. 2022 (cit. on p. 52).

[92] Mantas Mikaitis. “Monotonicity of multi-term floating-point adders”. In: *IEEE Transactions on Computers* (2024) (cit. on p. 182).

[93] Jean-Michel Muller. “A Few Results on Table-Based Methods”. In: *Reliable Computing* 5.3 (1999), pp. 279–288 (cit. on p. 143).

[94] Jean-Michel Muller. *Elementary functions*. 2006 (cit. on pp. 140, 172, 177).

[95] Jean-Michel Muller. *On the definition of ulp (x)*. Research report for INRIA, LIP. 2005 (cit. on p. 38).

[96] Jean-Michel Muller et al. *Handbook of Floating-Point Arithmetic, 2nd edition*. 2018 (cit. on pp. 35, 92, 97, 121).

[97] Raul Murillo, Alberto A Del Barrio, and Guillermo Botella. “Customized posit adders and multipliers using the FloPoCo core generator”. In: *International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2020, pp. 1–5 (cit. on p. 77).

[98] Raul Murillo, Alberto A Del Barrio, and Guillermo Botella. “Deep PeNSieve: A deep learning framework based on the posit number system”. In: *Digital Signal Processing* (2020) (cit. on pp. 52, 55).

[99] Badreddine Noune et al. “8-bit Numerical Formats for Deep Neural Networks”. In: *arXiv preprint arXiv:2206.02915*. 2022 (cit. on p. 53).

[100] Ian Parberry. “A computer-assisted optimal depth lower bound for nine-input sorting networks”. In: *Mathematical systems theory* 24.1 (1991), pp. 101–116 (cit. on p. 129).

[101] J-A Pineiro, Javier D Bruguera, and Milos D Ercegovac. "On-line high-radix exponential with selection by rounding". In: *International Symposium on Circuits and Systems (ISCAS)*. Vol. 4. IEEE. 2003, pp. IV–IV (cit. on p. 172).

[102] J-A Pineiro, Milos D Ercegovac, and Javier D Bruguera. "Algorithm and architecture for logarithm, exponential, and powering computation". In: *IEEE Transactions on Computers* 53.9 (2004), pp. 1085–1096 (cit. on pp. 169, 172).

[103] Open-Compute Project. *OCP 8-bit Floating Point Specification (OFP8)*. https://www.opencompute.org/documents/ocp-8-bit-floating-point-specification-ofp8-revision-1-0-2023-12-01-pdf-1. June 2023 (cit. on pp. 52, 53, 83).

[104] Open-Compute Project. *OCP Microscaling Formats (MX) Specification*. https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf. Sept. 2023 (cit. on pp. 53, 182).

[105] Dary Mochamad Rifqie et al. "Post Training Quantization in LeNet-5 Algorithm for Efficient Inference". In: *Journal Of Embedded System Security and Intelligent System (JESSI)* (2022), pp. 60–64 (cit. on p. 55).

[106] Hani H Saleh and Earl E Swartzlander. "A floating-point fused dot-product unit". In: *International Conference on Computer Design*. IEEE. 2008, pp. 427–431 (cit. on p. 95).

[107] M.J. Schulte and J.E. Stine. "Approximating Elementary Functions with Symmetric Bipartite Tables". In: *IEEE Transactions on Computers* 48.8 (1999), pp. 842–847 (cit. on pp. 143, 147).

[108] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. "The CORE-MATH Project". In: *29th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2022, pp. 26–34 (cit. on p. 171).

[109] Richard C. Singleton. "On Computing the Fast Fourier Transform". In: *Journal of the ACM* 10.10 (1967) (cit. on p. 133).

[110] Rys Sommefeldt. *Origin of Quake3's Fast InvSqrt()*. https://www.beyond3d.com/content/articles/8/. Accessed: 2025-06-18. 2006 (cit. on p. 159).

[111] J.E. Stine and M.J. Schulte. "The Symmetric Table Addition Method for Accurate Function Approximation". In: *Journal of VLSI Signal Processing* 21.2 (1999), pp. 167–177 (cit. on p. 143).

[112] Xiao Sun et al. "Hybrid 8-Bit Floating Point (HFP8) Training and Inference for Deep Neural Networks". In: *33rd International Conference on Neural Information Processing Systems (NIPS)*. 2019 (cit. on pp. 52, 74, 88).

[113] D. A. Sunderland et al. "CMOS/SOS frequency synthesizer LSI circuit for spread spectrum communications". In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 497–506 (cit. on p. 143).

[114] Earl E. Swartzlander. “Merged Arithmetic”. In: *IEEE Transactions on Computers* C-29.10 (1980), pp. 946–950 (cit. on pp. 60, 65).

[115] Ping-Tak Peter Tang. “Table-driven implementation of the exponential function in IEEE floating-point arithmetic”. In: *ACM Transactions on Mathematical Software (TOMS)* 15.2 (1989), pp. 144–157 (cit. on p. 170).

[116] Yao Tao et al. “Correctly rounded architectures for floating-point multi-operand addition and dot-product computation”. In: *24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE. 2013, pp. 346–355 (cit. on pp. 117, 119, 129, 130, 181).

[117] Manfred Tasche and Hansmartin Zeuner. “Improved Roundoff Error Analysis for Precomputed Twiddle Factors”. In: *Journal of Computational Analysis and Applications* 4 (Jan. 2002), pp. 1–18 (cit. on p. 133).

[118] MT Tommiska. “Efficient digital implementation of the sigmoid function for reprogrammable logic”. In: *IEEE Proceedings-Computers and Digital Techniques* 150.6 (2003), pp. 403–411 (cit. on p. 141).

[119] Yohann Uguen, Luc Forget, and Florent de Dinechin. “Evaluating the hardware cost of the posit number system”. In: *29th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2019, pp. 106–113 (cit. on p. 77).

[120] Pedro Valero-Lara et al. “Mixed-Precision S/DGEMM Using the TF32 and TF64 Frameworks on Low-Precision AI Tensor Cores”. In: *International Conference on High Performance Computing, Network, Storage, and Analysis*. ACM, 2023, pp. 179–186 (cit. on p. 99).

[121] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in Neural Information Processing Systems (NIPS)* (2017) (cit. on pp. 41–43).

[122] Álvaro Vázquez and Javier D Bruguera. “Iterative algorithm and architecture for exponential, logarithm, powering, and root extraction”. In: *IEEE Transactions on Computers* 62.9 (2012), pp. 1721–1731 (cit. on p. 172).

[123] Naigang Wang et al. “Training Deep Neural Networks with 8-Bit Floating Point Numbers”. In: *32nd International Conference on Neural Information Processing Systems (NIPS)*. 2018 (cit. on p. 52).

[124] Wm A Wulf and Sally A McKee. “Hitting the memory wall: Implications of the obvious”. In: *ACM SIGARCH computer architecture news* 23.1 (1995), pp. 20–24 (cit. on p. 45).

[125] Ron Zeno. *A reference of the best-known sorting networks for up to 16 inputs*. https://www.angelfire.com/blog/ronz/Articles/999SortingNetworksReferen.html. Accessed: 2023-04-04. 2002 (cit. on p. 129).

[126] Biao Zhang and Rico Sennrich. “Root mean square layer normalization”. In: *Advances in Neural Information Processing Systems* 32 (2019) (cit. on p. 44).

[127] A. Ziv. “Fast evaluation of elementary mathematical functions with correctly rounded last bit”. In: *ACM Transactions on Mathematical Software* 17.3 (1991), pp. 410–423 (cit. on p. 171).

FOLIO ADMINISTRATIF

THESE DE L'INSA LYON, MEMBRE DE L'UNIVERSITE DE LYON

NOM : **Desrentes** DATE de SOUTENANCE : **17 septembre 2025**

Prénoms : **Orégane**

TITRE : **Accélération Matérielle d'Arithmétique pour l'Apprentissage Artificiel et le Calcul Scientifique**

NATURE : **Doctorat** Numéro d'ordre : **2025ISAL0079**

École Doctorale : **Informatique et Mathématiques**

Spécialité : **Informatique**

RÉSUMÉ :
Dans un monde axé sur les données, l'apprentissage artificiel et le calcul scientifique sont devenus de plus en plus importants, justifiant l'utilisation d'accélérateurs matériels dédiés. Cette thèse explore la conception et l'implémentation d'unités arithmétiques pour de tels accélérateurs dans le Massively Parallel Processor Array de Kalray.
L'apprentissage artificiel nécessite des multiplications de matrices qui opèrent sur des formats de nombres très petits. Dans ce contexte, cette thèse étudie l'implémentation du produit-scalaire-et-addition en précision mixte pour différents formats de 8 et 16 bits (FP8, INT8, Posit8, FP16, BF16), en utilisant des variantes d'une technique classique de l'état-de-l'art, l'accumulateur long. Elle introduit également des techniques permettant de combiner différents formats d'entrée. Des méthodes radicalement différentes sont étudiées pour passer à l'échelle vers la grande dynamique des formats 32 et 64 bits utilisés en calcul scientifique.
Cette thèse étudie également l'évaluation de certaines fonctions élémentaires. Un opérateur pour la fonction exponentielle (cruciale pour le calcul de la fonction softmax) étend une architecture de l'état-de-l'art pour accepter des formats d'entrée multiples. La fonction racine carrée inverse (utilisée pour la normalisation des couches) est accélérée en combinant des techniques d'état-de-l'art pour la réduction de la dynamique, des tables multipartites correctement arrondies et des techniques logicielles de raffinement itératif.

MOTS-CLÉS : **architecture des ordinateurs, arithmétique des ordinateurs, nombre flottant, produit scalaire, fonctions élémentaires**

Laboratoire(s) de recherche : **CITI**

Directeur de thèse : **Florent de Dinechin**

Président du Jury : **Jean-Michel Muller**

Composition du Jury :

Javier Diaz Bruguera, Senior Principal Design Engineer, ARM
Olivier Sentieys, Professeur des Universités, Université de Rennes
Roselyne Chotin, Professeure des Universités, Sorbonne Université
Caroline Collange, Chargée de Recherche, INRIA de Rennes
Florent De Dinechin, Professeur des Universités, INSA de Lyon
Benoît Dupont de Dinechin, Directeur de la technologie, Kalray S.A.
Jean-Michel Muller, Directeur de Recherche, CNRS